Uporaba okvira za razvoj igara kao motivacija pri učenju

Asanović, Branimir

SVEUČILIŠTE U SPLITU
PRIRODOSLOVNO MATEMATIČKI FAKULTET

DIPLOMSKI RAD

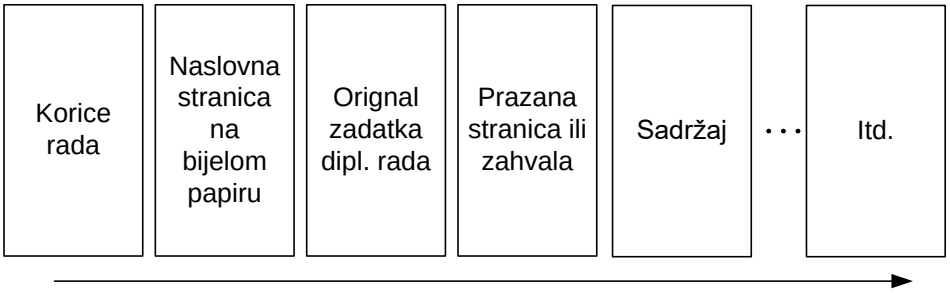
UPORABA OKVIRA ZA RAZVOJ IGARA KAO MOTIVACIJA PRI UČENJU

Branimir Asanović

Split, mjesec 2014.

Prilikom uvezivanja rada iza ove stranice ne zaboravite umetnutnuti original diplomskog zadatka kojeg ste preuzeli od mentora diplomskog rada.

Redoslijed prvih nekoliko stranica



Sažetak

Motivaciju možemo smatrati jako bitnim faktorom koji utječe na uspjeh individue u bilo kojem znanstvenom području pa tako i području računalne znanosti. U fokusu istraživanja su motivacija i način na koji se ista pokušava pobuditi u studentima. Osnovni cilj ovoga istraživanja je pobuditi motivaciju studenata preko projektne nastave i to izradom vlastitih video igara ili video igara izrađenih od strane više profesionalnih programera. Za izradu igara koristili su se odgovarajući okviri koji imaju ugrađene funkcionalnosti i biblioteke za razvoj igara. Pisani rad nudi nekakav uvod u ovo, još novo, područje istraživanja koje uključuje motivaciju, objektno-orijentirano programiranje, poteškoće programera početnika, video igre i okruženja za izradu video igara. Kroz rad dan je pregled sličnih istraživanja iz ovoga područja te su dani rezultati provedenog istraživanja gdje se može vidjeti koliki je interes studenata za učenjem programiranja izradom igara te koliko su truda isti studenti voljni uložiti za učenje istoga.

Summary

Motivation can be considered as crutional factor which has grat impact on success of individual in computer science just like in every other scientific area. Main focus of this resarch is motivation and ways to motivate students. Primary objective of this resarch is to boost student motivation through project based learning where students build their own games or learn from other games made by more advanced programmers. To build games students could use several computer game frameworks with existing libraries and functionalities. This paper can be considered as some kind of an introduction to this somehow new area of research which involves motivation, object-oriented programming, difficulties of novice programmers, video games and game engines. While the paper inclues it's own resarch that clearly shows student interests in making video games and amount of effort they are willing to put in their work, it also includes overview of other, similar, researches that give some interesting results.

Sadržaj

Uvod

Osnovni problem ovoga rada je nemotiviranost studenata na nastavni programiranja. To nije problem samo ovog rada nego i poučavanja programiranja [1] kao i ostalih predmeta iz STEM područja (eng. *Science, technology, engineering and mathematics*). Motivacija uz privrženost i samopouzdanje vezana je za neuspjeh u nastavi, a u konačnici i odustajanje od studija. Isto tako motivaciju se smatra i faktorom uspjeha u visokom obrazovanju [1].

Naravno, problem motivacije ne možemo navesti kao jedini problem u nastavi programiranja. Postoje mnoge miskoncepcije u nastavi programiranja posebno kod programera početnika što je dio ovoga rada.

Način na koji ovaj rad pristupa navedenim problemima su igre u nastavi, točnije izradom igara uz pomoć adekvatnih okvira za izradu igara (eng. *Game engine*).

Kroz ovaj rad nastojati ćemo povezati pojmove poput motivacije, programiranja, programskih paradigmi i računalnih igara u svrhu upoznavanja s područjem istraživanja ovoga rada.

U istraživanje uključeni su studenti Prirodoslovno-Matematičkog fakulteta u Splitu koji su imali na raspolaganju dva okvira za izradu igara. Svi rezultati ovog istraživanja temelje se na analizi projekata koje su studenti izradili za nastavni predmet Objektno orijentirano programiranje.

1. Motivacija u nastavi

Učenje programiranja je samo po sebi problem jer je ono vještina teška za steći. Da bi studenti uspješno pristupili programiranju potrebno ih je motivirati [2]. Motivaciju možemo promatrati kao pojam iz dvije perspektive. U prvom slučaju govorimo o atributu studenta koji ga potiče na rad i učenje dok u drugom slučaju govorimo o ulozi učitelja kao onoga tko motivira studente [3]. Motivacija je jako apstraktan koncept za koji ne postoji pravo mjerilo [4]. Ovaj rad ne pokušava stvoriti mjerilo za motivaciju nego pokušati dokučiti je li kod studenata postojao bilo kakav oblik motivacije.

Entwistle [4] definira tri generička tipa motivacije:

- Ekstrinzična – želja za polaganjem kolegija zbog neke očekivane nagrade
- Intrinzična – pobuđuje se interesima subjekta
- Dostignuće – potreba da „dobro radimo/učimo“ i (ponekad) da budemo bolji od vršnjaka

Primarno, studenti su motivirani jednim od ova tri navedena tipa, a nas najviše zanima intrinzična motivacija.

Fallows i Ahmet [5] pristupaju razlozima zašto studenti pridodaju vrijednost učenju te predlažu sljedeće:

- Učenikova želja da ugodi učitelju
- Potreba za razumijevanjem prezentiranoga materijala
- Pojedinačni stupanj interesa za temu prezentiranoga materijala
- Vlastite filozofske vrijednosti i vjerovanja
- Učenikov stav prema prezentiranom materijalu
- Akademske i poslovne čežnje učenika
- Očekivani poticaji i nagrade kao rezultat učenja

U konačnici Jenkins u svom radu [3] navodi, uz Entwistle-ove tipove motivacije, dodatna dva tipa:

- Socijalna – primarna motivacija je potreba za udovoljavanjem osobi čije se mišljenje cijeni

- Ništa (eng. *null*) – nema motivacije (negativni pogledi koji se karakteriziraju izjavom „Samo želim proći kolegij“)

Da bi studenti naučili programirati oni moraju biti motivirani, tj. programiranju moraju pridijeliti neku vrstu vrijednosti [3]. Osim što im to može imati neku vrijednost, studentima bi to što rade trebalo biti i zanimljivo.

1.1. Motivacija i stavovi o programiranju

Dio studenata je nemotiviran i to je činjenica, ali isto tako učenici koji tek započinju obrazovanje su jako motivirani. Učenici su puni života i znatiželje jer kroče u nešto novo, nepoznato, a s vremenom ta vedrina opada i učiteljima ponekad nije jasno što je krenulo u krivom smjeru [6].

Postoji više faktora koji pripomažu gubitku interesa za računalnom znanošću. Zanimljiv je podatak da studenti smatraju uvodne kolegije neinspirirajućim zbog zadataka koji su im predstavljeni, kao npr. „sortiraj listu brojeva“ ili „zbroji sve cijele brojeve do n“ [7]. Danas je računalna znanost jedna velika i razvijena grana te se iz dana u dan stvara sve više i više radnih mjesta. Računalna znanost nije samo „sortiranje brojeva“ i sl., ona je jedno jako zanimljivo područje. Jedan od faktora koji nas može privući u računalnu znanost su igre. U zadnje vrijeme pokazan je interes za ulaganje državnih sredstava u korištenje računalnih igara za privlačenje novih lica u računalnu znanost. Vjeruje se kako će današnje i nove generacije biti više zainteresirane i motivirane pristupom koji koristi računalne igre nego li tradicionalnim uvodnim kolegijima računalne znanosti [7].

Ne navodimo kako su igre idealan način za podizanje motivacije kod svih učenika. Uvijek postoji onaj faktor individualnih interesa i međugeneracijskih razlika. Npr. nove generacije odrastaju na igrama stoga njima možda bude jako zanimljivo izrađivati igru dok za starije generacije to ne mora biti slučaj.

Kroz nekolicinu radova pokazano je kako postoji manji broj žena u području računalne znanosti i to zbog stereotipa o području. U ovom području stereotipi dominacije muškaraca puno su jači od drugih područja poput matematike i kemije [8]. Istraživanja pokazuju kako i muškarci i žene jednake interese za matematiku i ostale grane znanosti [8]. Isto vrijedi i za prva iskustva korištenja računala, upoznatost s područjem računalne znanosti i sl.

Korištenjem okvira za programiranje igara pripovijedanjem priča za izradu animiranih filmova uspjelo se povećati interes za programiranjem kod djevojaka koje pohađaju srednju školu [7] tako da se tim istraživanjem pobijaju neki od mogućih stereotipa.

Predrasude i različiti stereotipi o pojedinim stvarima su svuda oko nas pa tako i u području programiranja. Već je spomenuto kako djeca u škole stupaju puni vedrine i želje za znanjem. Isto tako među odraslima postoje negativni pogledi prema programiranju [11]. Ovakvi pogledi su izraženiji kod pripadnica ženskog spola.

Neki vjeruju kako je programiranje dosadno i preteško za naučiti, vežu ga s visokom inteligencijom, logičkim razmišljanjem i genijima za koje se oni sami ne smatraju. Vjeruje se kako programerima nedostaje interpersonalnih vještina, da osjećaju socijalnu neugodu te da su opsjednuti tehnologijom jer nemaju drugih interesa.

Ovi stereotipi mogu stvoriti negativne stavove prema programiranju, obeshrabrujući individue da krenu učiti kako programirati i prihvaćati programiranje kao nužnu vještinu za njihov profesionalni razvoj. Negativni stavovi poput ovih mogu se promijeniti kada individua doživi vlastito iskustvo učenja programiranja [11]. Ako motivacija za učenje može održati pozitivne stavove odraslih prema programiranju, onda njihovi stavovi mogu utjecati na motivaciju njihove djece [11]. Isto tako možemo povući paralelu s budućim nastavnicima. Ako budući nastavnici imaju pozitivne stavove prema programiranju, onda će te iste stavove prenositi i na vlastiti razred stoga je potrebno graditi takve stavove kroz obrazovanje budućih nastavnika.

1.2. Projektna nastava kao odgovor na nemotiviranost

Jedan od načina stvaranja toga nečega „zanimljivoga“ što može podići motivaciju studenata je projektna nastava [9]. Projekta nastava je nastavna metoda u kojoj učenici stječu znanja i vještine kroz duži vremenski period istražujući i rješavajući kompleksne probleme.

Projekti trebaju biti bazirani na stvarnim problemima tako da stečeno iskustvo bude korisno za učenika i društvo. Za razliku od klasične, predavačke, nastave, učitelj ima ulogu savjetnika ili tutora koji učenike vodi prema cilju, na temelju plana kojeg su zajedno izradili. Kroz projektnu nastavu učenik se aktivno suočava s problemom iz čega proizlaze pitanja na koja učenik želi saznati odgovor. Kroz projektnu nastavu ohrabruje se

raznolikost ideja i postupaka kojima učenik pristupa nekom problemu. Isto tako projektna nastava pruža široku lepezu tema na kojima učenici mogu raditi jer su sve teme preslika iz stvarnoga života.

Studenti su spremni uložiti dodatne napore i vrijeme na svojim projektima bez da se to od njih traži te u konačnici studenti sami favoriziraju ovakav tip nastave [10]. Svaki budući učitelj će građenjem vlastitog iskustva radom u odgojno-obrazovnim ustanovama susretati se s motiviranim i nemotiviranim učenicima. Ti nemotivirani učenici će vjerojatno ulagati onoliko napora koliko im je potrebno da dobiju pozitivnu ocjenu, a na to bi svaki učitelj trebao reagirati te istome učeniku dati šansu da pokaže ono što njega kao pojedinca zanima te koliko je spreman raditi na tome. U tome slučaju projektna nastava može priskočiti u pomoć.

U projektnoj nastavi postoje dvije esencijalne komponente. Prva komponenta je pitanje ili problem koje služi za organizaciju i pokretanje aktivnosti za rješavanje tog problema ili odgovaranja na pitanje. Druga komponenta su aktivnosti koje u konačnici rezultiraju nekim produktom. Ove dvije komponente može definirati i organizirati učitelj, a isto tako i sam učenik [9]. Ovo je jako bitno jer se na ovaj način učeniku pruža velika sloboda za odabir problema (teme projekta) za koji je on sam zainteresiran. Odabirom učeniku interesantne teme smo korak bliže onoj situaciji gdje učenik ulaže puno više vremena i truda za realizaciju vlastitog projekta. Time učenik može sam organizirati vlastito vrijeme i aktivnosti. Međutim, u nekim slučajevima ovo sloboda može biti pogubna jer se učenik može naći u situaciji gdje je njegov problem nerazrješiv. Prema tome uvijek je dobro da učitelj u određenom postotku bude upleten u odabir problema i planiranja aktivnosti. Isto tako, u oba slučaja pitanja ili problemi ne mogu biti toliko ograničeni da će ukupni rezultat biti predefiniran ostavljajući učenike s malo prostora za razvoj vlastitih postupaka za traženje odgovora na pitanje ili načina razrješavanja problema.

Postoji veliko bogatstvo u dobro osmišljenim projektima koje može biti iskorišteno od strane učitelja i učenika. Projekti mogu povećati interes učenika jer se radi o autentičnim problemima koji ih vade iz konteksta škole i umjetno stvorenog sustava gdje je bitno dobiti što bolju ocjenu. Također, projekti mogu pridonositi razvoju socijalnih i metakognitivnih vještina.

Postoji niz faktora koje trebamo uzeti u obzir kada govorimo o podizanju motivacije učenika projektnom nastavom. Neki od faktora su već spomenuti, a to su vrijednost i

zanimljivost koju učenici pridodaju projektima. Ostali faktori mogu biti i sposobnost da se završi projekt određene težine, fokusiranost na to što žele naučiti naspram toga koju će nagradu ili ocjenu dobiti i konačno individualne razlike među učenicima.

Učenici će češće biti motivirani projektima, ali ipak i projektna nastava ima svoje nedostatke [9]. Projektna nastava nije nešto novo, njezini korijeni sežu još do Dewey-a. Još 1960. godine unatoč pozitivnim rezultatima poboljšanja učenja učenika i motivacije istih, globalna adaptacija promjena kurikuluma se nije dogodila kako se očekivalo.

2. Programske paradigme

Kroz povijest programiranja razvijeno je mnogo programskih paradigmi njihovih podvrsta, a ovdje ćemo spomenuti samo osnovne paradigme. Paradigma je ništa drugo nego način na koji radimo i mislimo o stvarima. Prema tome, programska paradigma je način na koji programer pristupa rješavanju nekog problema. Problemima možemo pristupiti iz različitih perspektiva te uvijek odabiremo onu koja se nama kao programeru čini najbolja ili najprimjerenija za taj tipični problem.

Neke osnovne programske paradigme su:

- Imperativno (proceduralno) programiranje,
- Deklarativno (logičko i funkcijsko) programiranje
- Objektno-orijentirano programiranje (eng. *Object-oriented programming*, skraćeno OOP)

Još neke od paradigmi navodimo kao simboličko programiranje, događajima pokretano programiranje, paralelno programiranje, agentski-orijentirano programiranje…

2.1. Imperativno programiranje

Imperativno programiranje najkraće možemo opisati frazom: „Prvo napravi ovo, a zatim napravi ono“. Imperativno programiranje temelji se na promjenjivim stanjima programa. U ovoj paradigmi najčešće govorimo o nizu naredbi (eng. *statement*) koje mijenjaju stanje programa, a tok izvođenja naredbi može biti kontroliran kontrolnim strukturama*: if-then-else, petlje (npr. while i for), break, continue*…

Proširenje ove paradigme nazivamo proceduralnim programiranjem. Za rješenje problema, ova paradigma koristi se promjenjivim strukturama podataka i potprogramima. Iz samoga naziva možemo zaključiti kako se paradigma temelji na skupu funkcionalnih dijelova programa zvanih „procedura“ (pod „procedurama“ podrazumijevamo sve vrste potprograma, procedure i funkcije).

Svaki problem, ovom paradigmom, rješavamo tako da ga rastavimo na podprobleme koji se opisuju pojedinim procedurama. Sve te procedure su međusobno zavisne te čine jednu

veću cjelinu (program) koja zapravo predstavlja rješenje zadanog problema. Rastavljanjem programa na potprograme stvaramo mogućnost ponovne upotrebe nekih od potprograma iako to može biti nemoguće kod velikih programa gdje se međusobna zavisnost teško uočava. Također, i najmanja promjena u programu može rezultirati rušenjem cijelog programa, a nepažnja pri rukovanjem dinamičkim strukturama podataka može izazvati neprimjetno „curenje memorije“.

2.2. Deklarativno programiranje

Deklarativnim programiranjem, nasuprot imperativnog zadajemo što se treba izračunati umjesto kako nešto treba izračunati. Sav račun se prepušta računalu što znači da je programer ne opisuje račun kontrolom toka naredbi.

Logičko i funkcijsko programiranje navodimo kao dvije podvrste deklarativnih paradigmi. U logičkim programskim jezicima, program se sastoji od logičkih izjava, a program prema tim izjavama traži dokaze za te izjave.

Za razliku od imperativnog programiranja, u funkcijskom programiranju nisu dozvoljene promjenjive varijable, pridruživanja, kontrolne strukture i sl. Svo programiranje vrši se korištenjem funkcija. Ova paradigma proizlazi iz matematičke discipline zvane „teorija funkcija“ što je čini puno čišćom i jednostavnijom paradigmom od imperativne. U funkcijskom programiranju računanje se opisuje kao evaluacija matematičkih funkcija, a rezultirajuća vrijednost ima svoju svrhu i upotrebu.

2.3. Objektno-orijentirano programiranje

Ova paradigma je bazirana na konceptu „objekta“. Objekt je jedan tip strukture podataka koju opisujemo atributima i metodama. Atributi su podatci koji opisuju objekt tj. osobine objekta dok su metode funkcionalni dio objekta koji možemo opisati kao ponašanje objekta. U OO paradigmi sve se smatra objektom, a tok programa određen je međusobnom komunikacijom među objektima. Primjer objekta može biti bilo koji čovjek, vozilo, neka pojava i sl. Svi ti objekti su opisani klasama. Klasa je također jedan od osnovnih koncepata OO paradigme. Klasom opisujemo skup objekata koji imaju neka zajednička obilježja, ponašanja. Npr. nećemo za svakog čovjeka raditi posebnu klasu, nego ćemo sve ljude opisati jednom klasom, a pojedina instanca te klase će biti jedna osoba koja ima vlastiti

identitet. Jedna od najbitnijih karakteristika OO paradigme je mogućnost preslike stvarnosti tj. problema iz svakodnevnog života u računalo.

Objektno orijentirano programiranje je danas najzastupljenije u softverskoj industriji zbog svih svojih prednosti koje je donijelo sa sobom. Objektno-orijentirano programiranje omogućilo je da programeri lakše i brže pišu programe koji su jako dobro organizirani. Također, brzina izvođenja programa je jako velika, a kao jedna od najbitnijih prednosti je mogućnost ponovnog korištenja koda (eng. *reusability*). U starim jezicima ako smo htjeli napraviti dvije slične stvari koje su dijelile dosta istih svojstava trebali smo ili mijenjati postojeći kod ili pisati novi koji je skoro identičan. To upućuje na ponavljanje istog programskog koda na više mjesta. Posljedica toga je nepotrebna nagomilanost programskog koda. U novijim jezicima kao C#, stari kod ne trebamo ponovno pisati, a nadograđivanjem ne utječemo na taj isti stari kod i ne unosimo mogućnost pojavljivanja pogreški (eng. *bugg*) unutar koda koji se do trenutka nadogradnje točno, bez grešaka, izvodio.

Za istraživanje odabran je kolegij objektno-orijentiranog programiranja i to iz razloga navedenih u prethodnom paragrafu.

Unatoč svim pozitivnim aspektima OO paradigme, postoje mnogi problemi s kojima se programeri početnici suočavaju. Mnogi OO koncepti odišu velikim stupnjem apstrakcije te stoga programerima početnicima može zadavati velike probleme.

Anna Eckerdal u svome radu [12] govori o problematici kod učenja osnovnih koncepata OOP-a. Kroz istraživanje klasificiraju se viđenja studenata o tome što su to objekt i klasa te koja je njihova uloga.

U prvoj kategoriji studenti objekt i klasu najviše vežu za programski kod i strukturu programa:

- Objekt studenti doživljavaju kao dio programskog koda
- Klasu studenti doživljavaju kao dio programa koji pridonosi strukturi programskog koda

U drugoj kategoriji su ti koncepti doživljeni kao nešto aktivno u programu:

- Objekt studenti doživljavaju kao nešto aktivno unutar programa
- Klasu studenti doživljavaju kao opis atributa i ponašanja objekta

Treća kategorija odnosi se na povezivanje koncepata s fenomenima iz stvarnog svijeta gdje:

- Studenti klasu doživljavaju kao opis atributa i ponašanja objekta kao modela fenomena iz stvarnog svijeta
- Studenti objekt doživljavaju kao model nekog fenomena iz stvarnog svijeta

Ove kategorije govore mnogo o razumijevanju studenata u području objektno-orijentiranog programiranja. Za oba koncepta treća kategorija pokazuje najbogatije shvaćanje istih jer za pravilno razumijevanje tih koncepata potrebno je gledati na programiranje iz konteksta koji nadilazi sam programski kod i sintaksu programskog jezika.

Programiranje je ništa drugo nego način na koji čovjek stvarna iskustva sa svijetom prenosi u računalo pa prema tome definiraju se još 3 dodatne kategorije studentskih odgovora za svrhu korištenja objekta i klase.

1. Kodna perspektiva: sintaksa zahtjeva korištenje tih koncepata jer daju programu dobru strukturu
2. Perspektiva korisnika i rezultata: razlog korištenja koncepata je olakšati programeru rješavanje problema
3. Perspektiva realnosti: razlog korištenja koncepata je da bi se stvarnost bolje spojila s programiranjem

Ponovno, treća kategorija govori najviše o razini apstrakcije i razumijevanja koncepata studenta te prema tome treba težiti pri podučavanju OOP-a. Najmanji broj studenata spada u treću kategoriju jer im je teško povezati da je programiranje refleksija stvarnosti, a da su svi OOP koncepti tu da nam osiguraju što bolje preslikavanje iste.

2.4. Multi-paradigmatsko programiranje

U današnje vrijeme mnogi programski jezici podržavaju više programskih paradigmi kao npr. objektno-orijentiranu paradigmu u kombinaciji s imperativnom i proceduralnom paradigmom. Neki od tih programskih jezika su Python, C++, C#, Java, Ruby, PHP, itd.

3. Poteškoće programera početnika

Postoji niz radova koji govore o problematici podučavanja programera početnika i miskoncepcijama s kojima se početnici susreću. Cilj nastavnika je da zapazi moguće miskoncepcije te pravilno reagirati u svrhu ispravljanja istih. U nastavku su navedeni neki od tipičnih pogrešaka kod programera početnika općenito.

3.1. Općenito o pogreškama

Pogreške možemo podijeliti u dvije kategorije, konstruktivno temeljene i nekonstruktivno temeljene.

Nekonstruktivno temeljene pogreške su pogreške koji nisu nastale zbog pogrešnog shvaćanja semantike jezika, a to su problem zbrajanja, optimizacije i problem prošlih iskustava. Tipični problemi programera početnika su granični problem (eng. *boundary problem*) i problem negacije (eng. *negation and whole-part problem*). Granične probleme svrstavamo u NO kategoriju dok problem negacije u MAYBE kategoriju. Primjer graničnog problema je „*off-by-one bug*“ (npr. kada pristupamo elementu niza van granica), a primjer problema negacije je kada učenik zaključi da je negacija od „dobre stvari“ „loša stvar“ kao npr. logički OR izraz je suprotno od logičkog AND izraza. Postoji još jedna kategorija, a to je YES kategorija. U YES probleme svrstavamo pogreške koje nastaju kao rezultat nerazumijevanja semantike određene naredbe i njezine jezične građe.

Konstruktivni problemi za razliku od nekonstruktivnih programerima početnicima stvaraju poteškoće pri učenju točne semantike jezičnih konstrukcija. Nazivi različitih programskih konstrukcija su najčešće u prirodnom jeziku (engleski) stoga početnici mogu krivo interpretirati instrukciju te na taj način pretpostaviti da računalo ima istu interpretaciju kao i oni. Početnici često pišu programe koji rade u osnovnim slučajevima, ali ne i općenito.

Istraživanja pokazuju kako problem programera početnika ne leži u strukturi programskog jezika nego da je pravi problem „spajanje dijelova“ [13]. Ti „dijelovi“ su zapravo komponente programa koje funkcioniraju same za sebe, ali isto tako spojene čine jednu cjelinu koju nazivamo programom. Učenje programiranja, u današnje vrijeme, najviše

uključuje učenje gotovih biblioteka stereotipnih rješenja. Često ćemo se i pronaći u situaciji gdje nam nije potrebno ni efikasno razvijati vlastite biblioteke nego koristit već postojeće, ali u kontekstu koji je nama zanimljiv. Stručnjaci su puno više upoznati sa semantikom i sintaksom programskih jezika pa su tako dobro upoznati ili su pak razvili vlastite velike biblioteke stereotipnih rješenja i strategije sastavljanja rješenja.

Učenike treba učiti da programiranju kao disciplini dizajna gdje je produkt našeg rada funkcionalni mehanizam koji obavlja vlastitu funkciju, a ne program sam za sebe. Programiranje je vezano uz mnoge discipline, a rješavanje problema se preslikava iz stvarnog svijeta.

Jedna od strategija koje bi mogle pomoći programerima početnicima je takozvana strategija „bistrenje korak po korak“ [13]. Ovu strategiju najlakše je opisati u tri koraka, a to su: (1) pretpostavka da posjedujemo „bačvu s rješenjima u limenkama“, (2) sagledavanje problema i traženje mogućih „limenki“ koje možemo iskoristiti za rješavanje istoga i (3) rastavljanje problema na podprobleme kako bi iskoristili (stereotipna) rješenja iz limenki. Što više „limenki“ posjedujemo te što više problema možemo riješiti postojećim „limenkama“ to je bolje. Ovo se izravno može preslikati na objektno-orijentirano programiranje i ponovno korištenje programskog koda jer je to dobra praksa za buduće projekte pojedinca.

Soloway i Ehrlich [14] znanje programiranja predstavljaju setom struktura blokovskog oblika koje nazivamo planovima. U vlastitom istraživanju iskazuju dokaz kako programeri eksperti imaju problema s razumijevanjem neplanski pisanih programa što ih vodi do zaključka kako se plansko prepoznavanje dogoditi u razumijevanju ukoliko se želi krenuti stopama eksperata.

Programeri eksperti su pokazali znatno manju snalažljivost u neplanski pisanim programima nego li u planski pisanim programima. Takav odskok, između planski i neplanski pisanih programa, nije se vidio kod programera početnika. Svaki dobar programer bi trebao imati sposobnost prepoznavanja veza među dijelovima programa tj. načina na koji isti komuniciraju. Interakcija takvih programa nije jednostavna zbog delokalizacije planova, program nije lokaliziran na jednome mjestu nego razbijen u manje dijelove.

U konačnici Soloway i Ehrlich [14] iznose zaključak kako je glavni nedostatak programera početnika posjedovanje manje programskog znanja i vještina za razvijanje pravilnih reprezentacija (ekspertovih).

3.2. Poteškoće u OOP

Opće poznata činjenica je kako je objektno-orijentirano programiranje kao i ostale vrste programiranja teško savladati, posebno kao početnik. Lucker tvrdi da učenje objektno-orijentacijskog predloška "zahtjeva ništa manje od potpunog pogleda na svijet",

Najosnovniji koncepti objektno-orijentiranog programiranja su objekt i klasa pa prema tome možemo reći kako je i najosnovniji problem miješanje tih dvaju pojmova što je Eckerdal utvrdio rekavši da studenti teško razdvajaju koncepte objekta i klase. Također, postoji i opasnost da početnici poistovjete objekt s varijablom. To se često može dogoditi zbog prethodnih iskustava s proceduralnim programiranjem (ukoliko se proceduralni pristup podučavao ranije).

Holland, Griffic i Woodman tvrde da se pogrešne koncepcije objektne orijentacije mogu teško ispraviti nego da se samo nesvjesno filtriraju i izobličuju.

Intervjuiranjem studenata o razumijevanju koncepata objekta i klase, koji su završili svoj prvi programski tečaj, dobivaju odgovore kako su koncepti problematični i preteški za učenje bez obzira na uloženi trud.

Među glavne poteškoće spadaju i razumijevanje konstruktora što su pokazali srednjoškolski učenici na svom prvom tečaju Jave (Reagonis i Ben-Ari).

Neki studenti čak smatraju kako je redukcija broja linija koda i klasa važnije od koncepta „učahurenja" podataka jer kako kažu nisu zadovoljni „skakanjem" iz klase u klasu kada čitaju programe (Fleury).

Krivi primjer također može stvoriti zablude. Npr. ako početniku dajemo primjere gdje se objekt ponaša kao baza podataka (skladište neaktivnih i nepromjenjivih podataka), stvaramo sliku kako je objekt neaktivan zanemarujući njegovu pravu svrhu. Dobar primjer aktivnog objekta može biti bilo kakvo živo biće jer može imati i promjenjive i nepromjenjive opise, a isto tako i sposobnosti koje ćemo opisati metodama. Ako isto

promotrimo iz perspektive računalne igre, veliki dio računalne igre je ništa drugo nego sustav aktivnih klasa koja imaju promjenjiva i nepromjenjiva stanja.

Prvi susreti početnika s metodama može ostaviti snažan pečat na njihovo viđenje istih. Poznavanje metoda kao funkcija koje isključivo obavljaju nekakve zadatke može biti jedan od primjera gdje programer početnik ne vidi drugu upotrebu metoda kao npr. prenošenje poruka.

Još neke od mogućih zabluda mogu biti:

- jedan objekt može biti referenciran samo preko jedne varijable umjesto više njih, a dvije varijable uvijek pokazuju na različite objekte i samo različite objekte
- kada instanciramo jedan objekt, varijabli koja sadrži njegovu referencu ne možemo pridijeliti (zamijeniti) referencu na neki drugi objekt iste klase
- dva objekta iste klase i istih atributa su isti objekt

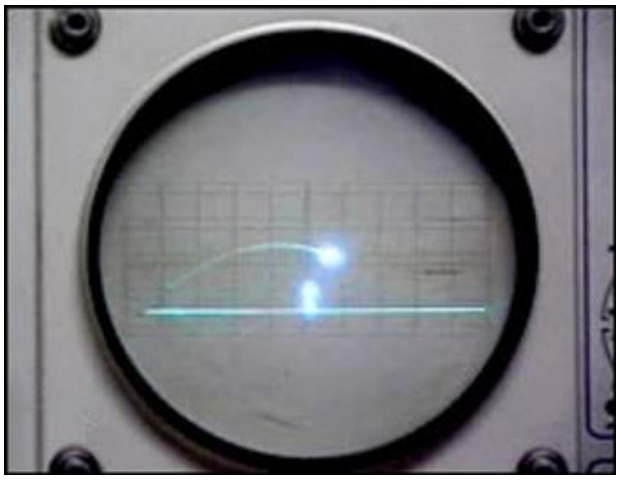
Pokazalo se kako studenti imaju tendenciju izrade programa s centraliziranom kontrolom i komunikacijom. Programi su tako osmišljeni da glavni objekt čini većinu odluka te se komunikacija između objekata vrše preko glavnoga objekta koji se ponaša kao centrala. Ovo je loša praksa možda zbog prijašnjih iskustava s malim programima na kojima su početnici učili svoje prve klase, strukture i sl. Ovako dizajnirani programi ne odišu duhom OOP-a te imaju jako malu mogućnost ponovne iskoristivosti jer su sve klase izrađene za točno specificiranu svrhu.

Uzmimo za primjer igre i zamislimo bilo kakvu klasu koja je mogla biti napisana za tu igru kao npr. klasa „vozilo“ ili klasa „čovjek“. Obadvije klase, unutar istoga okvira za izradu igara, mogu biti iskorištene za bilo koji projekt igre jer imaju univerzalnu funkcionalnost koja naravno nije striktno vezana za određeni program.

4. Računalne igre

Računalna igra je ništa drugo nego vrsta programa koji se najčešće koristi za zabavu onih koji ga koriste. Računalna igra, korisnika (čovjeka) stavlja u kontrolu što znači kako izvršavanje programa ovisi o željama korisnika. Esencijalni dio za interakciju čovjeka i računalne igre je korisničko sučelje. Korisničko sučelje (eng. *User interface*, skraćeno UI) omogućuje korisniku da, preko ulaznih jedinica računala „izrazi te svoje želje“ te kao povratnu informaciju dobije sliku i zvuk na izlaznim jedinicama računala (monitor, zvučnici).

Prva interaktivna elektronička igra datira još iz 1947. godine (Sl. 4.1), a napravio ju je Thomas T. Goldsmith Jr. Igra je simulator projektila inspirirana radarskim sučeljima iz drugoga svjetskoga rata. Igrač je upravljao snopom elektrona katodne cijevi (eng. Catode ray tube, CRT) tj. upravljao je točkom na ekranu koju je taj snop elektrona osvjetljavao.

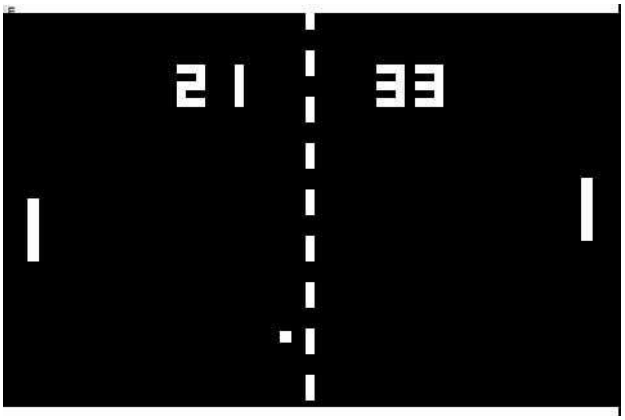


Sl. 4.1 CRT uređaj za zabavu

Alan Turing je također osoba koju možemo navesti u povijesti igara zbog svoga teoretskog računalnog programa za igranje šaha. On je 1947. godine napisao teoriju za program koji bih igrao šah što je prvi primjer inteligencije strojeva. Već 1950. godine Bryer Bettencourt gradi stroj za igranje igre „križić kružić“ (eng. Tic Tac Toe) prikazujući umjetnu inteligenciju stroja s mogućim odabirom težine igre tj. inteligencije stroja.

Kroz sljedećih 20-ak godina bilo je još nekolicina izrađenih igara, ali se od svih njih najviše ističe igra „Pong“ (Sl. 4.2). Igra je izdana 1972. godine i time postala prva uspješna

arkadna igra koja je popularizirala taj novi medij. Iako najpopularnija, ovo nije prva igra ovoga tipa. Igra se temeljila na jednostavnoj 2D grafici u kojoj se simulirao stolni tenis.



Sl. 4.2 Pong igra

Sedamdesete su zapravo bile godine začetka video igara i zlatno razdoblje arkadnih igara. Osamdesetih godina pojavljuju se prve izdavačke kuće za igre koje postoje ili su postojale preko 20 godina. Primjer jedne takve kompanije je „EA games“. Produkcijska i izdavačka kuća koja postoji već 33 godine te godišnji prihod broji u milijardama američkih dolara. Također, osamdesetih nastaju prvi žanrovi igara. Devedesetih godina dvadesetog stoljeća se također događa jedna od revolucija u računalnim video igrama, a to je prelazak iz rasterske 2D grafike u 3D grafiku. Prelaskom na 3D grafiku razvijaju se novi žanrovi od kojih su i danas, nakon 20-ak godina, jedni od najpopularnijih, a to su: FPS igre (eng. *frist person shooter*), strategijske igre (eng. *real-time strategy*) i mrežne igre poznate pod nazivom MMO (eng. *massive multiplayer online games*). Devedesetih, točnije 1994 svijetu je predstavljena prva PlayStation konzola koja je i u današnje vrijeme jedna od najpopularnih konzola na svijetu. Također, ovo razdoblje je zabilježeno kao i začetak PC igara (eng. *personal computer games*).

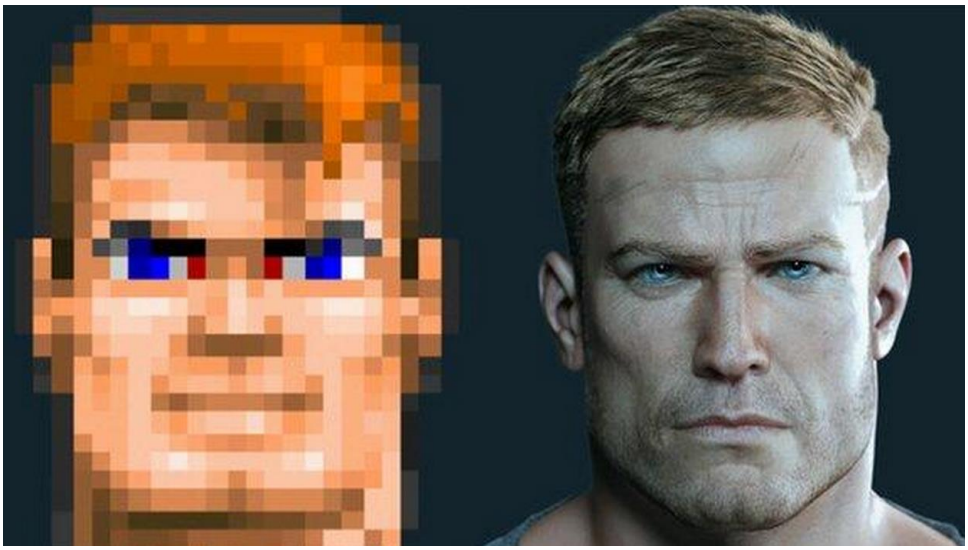
21. stoljeće također za sobom donosi nove tehnologije u području video igara. Tehnologija pa tako i same igre se strahovito brzo razvijaju. U prethodnih petnaest godina, osim razvoja već postojećih platformi za igre dolaze na tržište i nove koje imaju alternativan pristup igrama. Jedan takav primjer je proizvod zvan „Kinect“. Kinect omogućuje igranje igara bez kontrolora uz pomoć kamera koje prate čovjekove pokrete te tako pokreću elemente igre. Pojavom „pametnih mobitela“ (eng. *smartphone*) populariziraju se takozvane „*Casual*“ igre. Iako su mobilne igre postojale još 90-ih godina prošlog stoljeća, tek se porast tog dijela industrije dogodio razvojem mobitela nove generacije i tablet računala.

Casual igre su jednostavne, često 2D, igre gdje je za kontrolu potreban svega jedno dugme ili danas jedan pokret prstom preko ekrana na dodir (eng. *touch screen*). U tu kategoriju često svrstavamo logičke igre, puzzle i sl. Zadnjih nekoliko godina pojavljuju se tehnologije bliske budućnosti, a to su „Igre u oblaku“ (eng. *Cloud based games*) i virtualna stvarnost (eng. *virtual reality*, skraćeno VR).

„Igre u oblaku“ su najobičnije igre koje i danas postoje, ali je razlika u mjestu gdje je igra pokretana i mjesta s kojeg se upravlja igrom. Danas igrač igru pokreće na vlastitom računalu/konzoli te na istome i upravlja igrom dok nam „Oblak“ tehnologija omogućuje da igra bude pokretana na poslužitelju (eng. *server*), a kontrola igre bih se vršila preko mreže. Također, zadaća poslužitelja je da isporuči obrađenu sliku kako bi igrač mogao vidjeti sadržaj igre.

Virtualna stvarnost je pojam koji se do sada koristio za virtualne svjetove unutar računala dok danas popularizira još jedno značenje. Razvijene su tehnologije koje omogućuju korisniku da bude dio igre. To je omogućeno uz pomoć posebnog zaslona koji se postavlja na glavu poput naočala. Taj uređaj pred očima stvara takvu sliku da igrač ima osjećaj kako je dio virtualnog svijeta. Obadvije tehnologije predviđaju rast popularnosti kroz 2016. i 2017. godinu.

Primjer napretka računalne grafike najbolje se može vidjeti iz slike (Sl. 4.3). Koliko je god računalna grafika uznapredovala toliko su uznapredovala i druga područja računalstva bez kojih igre ne bih postojale.



Sl. 4.3 Usporedba računalne grafike iz igre Wolfstein 1992 i 2014

Uz svu navedenu povijest i tehnologiju koja okružuje industriju igara možemo igre promatrati iz dvije perspektive. Iz perspektive njene funkcije i stvaralačke perspektive. Funkcija igre je zabava čovjeka koji ju igra. Pod stvaralaštvo svrstavamo sva znanja potrebna za izradu jedne igre.

Za izradu jedne igre potrebna su mnoga znanja iz različitih područja znanosti i umjetnosti. Prije nego li krenemo na programiranje igre potrebno je izraditi svaki dio igre koji uključuje 2D slike, 3D modele, video i audio datoteke. Kombinacijom tih elemenata stvaramo objekte iz vlastite mašte ili replike iz stvarnog svijeta. Te objekte oživljavamo različitim tehnologijama animacije, a u konačnici to sve spajamo u jednu veliku cjelinu programiranjem. Programiranje nam omogućuje da sve te objekte u igri „oživimo“ i podarimo im vlastiti identitet. Sve današnje igre izrađene su u nekom obliku objektno-orijentiranoga jezika pa stoga možemo reći kako su one odličan primjer istoga. Razlog više zašto su u ovom radu pričamo o igrama je to što su one jako dobra vizualna reprezentacija OOP-a. Igra je puno više od samog OOP-a, a to možemo reći jer današnje igre u velikoj mjeri posjeduju jako naprednu umjetnu inteligenciju (eng. *artificial intelligence,* skraćeno AI). Nadalje, interakcija čovjeka i računala (eng. *human computer interaction,* skraćeno HCI) i mreže su također dva područja koja su uključena u samu izradu igara, a dio su kurikuluma mnogih sveučilišnih studija informatike.

4.1. Statistika o industriji igara

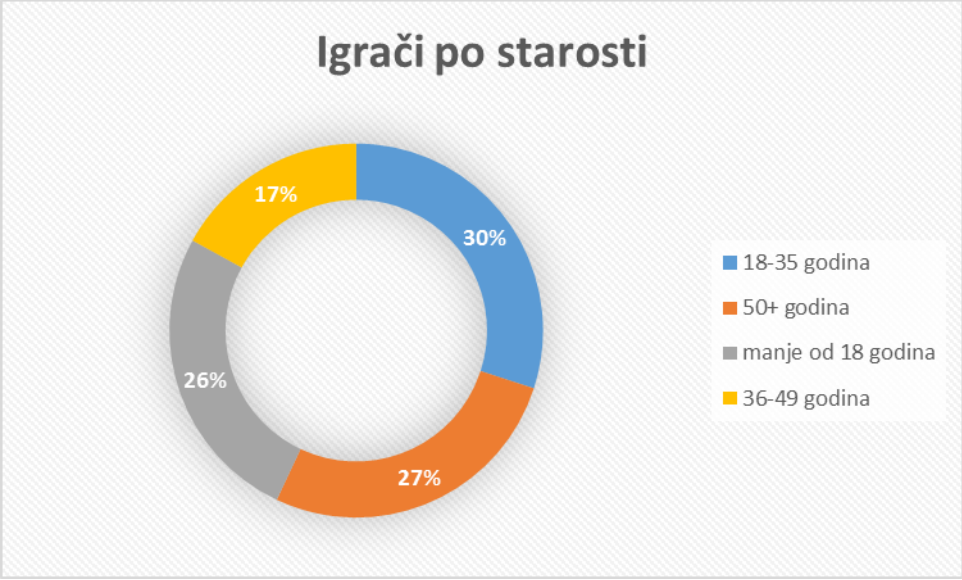
Današnja zajednica igrača (eng. *gaming community*) je raznolika, ali muška populacija i dalje dominira u ovom području. Ovime se misli na takozvane „*hardcore*“ igrače. Takvi igrači često prate veliki broj jako popularnih naslova koje definiramo kao industrijski „velike“ ili profesionalne igre. To s većinom 3D računane i igre na konzolama. U zadnje vrijeme do isticaja dolaze takozvane „*casual*“ igre poput pasijansa (eng. *solitarie*) i razno raznih puzli. To su često jednostavne 2D igre za koje nije potrebno ulagati puno vremena, ali imaju istu svrhu kao i druge igre, a to je zabava. Zbog tih „*casual*“ igara [15] postotak igrača različitoga spola je skoro pa ujednačen gdje npr. Amerika broji 44% žena koje igraju igre. Svakoj izjavi da će samo „*hardcore*“ igrači ili muškarci biti zainteresirani za izradu (programiranje) igara možemo proturječiti time da su i žene podjednako aktivne kao igrači.

Pojavom „pametnih“ mobilnih uređaja, tablet računala raste broj „*casual*“ igara pa tako i igrača istoga tipa igara. Također, iste igre su jako pristupačne, a većina istih je i besplatna te uz određene alate ih svatko može izraditi i publicirati. Iako su „*casual*“ igre puno jednostavnije to ne umanjuje činjenicu da su to igre te da za njih nisu potrebe određene vještine za izradu kao kod industrijski velikih igara.. Također, svaka igra bez obzira na tip sadrži sve osnovne koncepte programiranja osobito OOP-a. Naprotiv, „*casual*“ igre su od svih igara najbolji izbor za igre u nastavi jer ih ta osobina jednostavnosti čini jako dobrim kandidatom za podučavanje programera početnika.

Možemo zaključiti kako izrada igara ne mora značiti izradu zahtjevnih 3D igara, a to potvrđuje i istraživanje [15] gdje je utvrđeno kako su najčešći tipovi igara koje igrači igraju: 31% društvene igre, 30% akcijske igre i 30% puzzle/kartaške igre/kvizovi i sl.

Također, napretkom tehnologije imamo široku lepezu uređaja za koje igre mogu biti izrađene, a statistika pokazuje kako 62% igrača posjeduje stolna računala, 56% iste populacije posjeduje igraće konzole, 35% „pametne“ mobitele, 35% bežične uređaje na kojima također igraju igre te 21% igra igre i na posebnim ručnim igraćim konzolama.

43% vlasnika tablet računala provodi više vremena na istima nego li na stolnim računalima ili televizorima, a 84% spomenute populacije koristi tablet računala za igranje igara.



Sl. 4.4 Podjela igrača prema starosti istih

Na slici (Sl. 4.4) vidimo kako je najveći postotak igrača pripada mladima tj. osobama mlađima od 18 godina i osobama između 18 i 35 godina. U taj isti raspon godina spadaju

polaznici osnovnih i srednjih škola, a također i fakulteta što može biti zanimljiv podatak kada želimo uključiti igre u nastavu.

Uz današnje igre često vežemo i pojam socijalizacije gdje se navodi kako 56% igrača igra igre s drugima od kojih je: 42% prijatelji, 21% obitelj, 16% roditelji, 15% partner. Čak 54% igrača navodi kako im igre pomažu u povezivanju s prijateljima, a isto tako 45% smatra kako im igre pomažu da više vremena provode s vlastitom obitelji. Sve više roditelja počinje igrati igre kako bi se mogli zabaviti s vlastitom djecom te čak 59% roditelja igra igre sa svojom djecom barem jednom tjedno te 63% smatra igre kao pozitivan dio života vlastite djece.

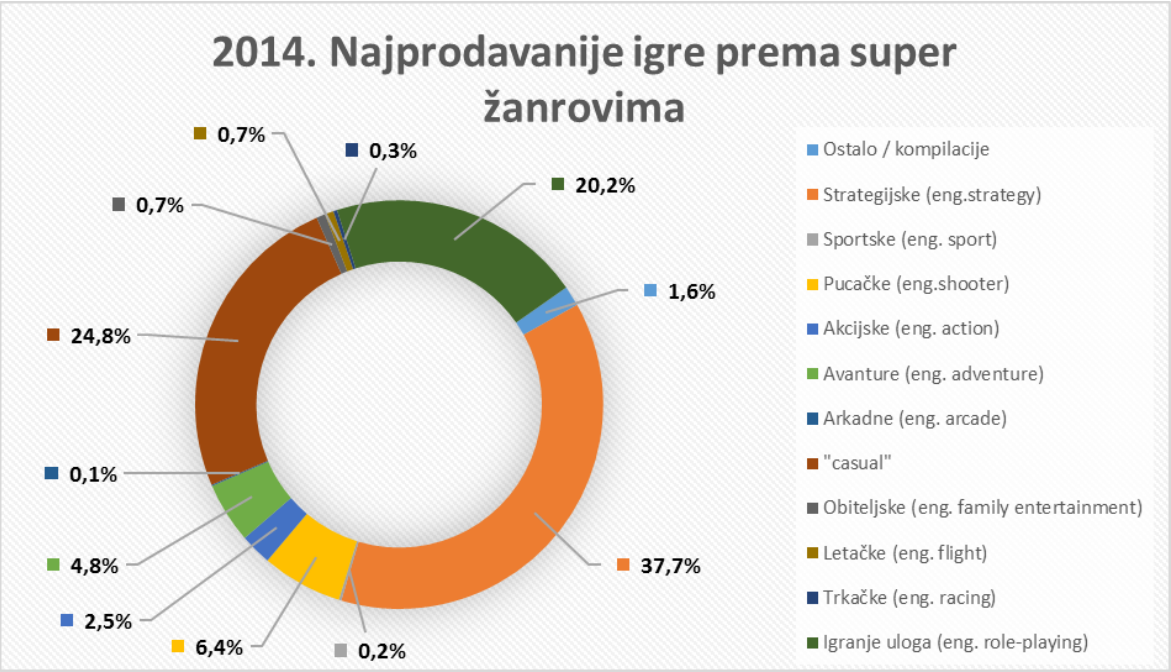
Neki od roditelja navode sljedeće razloge igranja igara s vlastitom djecom:

- To je zabava za cijelu obitelj
- Zato što je to bila želja djece
- To je dobar način za socijalizaciju s vlastitom djecom
- To je dobar način za praćenje sadržaja s kojima se djeca susreću
- Zato što uživaju u igrama koliko i vlastita djeca

„Igre su jako kompleksni sistemi sastavljeni od pravila koja su u međusobnoj interakciji. Igrač mora razmišljati poput dizajnera igre i stvarati hipoteze o interakciji tih pravila kako bi mogli postići željeni ili predviđeni cilj te u konačnici dobiti nekakve rezultate. Razmišljati kao dizajner kako bi razumio nekakav sustav je ključna sposobnost 21. stoljeća“. – Dr. James Paul Gee.

Na slici (Sl. 4.5) možemo vidjeti podjelu igara prema nekakvim super žanrovima te koliko je igara prodano za pojedini žanr od ukopnog broja prodanih igara za 2014. godinu. Ovi podatci ne sadržavaju podatke o piratskim i besplatnim igrama.

Najprodavanija igra 2014. godine je „The Sims 4“ dok ju slijedi vlastiti prethodnik „The Sims 3: Starter Pack“. U top 20 najčešće se nalaze takozvane „MMORPG“ igre poput World of Warcraft. Igra World of Warcraft je fenomen sama za sebe jer je jedna od najuspješnijih igara već 10 godina za redom gdje broji milione igrača koji su mrežno povezani.



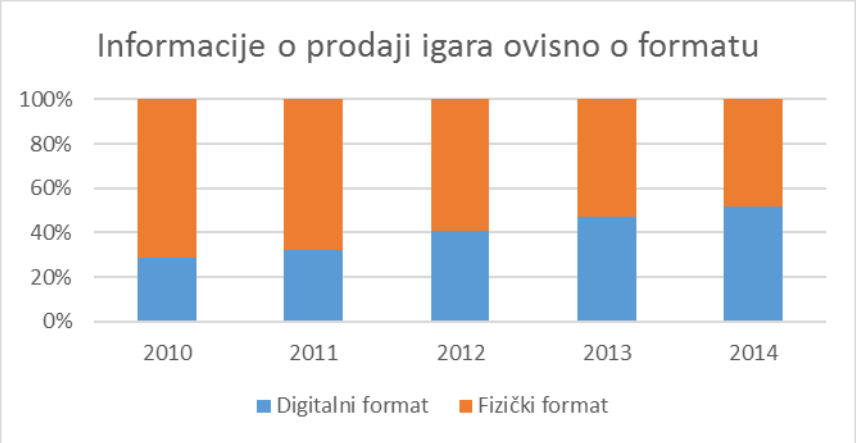
Sl. 4.5 Najprodavanije igre 2014. godine

U nastavku navedeni su razlozi odabira igara (Sl. 4.6). Neki od interesantnijih razloga su obrazovna vrijednost, zanimljiva priča i mogućnost igranja s drugim igračima. Iz navedenoga se može vidjeti kako igrači vide određenu vrijednost u igrama.

Sl. 4.6 Faktori koji utječu na kupovinu igre

Svjetska industrija video igara narasla je za 9% 2013. godine s vrijednošću od 75 milijardi američkih dolara te se smatra kako će do 2016 godine prijeći 86 milijardi američkih dolara.

2014 godine je potrošeno čak tri puta više novca na sam sadržaj (igre) nego li na hardver koji ih podržava s ukupnom potrošnjom od 22.41 milijardu američkih dolara. Postoji i trend (Sl. 4.7) sve veće kupovine igara u digitalnom formatu naspram fizičkog (CD, DVD).



Sl. 4.7 Prodaja igara ovisno o formatu u kojem dolaze

Iz nekolicine prethodno spomenutih podataka vidimo kako je industrija igara bez sumnje jedna od jakih te kako jako dobro opstaje u današnjem svijetu. Također ako malo više istražimo naslove popularnih igara, većinom ćemo nalaziti iste naslove s različitim ekstenzijama. Primjer jedne takve je fenomen zvan „World of Warcraft“ koji trenutno broji 6 nastavaka kroz 10 godina. Proizvođači iste igre, kako kažu, imaju u planu barem još 10 godina. To je tip igre MMORPG gdje svaki igrač preuzima ulogu jednoga ili više heroja koji je smješten u virtualni umreženi svijet koji trenutno broji oko 7 miliona pretplatnika koji su svakodnevno u interakciji. Igra je nekoć brojila i 11 miliona pretplatnika. Tijekom 10 godina broji čak preko 100 miliona igrača preko 244 države svijeta. Zbog svoje velike baze pretplatnika igra je svih 10 godina na vrhu ljestvice masivnih multi-igračkih igara (eng. *massive multiplayer online game*, skraćeno MMO) gdje broji godišnje prihode od 1,041 milijardu američkih dolara te zauzima 36% tržišta ovog tipa igara.

4.2. Okviri za izradu igara

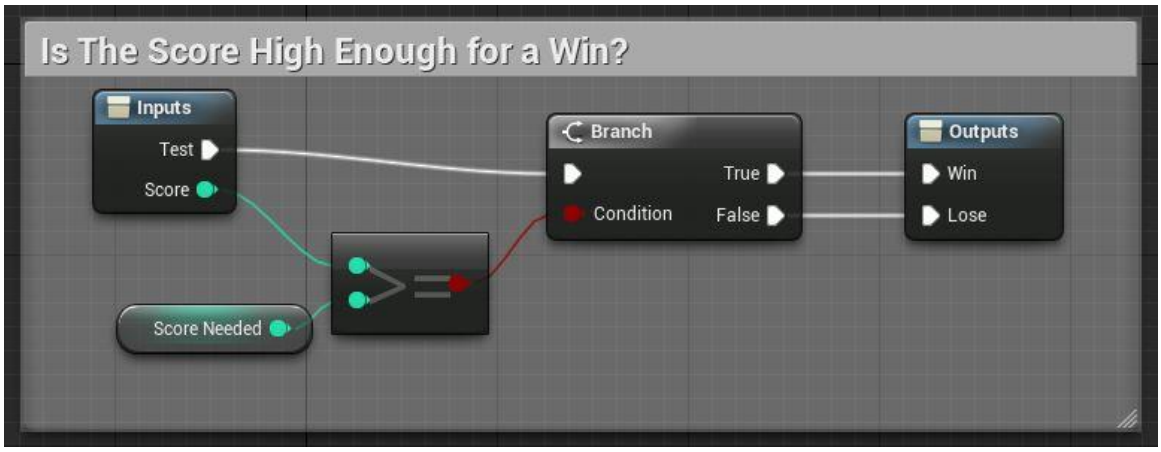
Okviri za izradu igara (eng. *Game engine*) su programski okviri koji, kako samo ime govori, služe za izradu video igara za raznolike platforme. Takav programski okvir sastoji se od nekolicine modula koji su zaslužni za pojedinu funkcionalnost igre, a to su:

- *Renderer* za 2D i 3D grafike (eng. *rendering engine*),
- Fizika (eng. *physics engine*),
- Detektor sudara (eng. *collision detection*),
- Zvuk,
- Skriptiranje,
- Animacija,
- Umjetna inteligencija,
- Upravitelj memorijom (eng. *memory management*),
- Mogućnost spajanja na mrežu i podržavanje poslužitelja,
- Ostali moduli…

Danas postoji mnogo okvira za izradu igara koji mogu služiti za različitu svrhu kao što je i obrazovanje. Ovi alati nas ne ograničavaju samo na izradu igara, mnogi se koriste za razne vizualizacije npr. interijera i eksterijera ili čak izradu raznoraznih simulacija.

Većina proizvođača igara posjeduje vlastite tehnologije koje nisu dostupne javnosti, ali isto tako postoji nekolicina profesionalnih okvira za izradu igara koji su dostupni javnosti bez ikakvih troškova korištenja. Trenutno, kao najpopularnije možemo navesti „Unreal Engine“ i „CryEngine“ koji oduzimaju dah mogućnostima koje pružaju. Ti *engine-i* kvalificiraju se kao vizualno najbolji te su namijenjeni za proizvođače s velikim timovima. Njihovo korištenje može biti jako kompleksno jer su potrebna široka znanja programiranja, a svo skriptiranje se izvodi u programskom jeziku C++.

Iako kompleksni, Unreal Engine s četvrtom inačicom okvira izdaje zanimljiv alat koji omogućuje, za neke, jednostavniji pristup izradi skripti koje pokreću objekte u igri. Proizvođači tog okvira taj alat nazivaju grafovima (eng. *Graphs*) ili nacrtima (eng. *Blueprints*). Taj „*Blueprint*“ sistem je jedan oblik vizualnog programiranja gdje umjesto pisanja programskog koda programer odabire okvire koje predstavljaju varijable, objekte, funkcije i sl. u programu te stvara žičane veze između istih kako bih ostvario interakciju/komunikaciju između istih. Rezultat povezivanja tih okvira je graf koji je zapravo vizualna reprezentacija programskog koda kojeg bi programer napisao kako bi ostvario istu funkcionalnost unutar igre.



Sl. 4.8 Unreal Engine 4, Blueprint (graph)

Na slici (Sl. 4.8) prikazan je jedan primjer tog „blueprinta“ s grananjem. Iz primjera možemo rekonstruirati programski kod gdje bi okvir grananja (eng. *Branch*) realizirali IF-ELSE operatorima. Ovaj sistem, iako nije vidljivo na slici, omogućava pregled izvođenja skripte tako da pokazuje protok podataka linijama koje vežu pojedine okvire te tako olakšava pronalazak potencijalnih grešaka u programu. Naravno ovaj okvir za izradu igara pruža mogućnost izravnog pisanja skripti unutar razvojnog okruženja Visual Studio u programskom jeziku C++, ali se za početnike i „ne-programere“ preporuča korištenje „*blueprint*“ sistema.

Manje zahtjevniji alati za izradu igara su „Construct 2“ i „Unity game engine“. Construct 2 je 2D okvir za izradu igara te služi za izradu jednostavnih mobilnih igara uz mogućnost izrade i PC i HTML5 igara. Podržavanje HTML5 tehnologije u zadnje vrijeme postaje sve popularnije pa čak i „jaki“ okviri poput Unreal engine-a 4 imaju podršku za istu tehnologiju.

A razliku od Construct-a 2, Unity game engine podržava 2D i 3D igre te ima široku lepezu platformi na koje se igre mogu isporučiti kao što su: HTML5, različiti mobilni operacijski sustavi, konzole poput XBOX-a i PlayStation-a, Windows, Linux itd. Unity postaje jako popularan te se može reći kako postavlja standarde za izradu modrenih video igara. Osim svoje prilagodljivosti različitim platformama, omogućuje izradu video igara s različitih operacijskih sustava u jezicima C#, JavaScript i Boo. Najnovija, peta, inačica Unity alata svojom kvalitetom postaje ravnopravna konkurencija alatima poput prethodno spomenutog Unreal Enginea. Alat se preporuča početnicima programerima koji žele izrađivati vlastite igre jer je jako jednostavan, intuitivan te je podržan jako dobrom dokumentacijom, a svojom popularnošću i raznovrsnim video tečajevima.

Navedeni okviri za izradu igara korišteni su na nekolicini svjetskih sveučilišta pa čak i na nekim splitskim sveučilištima. Naravno, da bi se određeni okvir za izradu igara koristio u nastavi on treba imati svoju obrazovnu vrijednost.

4.3. Igre u nastavi

Igre u nastavi su pojam koji može biti dvosmislen jer igre u nastavi ne znače isključivo izradu igara. Potrebno je razlikovati obrazovne igre od izrade igara u svrhu obrazovanja.

U prvom slučaju govorimo o igrama koje su napravljene isključivo za podučavanje određene teme ili razvijanje određenih sposobnosti. Ovakve igre najčešće izrađuju odgojitelja, učitelja dok su učenici ti koji uče kroz igru. Ta igra ne mora biti nužno računalna igra što i pokazuju povijesni izvori koji govore o šahu kao igri koja je služila za učenje strategija ratovanja u srednjem vijeku. Isto tako korištenje igara u obrazovanju, bilo računalnih ili ne, nije ograničeno samo na informatičke predmete nego njihova uporaba ovisi o samoj kreativnosti učitelja i interesu učenika.

Igre su često vezane za nekakav oblik fantazije gdje se učenik uključuje u aktivnosti učenja kroz naraciju i priče. Igre potiču aktivno učenje, interakciju između više osoba i timski rad. Učenje bazirano na igrama omogućuje prilagodljivost za više stilova učenja, a mogu otjecati na kognitivne i psihomotorne sposobnosti osobe. Dok učenje kroz igre može biti vrlo učinkovito, ono može postati i smetnja tako da učenik više pažnje posveti samoj igrici naspram učenja.

Drugi način korištenja igara u obrazovanju je izrada igara. Suprotno prethodnome pristupu, učenik je taj koji izrađuje igru te na taj način uči nove koncepte i stvara vlastite planove kako riješiti određene probleme. Uloga učitelja je da kontrolirano vodi učenika kroz taj proces učenja te da, ukoliko učenik nije sposoban za to, sam zadaje takve zadatke koji će imati obrazovnu vrijednost jer igra nije tu sama zbog sebe nego zbog učenja. Također, način na koji će igra biti izrađena je bitan, a isto tako i tip igre. Igra može biti izrađena uz pomoć gotovih okvira za izradu igara bilo industrijski profesionalnih ili edukacijski namijenjenih.

Neke igre se čak mogu izraditi i bez upotrebe okvira za izradu igara nego nekakvim načinom improvizacije gdje postojeće biblioteke možemo izmijeniti i prilagoditi tako da programi budu interaktivni poput igre. Primjer jedne takve biblioteke je „Windows Forms

Application“ gdje s nekolicinu izmjena formi aplikacije možemo ostvariti kretnje objekata u igri. Takav program je također korišten u svrhu ovog istraživanjem pod imenom „Game SDK“.

Naravno postoje posebni edukacijski okviri za izradu igara za različite uzraste. Takvi alati izrađivani su od strane stručnjaka koji su se posvetili igrama u nastavi, ali o tome više u nastavku.

Neka istraživanja čak pokazuju kako je bitan i izbor grafike igre kao jedan od faktora kompleksnosti gdje 2D igre imaju prednost nad kompliciranijim 3D igrama.

U meta-analizi istraživanja algoritamske vizualizacije, Hundhausen [2002] dolazi do zaključaka kako su aktivnosti i uključenost studenata u te aktivnosti puno važnije o samog sadržaja i grafičkih elemenata vizualizacije. Njegova otkrića vode istraživanju koje veže razinu uključenosti studenata i vizualizacijskog alata te stvara takozvana taksonomija uključenosti (eng. *Engagement taxonomy*) [21] koju opisuje Naps [2002].

Definirano je šest različitih forma uključenosti u obrazovne situacije koje uključuju vizualizaciju, a iste forme mogu poslužiti za postavljanje hipoteza za buduća istraživanja:

1. Bez gledanja (nije korištena vizualizacijska tehnologija)
2. Gledanje (pasivna forma, dio je svih sljedećih formi)
3. Odgovaranje (učenik koristi vizualizaciju kao izvor za odgovaranje na pitanja)
4. Mijenjanje (forma se odnosi na modificiranje vizualizacije, mijenjanje ulaznih podataka kako bi se istražilo ponašanje algoritma za različite slučajeve)
5. Konstruiranje (učenici konstruiraju vlastite vizualizacije algoritama koje ne mora uključivati i kodiranje algoritma)
6. Predstavljanje (učenik prezentira vizualizaciju publici radi povratnih informacija i rasprave)

Iako Naps govori i vizualizaciji algoritama, ove razine uključenosti mogu se prilagoditi i na ovo ili slično istraživanje jer je i igra vizualna reprezentacija programa i algoritama koji ju pokreću. Naps navodi kako zadnje tri forme pokazuju najveću učinkovitost pri učenju, a igre se vrlo dobro mogu uklopiti u njegovu taksonomiju. Cilj nam i jest poduprijeti učenike da barem mijenjaju gotove igre ili okvire kako bi ih prilagodili vlastitim potrebama, ali isto tako da pokušaju konstruirati vlastitu igru što zahtjeva veliku uključenost u razne aktivnosti koje rezultiraju samim produktom tj. igrom.

Ako se ponovno vratimo na OOP i njegovu definiciju te igre sagledamo iz OO perspektive možemo zaključiti sljedeće:

- Igra je također preslika stvarnosti koju svaka osoba može prepoznati
- Ta preslikana stvarnost postoji zahvaljujući programiranju i konceptima OOP-a

Ako je igra vizualna reprezentacija OOP koncepata koji oponašaju fenomene iz stvarnoga svijeta to bi trebalo, na neki način, značiti kako će studenti bolje shvatiti OOP koncepte i njihovu ulogu ako vide kako se njihove klase i objekti ponašaju dok oni njima upravljaju. Naravno navedeno nam ništa ne znači ako student nije aktivan u zadanim aktivnostima, tj. ako ne spada barem u razinu uključenosti modificiranja.

4.3.1. Istraživanja koja uključuju okvire za izradu igara

Ovo područje je jako novo i neistraženo, ali isto tako budi znatiželju nekolicine istraživača pa su u nastavku predstavljena neka od istraživanja koja se bave sličnom tematikom kao i ovaj rad.

Za jedno od istraživanja s Korejskog sveučilišta [16] izrađen je poseban edukacijski okvir za izradu igara koji je bio prilagođen stupnju znanja učenika. Učenici su izrađivali igre koje nikako nije bilo moguće izraditi prema kurikulumu fokusiranom na stjecanje činjeničnog znanja. Istraživanje je pokazalo da su studenti sposobni za izradu igre u okviru za izradu igara koji nije napredan iako sadrži sve bitne elemente kao i napredni okviri. Istraživanje naglasak stavlja na programerske sposobnosti prije nego li znanje o programskom jeziku.

Istraživači s brazilskog sveučilišta u Sao Paulu [17] za istraživanje koriste GameMaker okvir za izradu igre jer je jednostavan te se studenti brzo mogu prilagoditi istome. GameMaker je alat za vizualno programiranje, a korištenjem istoga za obrazovanje programera početnika na prvo mjesto postavljaju se principi programerske logike naspram programske sintakse iako pruža mogućnost pregledavanja i pisanja programskog koda. U ovom istraživanju 2D igre se navode kao bolji izbor u odnosu na 3D igre zbog jednostavnije interakcije učenika i samog alata. Istraživanje je pokazalo kako profesori mogu puno lakše studente uvesti u za njih novi svijet objašnjavajući najbitnije koncepte bez detaljnog ulaska u sintaksu programskih jezika ili posebnosti određenih paradigmi.

Sa sveučilišta u Ljubljani [18] također dolazi jedno istraživanje na sličnu temu. Korišten je XNI okvir za izradu igara za Apple uređaje. Na kraju semestra izrađena je nekolicina igara te su sve postavljene na online Apple trgovinu kao besplatne za korištenje. Cilj je bio da će objavljivanje igara dodatno motivirati studente da više slobodnog vremena potroše na dorađivanje igre prije objavljivanja. Objavljene igre su sveukupno imale oko 9 000 preuzimanja u prva dva mjeseca. To je bila jako dobra povratna informacija studentima o njihovim sposobnostima. Studenti ocjenjuju ovako osmišljen kolegij kao najbolji od svih koje su do sada pohađali.

Colin A Depradine [19] također u nastavu uključuje okvir za izradu igara u svrhu povećanja interesa studenata za programiranjem i smanjivanjem loših ocjena na kolegijima poput OOP. Istraživanje je pokazalo kako studenti ovakvim pristupom bolje savladavaju OOP kolegij. Zanimljiva je informacija ta da je dvoje studenata, koji su ponavljali kolegij barem četiri puta, uspjelo položiti kolegij s jako dobrim uspjehom.

Selvarajah Mohanarajah i Shan Suthaharan [20] na američkim sveučilištima također vrše istraživanje na temu programiranja igara. U svrhu istraživanja napravljen je prototip igre u Javi te je dan studentima da izrade algoritme koji pokreću neki dio igre. Studenti su zatim testirali vlastiti algoritam igrajući igru i popravljali ju kada primijete neželjeno ponašanje algoritma u igri. Ovime se omogućuje zabavan način testiranja ishoda vlastitih algoritama. Studenti su pokazali pozitivne reakcije jer im je, kako kažu, draže igrati igru kako bi pronašli grešku u programskom kodu nego li koristiti bilo kakav tekstualni materijal ili način pronalaska greške. Neki su čak bili ponosni što im je pružena prilika popravljati greške u nekoj igri iako to nije industrijski standardna igra.

Iz navedenoga možemo zaključiti da ako želimo krenuti smjerom igara onda bi bilo dobro uvesti studente u svijet jednostavnih 2D igara koje mogu također plasirati na tržište, a tek nakon toga, na nekim naprednijim (opcionalnim) kolegijima ponuditi opciju izrade 3D igara.

4.3.2. Programiranje kao igra

Da učenje može biti igra je već stara ideja. Također, programiranje može biti zabavnije nego li se to čini na prvi pogled stoga se izrađuju okruženja posebno namijenjena za podučavanje programiranja izradom video igara ili kroz igru poput robota. U nastavku predstavljena su neka od popularnih okruženja takvoga tipa.

Scratch

Scratch je alat za izradu 2D igara kojem je primarna zadaća podučavanje programera početnika, u najčešćem slučaju djece. U ovom alatu mogu se izrađivati interaktivne priče, igre i razno razne animacije koje naravno možemo preko interneta podijeliti s drugima. Na vlastitoj web stranici postoji veliki izbor gotovih projekata koje je moguće pregledavati u web pretraživaču.

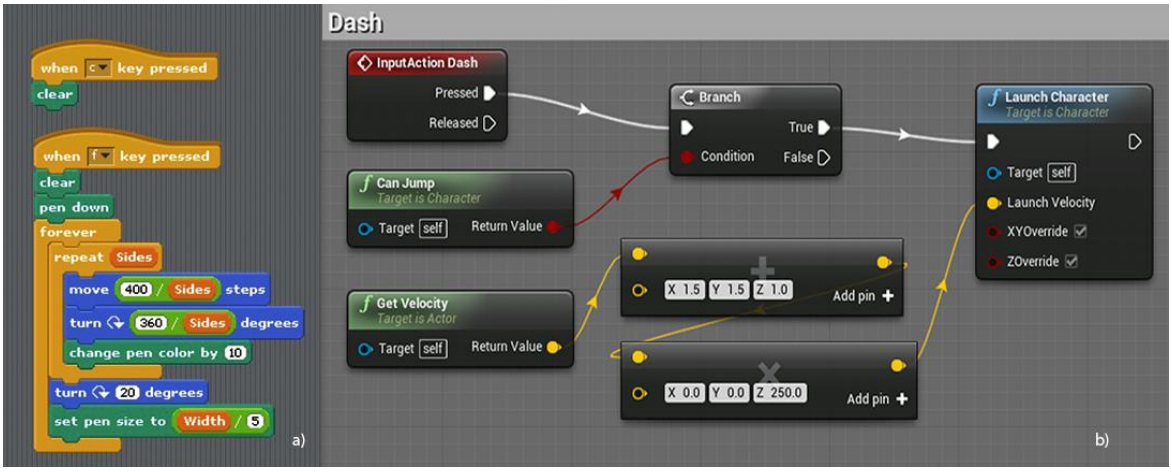
Način programiranja u scratch-u je vizualan što uvelike olakšava programiranje.

Alat se sastoji od tri osnovna dijela, a to su scena, operacija i skripti. Scena je prostor u koji programer dodaje 2D sličice objekata koji će biti sadržani u igri ili animaciji. Operacije su vizualizirane kao blokovi (Sl. 4.9) koje slažemo poput puzzli te tako opisujemo kompleksnija ponašanja likova u igri. Skripta je skup tih blokova koji su vezani za jedan objekt u sceni.

Sl. 4.9 Sučelje okvira za izradu igara „Scratch“

Na lijevoj stranici slike (Sl. 4.9) možemo vidjeti kategorije prema kojima su raspodijeljene pojedine operacije koje vršimo nad objektima u igri. U sredini je prikaz skripte u kojoj je skup povezanih operacija koje daju jedinstvenu funkcionalnost objektu igre tj. mački sa slike.

Ovim načinom programiranja izbjegnute su sintaksne pogreške dok se logičke pogreške i dalje mogu javljati, a iste se uklanjaju boljim razumijevanja operacija i funkcioniranja programa kojeg slažemo.



Sl. 4.10 Vizualno programiranje; a) Scratch; b) Unreal Engine 4

Na slici (Sl. 4.10) imamo usporedbu vizualnog načina programiranja u alatu za podučavanje djece programera početnika (a) i profesionalnog okruženja za izradu najkvalitetnijih igara današnjice (b). Možemo vidjeti kako oba slučaja počivaju na istome konceptu, ali su realizirani na različite načine. Čak i profesionalni alati posjeduju mogućnost vizualnog programiranja s već gotovim blokovima koji u različitim kombinacijama stvaraju jedinstvena stanja u programu.

Iz priloženoga možemo primijetiti kako oba slučaja posjeduju blokove koji opisuju osnovne dijelove programa poput petlji, grananja, varijabli i sl. Uz to u oba slučaja postoje i posebni blokovi koji možemo predstaviti kao stereotipna rješenja manjih problema, ovaj dio priče se može izravno vezati na poglavlje (3.1) gdje su spomenuta stereotipna rješenja u limenkama. Iako možda ne znamo od čega su ti blokovi sastavljeni, ono što znamo je njihovo ponašanje i funkcija koju mogu obavljati u programu. Na nama je samo razumjeti kako se blokovi ponašaju te vezanjem istih stvarati nove ideje i rješenja problema.

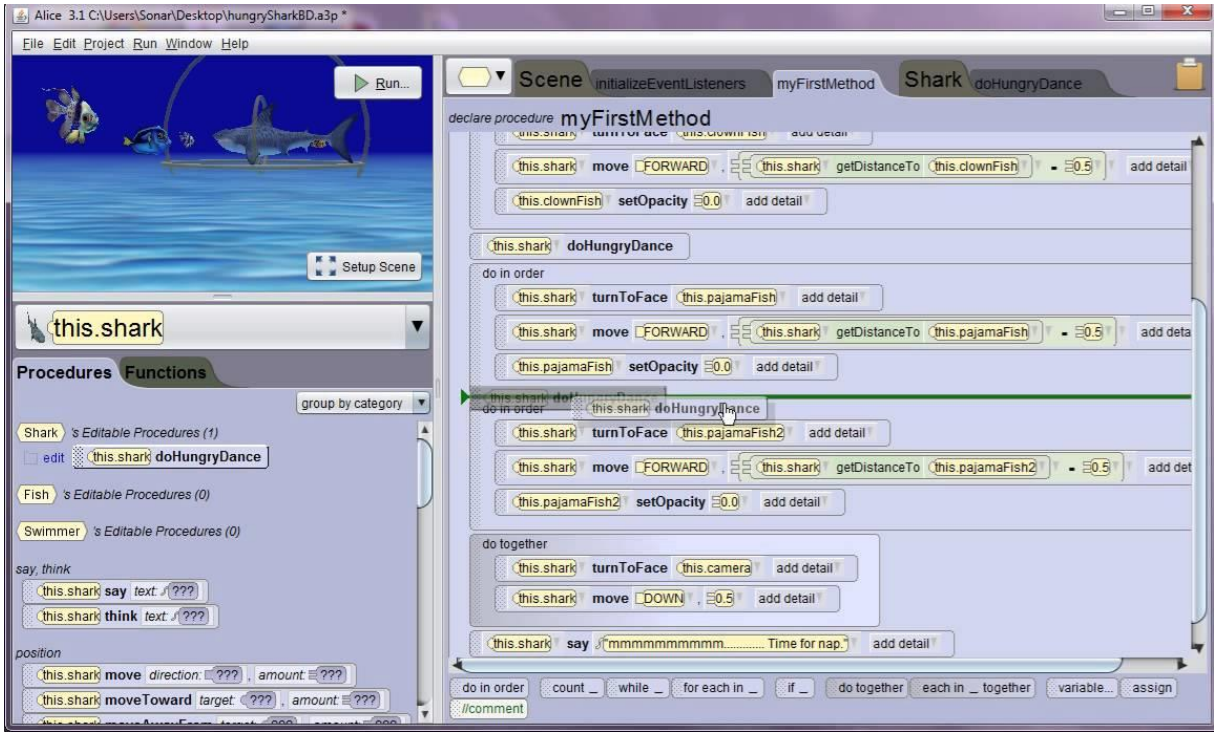
Iz ovoga možemo zaključiti kako programiranje u niti jednom slučaju ne treba shvatiti kao čisto pisanje „teksta koji nešto radi u programu“.

Alice

Alice je inovativno programsko okruženje za programiranje 3D igara, animacija ili interaktivnih priča poput scratch-a. Alat je namijenjen programerima početnicima koji se do sada nisu susretali s objektno-orijentiranim programiranjem. Omogućuje učenje

fundamentalnih koncepata programiranja kroz kontekst igara. Način programiranja je također vizualan kao i kod scratch okruženja što omogućuje povuci i ispusti (eng. *drag and drop*) sistem programiranja. Svaki blok odgovara standardnim instrukcijama koje možemo pronaći u bilo kojem objektno-orijentiranom programskom jeziku poput Java, C++ i C#. Svaki blok ili skup blokova moguće je odmah ispitati pokretanjem igre što omogućuje bolje razumijevanje veza između programskih naredbi i ponašanja objekata. Alice alat je besplatan te dolazi s velikom galerijom 3D modela opisanih Java klasama.

Na slici (Sl. 4.11) prikazano je alice 3 okruženje za koje već na prvi pogled možemo reći da je dosta kompliciranije nego li prethodno razmatrano 2D scratch okruženje. Ista veza između kompleksnosti 2D i 3D igara postoji i kod programiranja koristeći čisti programski kod jer je 3D virtualni svijet kompleksniji za opisati.



Sl. 4.11 Alice 3 programsko okruženje

Za razliku od scratch okruženja, alice u svojoj trećoj inačici (Alice 3) omogućuje izravan prijelaz na Javu što znači da sve što izradimo u obliku blokova možemo prevesti u programski kod. Uporama Alice alata ne zahtjeva poznavanje sintakse iako se ona može iz ovoga okruženja i naučiti.

Taj prelazak s vizualnog, blok, programiranja na programski kod nazivamo posredovanim transferom. Cilj ovoga je spojiti prednosti Alice intuitivnog razumijevanja osnovnih

programskih koncepata i izravnog prijelaza na Javu što znači da se učenje treba transferirati u drugi kontekst. Situacija iz prvog konteksta mora biti što sličnija situaciji drugog konteksta kako bi se dobio pozitivan efekt. To znači da bi program u Alice trebao biti što sličniji onome u Javi. Već je Alice 2 omogućio da blokovi što više oponašaju Java programski kod, ali je tek Alice 3 omogućila da se vidi Java kod na velikoj razini detalja. Prelaskom na Javu to i dalje ostaje isti program što se vidi kao olakšanje za učenike. Nakon prelaska na Javu svaki učenik bi trebao biti u mogućnosti vršiti promjene koda te u konačnici pisati vlastiti Java kod kako bi ostvario iste funkcionalnosti kao i s blokovima.

Tynker

Tynker možemo nazvati okvirom za izradu 2D igara, ali sama izrada igara je jedan manji dio cijelog sustava. Tynker je zapravo sustav koji omogućuje izradu 2D igara uz mogućnost kontrole projekata cijeloga razreda za nastavnika te također posjeduje jako detaljno razrađen tutorski sustav. Igre koje se izrađuju u ovom okruženju su jako vizualno atraktivne te su oko njih izrađeni cijeli tečajevi koji su već organizirani što znači da imaju već predefinirane zadatke i cilj koji je potrebno ostvariti. Tečajevi su organizirani prema uzrastu za koji su namijenjeni točnije razredu za koji se preporučaju.

Za razliku od prethodno navedenih alata, Tynker nije besplatan, ali je zato jako dostupan jer je dovoljno biti povezan na mrežu kako bih se koristio jer je cijeli sustav pokreće na web pretraživaču. Također, postoji i verzija tynker-a za iPad tablet računala.

Osim već gotovih 2D predložaka i interaktivnog sučelja za programiranje Tynker posjeduje i „umjetnički studio“ (eng. art studio) gdje djeca mogu crtati ili kombinacijom predložaka izrađivati vlastite objekte u igri.

Slično kao Alice 3, Tynker omogućuje transfer iz blokova u programski kod, a primjer toga vidljiv je na slici (Sl. 4.12).

Sl. 4.12 Prijelaz s vizualnog programiranja na programski kod

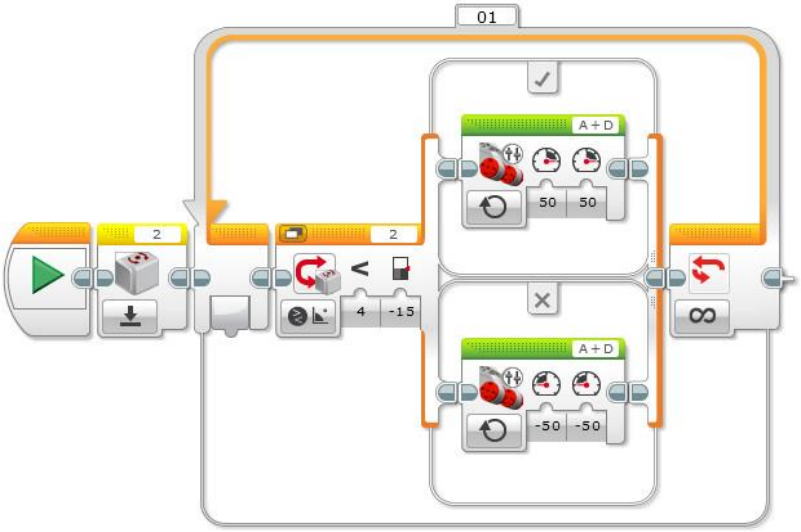
Iz slike (Sl. 4.12) također možemo primijetiti jako veliku sličnost sa scratch-om gdje su blokovi dizajnirani na jako sličan način. Transfer se odvija na programski jezik JavaScript. Za razliku od scratch-a, Tynker posjeduje puno više blok naredbi te posjeduje poseban dio namijenjen animacijama i fizici u igri.

Tynker ima ugrađen posebni dnevnik za pregled svih uspjeha studenata koji se prati preko raznih kvizova, puzzli i analize programskog koda.

Lego mindstorms

Iako se ne radi o izradi igara vrijedno je spomenuti i lego robote. Za razliku od korištenja okvira za izradu igara gdje upravljamo virtualnim objektima, lego mindstorms nam omogućuje da programiramo i upravljamo fizičkim objektima tj. robotima. U lego mindstorms uključeni su programibilni roboti koji se mogu sastavljati, a konačn izgled robota je ograničen samo maštom onoga tko ga sastavlja. Svaki robot posjeduje vlastite motore, senzore, spojnice i programibilno blok računalo.

Roboti dolaze uz vlastiti softver za programiranje istih. Koji je također vizualan tj. blokovski. Cijeli sustav baziran je na kontroli motora robota uz pomoć blokova koji predstavljaju same motore, senzore, a isto tako i osnovne programske koncepte poput varijabli, petlji i sl.. Na slici (Sl. 4.13) prikazan je jedan kratki program koji upravlja robotom.



Sl. 4.13 Lego minstorms ev3 programski blok

Za podučavanje programera početnika mogu se koristiti računanom kontrolirani modeli jer mogu pobuditi interes učenika [22]. Moguće je programiranje bez poznavanja hardvera koji koristimo. Iako su prethodna dva aspekta pozitivna ovaj pristup podučavanja programiranja može imati i svoje negativne stvari zbog nepredvidljivosti ponašanja fizičkog objekta u odnosu na simulaciju, a isto tako i interes za učenjem programiranja robota može prerasti u želju za igranjem s tim robotom zanemarujući obrazovni aspekt.

Ostalo

Prethodno navedeni alati su jedni od najpoznatijih u svijetu, ali vrijedno je spomenuti još neke pridonose uključivanju izrade igara u nastavu:

- SAGE (eng. *A Simple Academic Game Engine*), jednostavni okvir za izradu 3D igara namijenjen za podučavanje studenata programera početnika
- Kodu, je vizualni programski jezik Microsoftovog istraživanje za petogodišnjake koji žele raditi igre za PC i Xbox 360
- Code Monster from Crunchzilla, web stranica napravljena u obliku tečaja koja omogućuje upoznavanje s jednostavnim JavaScript naredbama za izradu igara
- Knjiga 3D Game Programming for Kids: Create Interactive Worlds with JavaScript (Chris Storm)
- Unity 3D, već spomenut okvir za izradu igara prilagodljiv profesionalcima i početnicima, korišten u obrazovanju. Podupire ga knjiga Unity 3D Game Development by Example Beginner's Guide (Ryan Henson Creighton)

5. Istraživanje

U nastavku iznesen je cilj te očekivanja provedenog istraživanja iz naslova samoga rada. Dalje je prikazan način na koji se pristupilo istraživanju te u konačnici rezultati i dodatna zapažanja.

5.1. Cilj i očekivani rezultati

Do sada studenti nisu pokazivali prevelike ambicije za izradu igara možda iz razloga jer smatraju to nečime izvan njihovih sposobnosti. Osnovni cilj ovog istraživanja je poticanje motivacije kod studenata uporabom okvira za izradu igara. Uz to želimo pokazati kako igra ne mora biti nešto kompleksno nego čak nešto zanimljivo. Ako igrom uspijemo pobuditi interes studenata onda bih trebao i njihov angažman za projekt biti veći te na taj način pristupiti programiranju na hrabriji način.

Osim osnovnog cilja koji je će u konačnici biti evaluiran ovim istraživanjem postoji još jedna skrivena namjera ovoga istraživanja, a to je da studente (buduće nastavnike informatike) nesvjesno podučavamo zanimljivim pristupima podučavanja programiranja koje u ovom slučaju uključuje video igre i ilustracije koje podupiru teorijski dio programiranja.

Očekivani rezultati su da ćemo poticanjem studenata da izrađuju igre povećati motivaciju pri učenju, da će više studenata odabirati projekte temeljene na izradi igara. Poželjni rezultati bi bilo da će ti isti studenti koji su odabrali projekte temeljene na igrama ostvariti bolji uspjeh na kolegiju OOP.

Iako ne postoje alati za mjerenje motivacije, kroz ovo istraživanje usporediti ćemo strukturu projekta onih koji biraju kompleksnija (2D ili 3D) okruženja od onih koji biraju jednostavne okvire za izradu igara ili onih koji izrađuju igre upotrebom nekih drugih biblioteka.

Nulta i alternativna hipoteza za uspjeh na kolegiju OOP koji ne/ovisi o vrsti projekta:

- $H0_1$ – ne postoji statistički značajna razlika u uspjehu na kolegiju OOP između skupine koja je izrađivala igre i skupine koja je izrađivala projekte tematski nevezane za igre
- $H1_1$ – postoji statistički značajna razlika u uspjehu na kolegiju OOP između skupine koja je izrađivala igre i skupine koja je izrađivala projekte tematski nevezane za igre

Nulta i alternativna hipoteza za angažman studenta koji je vidljiv iz programskog koda:

- $H0_2$ – ne postoji značajna razlika u količini, vlastitog, nadograđenog dijela programa između skupina koje su radile igre bez okvira, sa Game SDK (jednostavnim) okvirom i naprednim okvirom za izradu igara
- $H1_2$ – postoji značajna razlika u količini, vlastitog, nadograđenog dijela programa između skupina koje su radile igre bez okvira, sa Game SDK (jednostavnim) okvirom i naprednim okvirom za izradu igara

5.2. Pristup korišten za istraživanje

Za dosadašnja predavanja na kolegiju OOP korišteni su primjeri programskog koda igara te je postojala opcija izrade igre za projekt. U svrhu istraživanja napravljen je korak dalje. Kolegij smo odlučili velikim dijelom posvetiti igrama tako da su za teorijski dio nastave izrađene ilustracije na temu igre World of Warcraft uz pisana objašnjenja istih ilustracija. Prvi dio vježbi organiziran je preko zadataka kojima se podučavaju određeni OOP koncepti dok je drugi dio vježbi posvećen igrama. Za dio vježbi koji je posvećen igrama izrađene su igre u dva različita okvira za razvoj igara, a to su:

- Unity game engine (2D i 3D)
- Game SDK izrađen uz pomoć Windows Forms Applications (2D)

Za primjer igre izrađene u Unity okviru napisan je detaljan dokument koji ima ulogu potpore pri izradi igre i upoznavanja s Unity okvirom. Dokument je napisan iz razloga što je Unity okvir dosta teži od onoga s čime su se studenti do sada susretali na nastavi programiranja.

Sve igre, bez obzira na odabrani okvir, pisane su u C# jeziku koristeći Visual Studio i MonoDevelop. Svi studenti imaju pravo izbora teme projekta, a ako odaberu temu igara imaju također pravo izbora okvira u kojem žele izrađivati igre. Također, studenti mogu, uz ponuđene okvire, odabrati i neke alternativne.

Za drugi dio vježbi nastavnik treba kratko uvesti studente u ponuđene okvire te do kraja semestra biti na raspolaganju za bilo koji problem na koji naiđu pri izradi projekta. Studenti mogu to vrijeme iskoristiti za upoznavanje s okvirima za izradu igara te izradu vlastitog projekta.

Temelj na kojem će se zasnivati rezultati ovog istraživanja biti će studentski projekti. U istraživanju sudjelovalo je 144 studenata 2. godine preddiplomskog studija različitih usmjerenja i to: informatika, informatika i tehnika, matematika i informatika te fizika i informatika. Uzorak studenata sastoji se od studenata s dvije različite akademske godine:

- 71 student iz akademske godine 2013-2014
- 73 studenta iz akademske godine 2014-2015

Osim podataka o odabranim temama projekata napravljena je analiza programskog koda svih projekata temeljenih na igrama. Motiviranost studenata ćemo provjeriti analizom projekata tako da ćemo u obzir uzeti način na koji je projekt organiziran te sam opseg i težina projekta. Samim odabirom projekta temeljenog na izradi igre projekt dobiva na težini i opsegu što znači da student treba uložiti puno više vremena i napora za izradu istog. Sav taj uloženi trud i vrijeme bi značio kako je student bio jako motiviran da nauči nešto i prihvati svaki izazov.

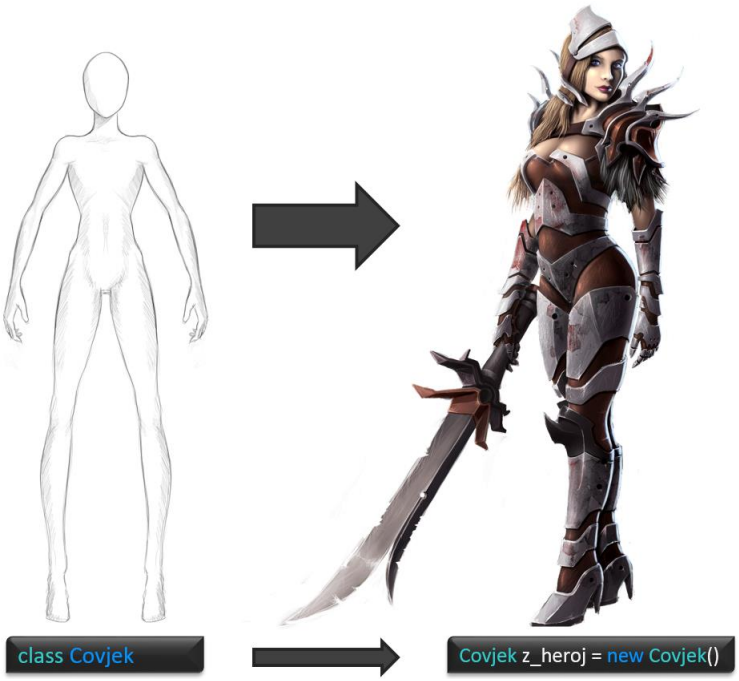
5.2.1. Alati koji su na raspolaganju studentima

Svi studenti imali su na raspolaganju Visual Studio i Monodevelop okruženja za pisanje programa. Sve igre na kolegiju OOP su pisane u C# programskom jeziku te su mogle biti pisane u bilo kojem od navedena dva okruženja.

Osnovi koncepti

Za teorijski dio kolegija izrađena je posebna skripta u kojoj je svaki osnovni koncept objektno-orijentiranoga programiranja opisan vlastitim ilustracijama koje su temeljene na igrama. Za temu odabrana je igra World Of Warcraft kako bi sve ilustracije bile tematski povezane. U samom uvodu opisan je svijet u kojem se nalaze svi likovi koji su kasnije predstavljeni na ilustracijama.

Skripta uključuje koncepte klase, objekta, imenskih prostora, polja, metoda, konstruktora, nasljeđivanja, prava pristupa, učahurivanja, polimorfizma i kratak uvod o kontrolama i događajima.



Sl. 5.1 Vizualna reprezentacija klase i objekta

Na slici (Sl. 5.1) prikazan je primjer ilustracije iz navedene skripte gdje je predstavljena usporedba koncepta klase i objekta. Način na koji je klasa predstavljena je skica koja vrijedi za svakoga čovjeka dok je objekt predstavljen kao živuća osoba koja ima vlastiti identitet i sposobnosti.

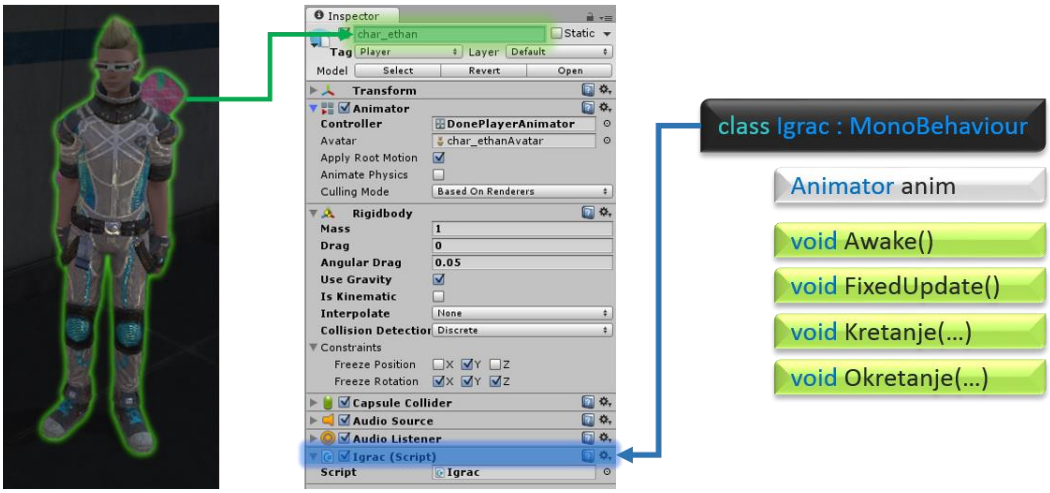
Sljedeći korak, koji na žalost nije napravljen u ovom istraživanju bi bile animacije istih likova pa čak i izrada igre s istim likovima kao i u navedenoj ili animiranoj skripti.

Unity igra

Za upoznavanje s Unity okvirom za izradu igara također je napravljena posebna skripta koja se odnosi na igru koja je preuzeta s unity web stranice pod nazivom „Stealth“. Sam proizvođač Unity okruženja nudi igru „Stealth“ kao primjer za podučavanje programiranja igara. Igra je analizirana s aspekta programskog koda te prilagođena za studente tako da su izbačeni nepotrebni koncepti. Osim toga među objektima su stvorene jasnije veze. Skripta se sastoji od 7 poglavlja koja se većinom odnose na pojedini objekt u igri.

Prvo poglavlje odnosi se na kratki uvod u razvojno okruženje gdje su predstavljene samo osnovne (potrebne) funkcionalnosti Unity alata. Ovaj dio je potrebno skratiti kao bi se više vremena posvetilo programiranju naspram upoznavanja s alatom. Prve klase koje student izradi ne uključuju naprednije koncepte poput nasljeđivanja i polimorfizma. Svako sljedeće poglavlje uvodi nove klase kako bi se predstavio novi koncept objekto-orijentiranog

programiranja. Svako poglavlje počinje kratkim podsjetnikom na koncept o kojem govorimo. Zatim predstavljamo listu klasa iz Unity biblioteke kao pomoć pri opisivanju objekta iz igre. Nakon toga je predstavljen lik iz igre kao na slici (Sl. 5.2) kojeg podupire okvirni sadržaj klase.



Sl. 5.2 Objekt igre i klasa koja ga opisuje

Nakon predstavljenog objekta iz igre, dana je lista aktivnosti koje je potrebno odraditi kako bi se napisala klasa u cijelosti uz opis poželjni ponašanja objekta u igri.

Dio igre koju student prema uputama može izraditi sastoji se od 14 klasa koje imaju različitu namjenu što u igri što u opisu različitih koncepata objekto-orijentiranog programiranja.

Game SDK

Osim naprednog okruženja (Unity) studenti imaju na raspolaganju i jednostavni okvir za izradu igara izrađen uz pomoć biblioteka „Windows Forms“ gdje je ugrađena funkcionalnost dinamičkog stvaranja 2D objekta te njihovo upravljanje preko događaja.

Kao što je vidljivo iz slike (Sl. 5.3), Game SDK služi za izradu jednostavnih igara gdje nije potrebno trošiti vrijeme na upoznavanje s sučeljem jer se igre rade izravno iz Visual Studio okruženja s kojim su studenti već prethodno upoznati.

Sl. 5.3 Primjer jednostavne 2D igre izrađene u Game SDK

Za Game SDK postoji nekolicina gotovih (manjih i većih) projekata te opisa tih istih projekata stoga nisu izrađivane posebne skripte kao kod Unity okruženja.

5.3. Rezultati

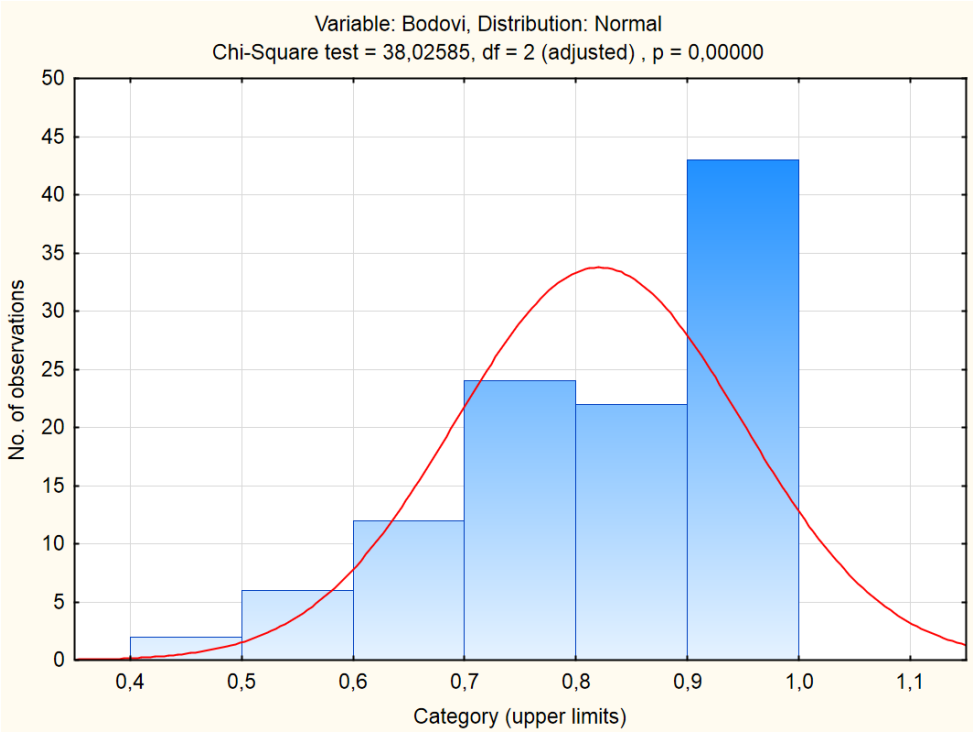
U ovom poglavlju predstavljeni su rezultati istraživanja. Prvi dio odnosi se na testiranje hipoteza postavljenih u uvodnom poglavlju, a drugi dio odnosi se na dodatne zanimljive podatke zapažene tijekom istraživanja.

5.3.1. Statističko testiranje hipoteza

U svrhu ispitivanja prve nulte hipoteze:

$H0_1$ – ne postoji statistički značajna razlika u uspjehu na kolegiju OOP između skupine koja je izrađivala igre i skupine koja je radila na drugoj vrsti projekta

ispitanici su podijeljeni u tri skupine od kojih su samo dvije skupine bile uključene u statističke testove. Prva skupina su studenti koji nisu odabrali projekt, druga skupina su studenti koji su izrađivali igru i treća skupina su studenti koji su odabrali temu projekta nevezanu za igru. Za testove odabrane su skupine studenata koje su odabrale projekt bilo tematski povezano s igrama ili ne. Skupina koja nije radila projekte je izostavljena iz statističke analize jer za testiranje navedene hipoteze podatci su nevaljani.



Sl. 5.4 Raspodjela ukupnog uspjeha studenata na kolegiju OOP promatrana u postotcima

Prvi test izvršen je u svrhu ispitivanja normalnosti raspodijele podataka prema Gaussovoj krivulju. Kao što je vidljivo iz slike (Sl. 5.4) podatci ne slijede krivulju normalne raspodijele te stoga treba koristiti neparametrijske testove za ispitivanje hipoteza.

Mann-Whitney U test je pokazao da nema statistički značajne razlike (Tablica 5.1) u dvije odabrane skupine stoga se prihvaća nulta hipoteza:

$H0_1$ – ne postoji statistički značajna razlika u uspjehu na kolegiju OOP između skupine koja je izrađivala igre i skupine koja je izrađivala projekte tematski nevezane za igre

Tablica 5.1 Rezultati Mann-Whitney U testa

variable	Mann-Whitney U Test By variable Uspjeh Marked tests are significant at p<,05									
	Rank Sum I	Rank Sum P	U	Z	p-value	Z adjusted	p-value	Valid N I	Valid N P	2*1sided exact p
Uspjeh (%)	2490,500	3504,500	1158,500	1,69995	0,41564	1,470012	0,141560	41	68	0,141221

Skupine prikazane u tablici su označene kraticama (I) i (P) gdje (I) označava projekte temeljene na igrama, a (P) projekte tematski nevezane za igre.

U svrhu testiranja druge nulte hipoteze:

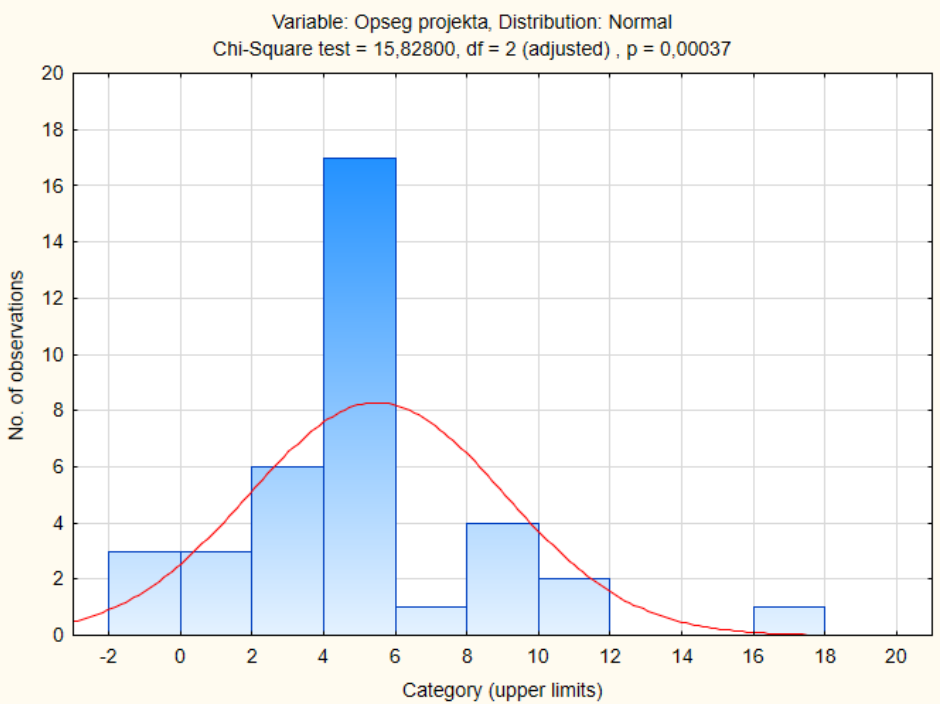
$H0_2$ – ne postoji značajna razlika u količini, vlastitog, nadograđenog dijela programa između skupina koje su radile igre bez okvira, sa Game SDK (jednostavnim) okvirom i naprednim okvirom za izradu igara

provedena su dva neparametrijska statistička testa. U ovom slučaju promatramo populaciju studenata koji su odabrali igre kao projekt. Ti studenti podijeljeni su u tri kategorije:

a. studenti koji su izrađivali igre uz pomoć jednostavnog okvira (Game SDK),
b. studenti koji su izrađivali igre uz pomoć naprednog okvira (Unity, XNA i dr.),
c. studenti koji su izrađivali igre bez korištenog okvira za izradu igara.

Obadva testa provedena su samo na 37 studenta zbog nedostupnosti četiriju projekata.

Prvi test odnosi se na ukupan opseg projekta gdje opseg projekta opisuje količina funkcionalnih klasa koje je student ugradio u vlastiti projekt. Količina korištenih koncepata poput nasljeđivanja, učahurivanja i sl. je zanemarena jer je to ionako sadržano u tim klasama.



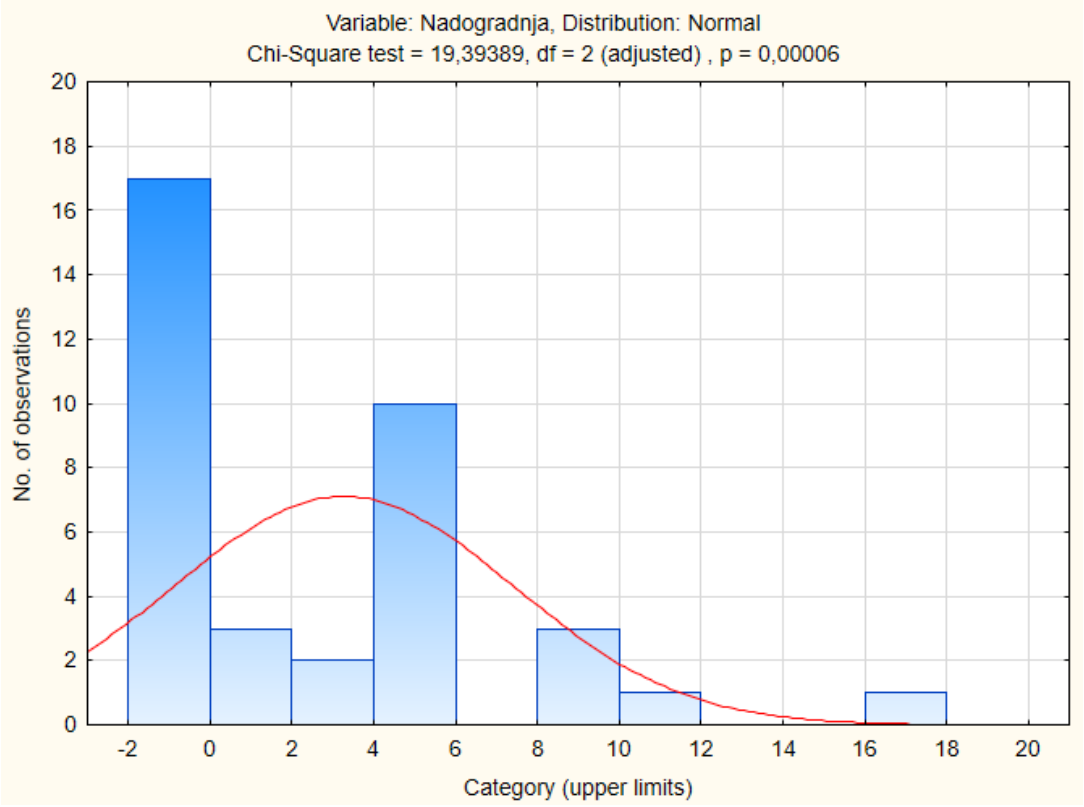
Sl. 5.5 Raspodjela broja ugrađenih klasa u projekte

Testirani podatci (Sl. 5.5) ne prate normalnu krivulju stoga je korišten test prikazan u tablici (Tablica 5.2). Podatci iz toga testa govore kako nema statistički značajne razlike u opsegu projekta između grupa A, B i C što je vidljivo iz tablice (Tablica 5.2).

Tablica 5.2 Rezultati testa za opseg projekta

Depend: Opseg projekta	Kruskal-Wallis ANOVA by Ranks; Opseg projekta Independent (grouping) variable: Okvir Kruskal-Wallis test: H (2, N=37) =,6340054 p=,7283		
	Valid N	Sum of Ranks	Mean Rank
A	8	145,5000	18,18750
B	6	133,0000	22,16667
C	23	424,5000	18,45652

Za drugi test u obzir uzimamo stvarni broj nadogradnji studenta. To se odnosi na one klase koje je student sam pisao. Neki studenti su izrađivali igre tako da prekopiraju programski kod gledajući tutorska videa koristeći Internet. Takvim projektima je broj nadogradnji označen brojem 0.



Sl. 5.6 Raspodjela broja ugrađenih klasa u projekte gdje je student pisao vlastiti programski kod

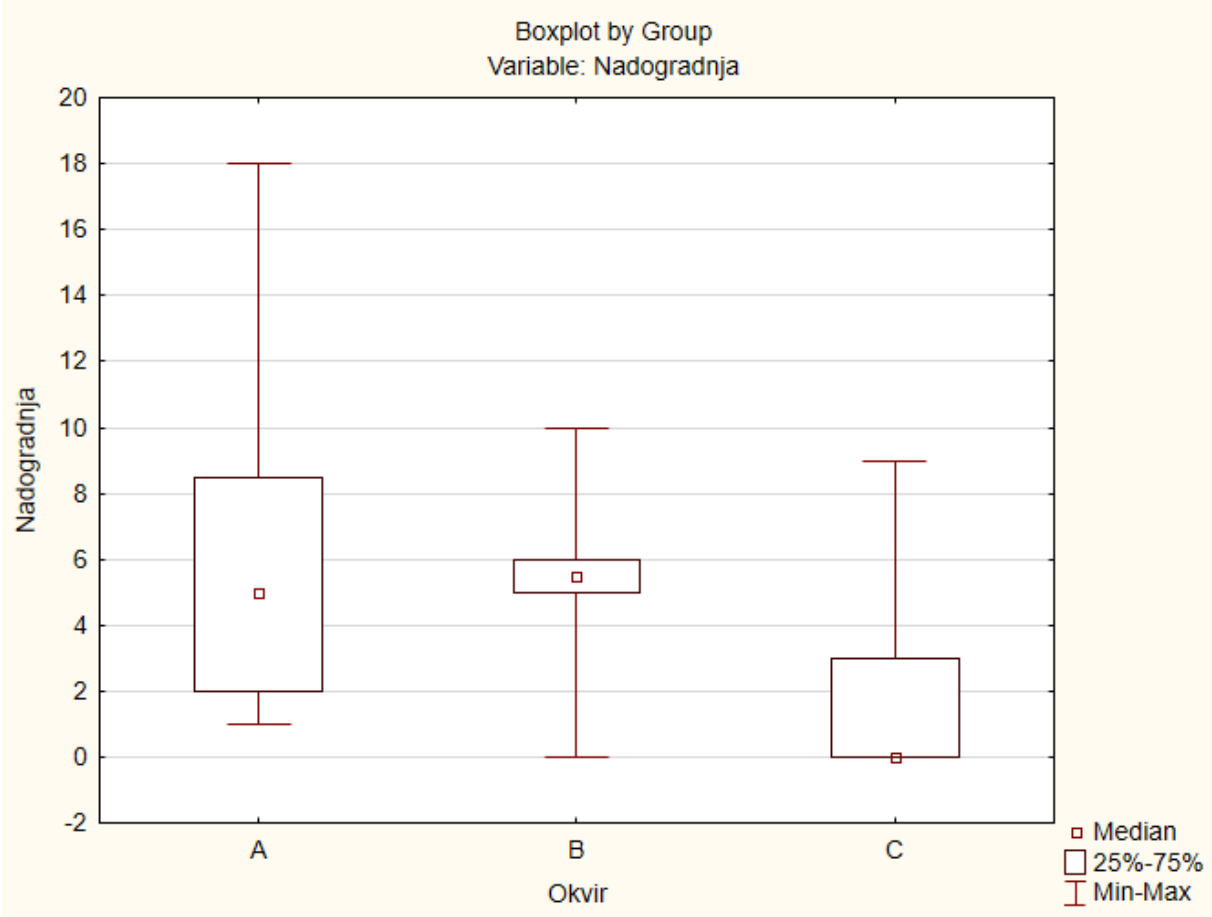
U ovom slučaju podatci također ne slijede normalnu krivulju (Sl. 5.6). Za testiranje nulte hipoteze korišten je test prikazan u tablici (Tablica 5.3), a rezultati pokazuju kako postoji statistički značajna razlika između skupina A,B i C.

Tablica 5.3 Rezultati testa za stvarnu nadogradnju koju je student realizirao

Depend: Nadogradnja	Kruskal-Wallis ANOVA by Ranks; Nadogradnja Independent (grouping) variable: Okvir Kruskal-Wallis test: H (2, N=37) =12,07279 p=,0024

	Valid N	Sum of Ranks	Mean Rank
A	8	213,0000	26,62500
B	6	158,0000	26,33333
C	23	332,0000	14,43478

U konačnici to bi značilo kako su studenti koji su koristili bilo kakav okvir za izradu igara više projekta samostalno izradili, a to pokazuju rezultati testa (Tablica 5.3) i grafika sa slike (Sl. 5.7).



Sl. 5.7 Grafički prikaz količine vlastito ugrađenih klasa u projekt

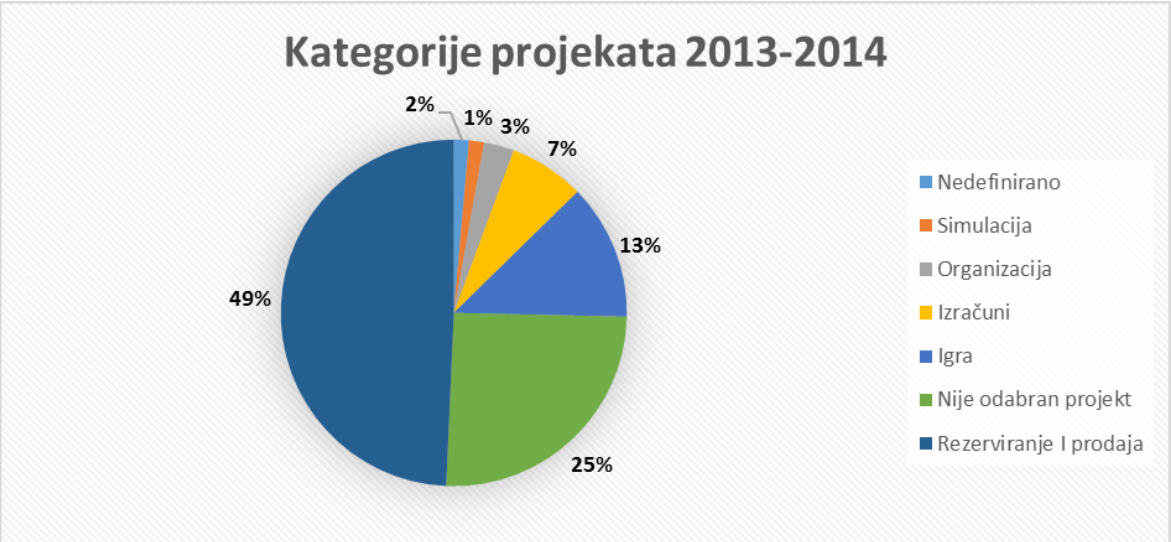
Iz same slike (Sl. 5.7) vidi se velik razlika u dane tri grupe gdje grupa C najviše odstupa što je bio predviđeni rezultat. Prema tome možemo odbiti nultu hipotezu te prihvatiti alternativnu hipotezu:

$H1_2$ – postoji značajna razlika u količini, vlastitog, nadograđenog dijela programa između skupina koje su radile igre bez okvira, sa Game SDK (jednostavnim) okvirom i naprednim okvirom za izradu igara

5.3.2. Dodatna zapažanja

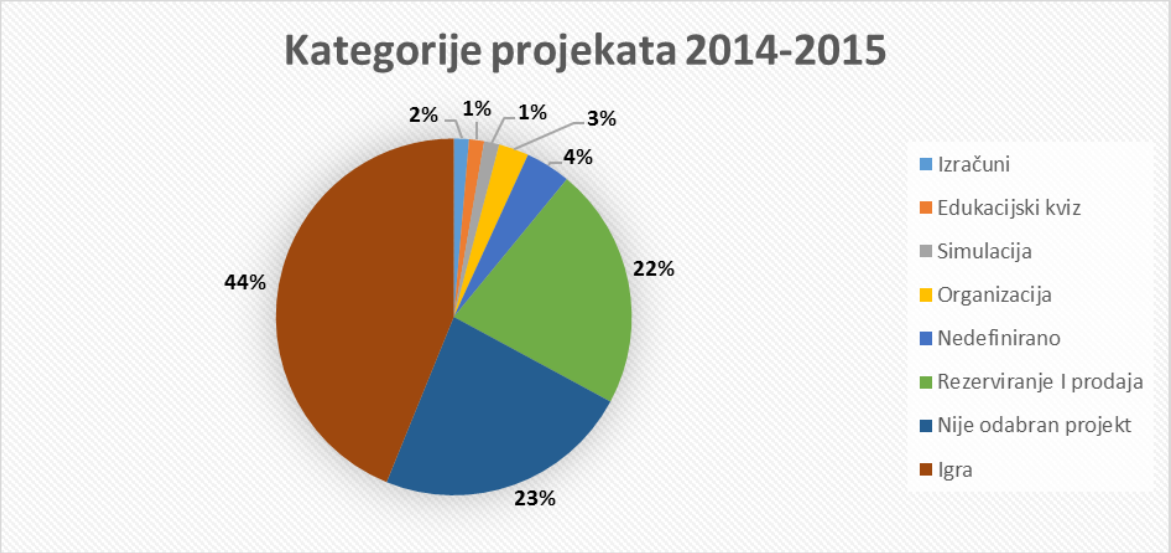
Studenti su jako dobro prihvatili ilustracije koje su korištene za objašnjavanje osnovnih koncepata, ali su isto tako pokazali jako negativne reakcije na projekt izrade igre u naprednom okviru poput Unity 2D/3D.

Grafički prikazi sa slika (Sl. 5.8 i Sl. 5.9) prikazuju veliki odskok u zainteresiranost studenata za izradom projekata na bazi igara.



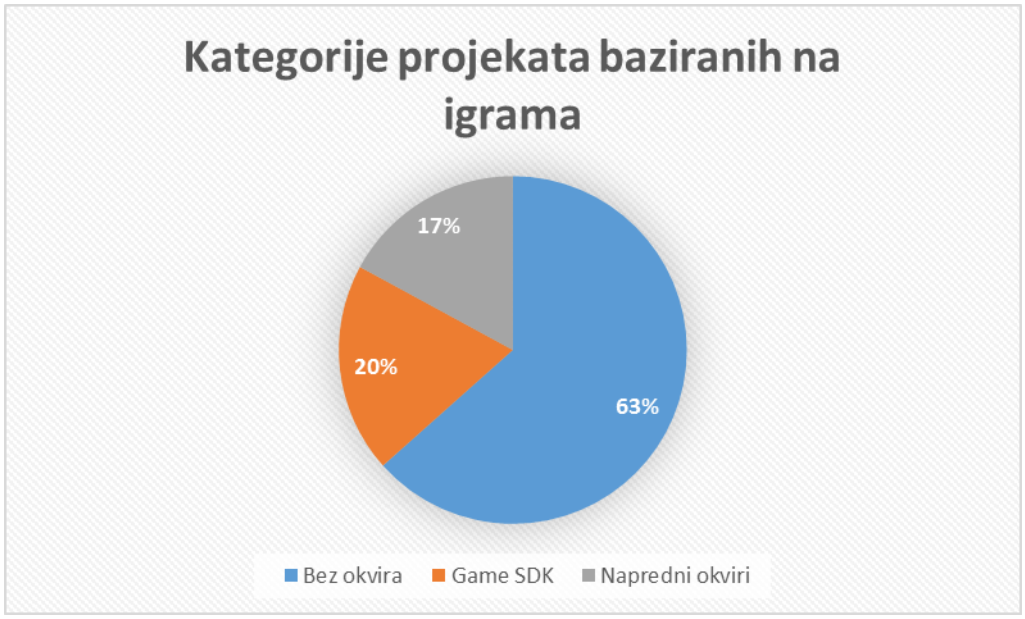
Sl. 5.8 Kategorije projekata za akademsku godinu 2013-2014

Također, iz priloženih slika (Sl. 5.8 i Sl. 5.9) vidi se smanjenje projekata na temu rezerviranje i prodaje koji su na početku ovoga rada opisani bezidejnima i nekreativnima. Ovo je također jedan od željenih rezultata istražaivanja.



Sl. 5.9 Kategorije projekata za akademsku godinu 2014-2015

Na sljedećoj slici (Sl. 5.10) vidi se koliki postotak studenata odabire pristup na koji želi izrađivati igru.



Sl. 5.10 Raspodjela studenata u odnosu na to jesu li koristili okvir za izradu igara

Dodatna zanimljivost je da se više ženske populacije odlučilo na izradu igara s tim da trebamo imati na umu kako je više ženske populacije pohađalo kolegij OOP (Tablica 5.4). Unatoč tomu većina muškaraca izrađuje igre u nekom od okvira dok više žena izrađuje igre tako da gradi program koristeći tutorska videa (isti programski kod i rješenja kao kod tutorskih videa).

Tablica 5.4 Odnos muškaraca i žena ovisno o odabiru načina polaganja OOP-a

Spol	Projekt igre	Ostali projekti	Bez projekta	Korišten okvir za izradu igara
M	18	21	16	13
Ž	23	47	19	2

Još jedna zanimljivost je da studenti koji su koristili okvir za izradu igara ostavljaju puno više i dosta opsežnije komentare vlastitoga koda u odnosu na one studente koji su nisu koristili okvire. U tablici (Tablica 5.5) možemo vidjeti kako većina studenata koji izrađuju naprednije igre ostavljaju komentare u vlastitom kodu dok druga skupina nema istu naviku

Tablica 5.5 Broj studenata koji komentiraju vlastiti programski kod

Okvir za izradu igara	Nema komentara	Komentirani samo ključni dijelovi	Detaljni komentari
Game SDK	18	4	1
Napredni	6	3	5

Od 42 ispitanika samo 1 ispitanik odabire 3D igru dok samo nekolicina pokazuje interes za izradom 3D igara.

Tablica 5.6 Broj studenata ovisno o organizaciji projekta (Igre)

Organizacija projekta	Broj studenata
Centralizacija unutar jedne klase	4
Podjela na funkcionalne module bez hijerarhije klasa	9
Podjela na funkcionalne module s jasno definiranom hijerarhijom klasa (nasljeđivanjem)	20
Podjela na funkcionalne module s nasljeđivanjem bez smisla	2

Većina studentskih projekata posjeduje hijerarhiju klasa koja ima smisla i gdje je razumljiv razlog korištenja nasljeđivanja. Centralizacija svih funkcionalnosti igre unutar jedne klase je rijetko zastupljena što se vidi iz priloženoga (Tablica 5.6).

6. Rasprava

Cjelokupno istraživanje pokazalo se djelomično uspješno, ali ne zbog dobivenih rezultata nego zbog određenih prepreka koje su se stvorile tijekom istraživanja. Sami rezultati istraživanja su zadovoljavajući iako nije svaki detalj predviđen.

Za početak prodiskutirajmo prepreke koje su se postavile pred nas. Veliki trud i vrijeme uloženo je u pronalazak adekvatne igre koja bi mogla biti dobar i zanimljiv primjer za studente, a još više truda i vremena na sastavljanje dokumentacije koja je služila kao pomoć pri učenju. Pošto su studenti imali slobodu izbora alata s kojim žele raditi, postojala je šansa kako će taj trud biti uzaludan ako studenti ne vide ništa interesantno u izradi te kompleksnije 3D igre. U konačnici se to i dogodilo, samo 3 studenta biraju napredni okvir za izradu igara (Unity) za koji je napravljena potpuna dokumentacija. Već tu možemo primijetiti kako studenti nemaju velikoga interesa da odaberu nešto izazovno. Je li to strah da ne otežaju već po sebi težak predmet koji je programiranje, točnije objektno-orijentirano programiranje, pitanje je za možda neko drugo istraživanje. Na kraju je jedno sigurno, naši studenti još sami nisu spremni da naprave taj iskorak i prihvate izazov čiji rezultat može biti nešto zanimljivo i bogato novim znanjima i vještinama.

Možemo se pitati je li takav ishod bio predvidljiv i je li bilo potrebno toliko truda ulagati u nešto što možda nitko neće pokušati naučiti? Na neki način odgovor bi bio da jer je nama kao nastavnicima cilj da uvijek pronalazimo nove načine kako pobuditi motivaciju naših učenika/studenata te im pokazati što to ima u tom svijetu kojeg su sami odabrali kada su odlučili baviti se informatikom. Naravno da ni mi kao nastavnik uvijek nećemo biti uspješni, ali je važno da mi ne odustajemo od onih koje podučavamo kako ni oni ne bih odustali od samih sebe.

U ovom slučaju studenti nisu pokazali veliki interes za izradom za njih kompleksnije igre, ali je ipak iz svega toga proizašao zanimljiv podatak da studenti žele izrađivati igre, a to ne ovisi o nekom posebnom okviru. Ovaj podatak nam govori kako su studenti ipak otvorili ruke igrama jer im se to čini zanimljivijim od nekog projekta gdje će napraviti nekoliko klasa koje će oponašati skladište iz kojeg ćemo naručivati proizvode.

Iz rezultata je vidljiv veliki porast onih studenata koji odabiru igre za projekte iz objektno-orijentiranoga programiranja što se i predviđalo da će se dogoditi. Ova informacija može biti značajna za one koji žele obaviti daljnja istraživanja na sličnu temu tako da ispituje što to studente točno zanima kada su igre ili općenito projekti u pitanju te zašto odabiru baš igre, da li zato što inače vole igrati igre ili zbog zanimljivosti koje pridodaju programiranju ili pak nekog trećeg razloga.

Iz rezultata smo mogli vidjeti kako, unatoč predviđanjima, nema statistički značajne razlike u uspjehu na kolegiju između onih studenata koji su izrađivali igre i onih koji nisu izrađivali igre. Iz toga možemo zaključiti kako je i jedna i druga grupa studenata savladala gradivo OOP-a jednako dobro, ali u tu moramo uzeti u obzir činjenicu kako su igre izrađivane u Game SDK okruženju koje je izvedeno iz Windows formi u kojima je ostatak studenata izrađivao svoje projekte. Ta činjenica nam je bitna jer nismo bili u mogućnosti kvalitetno istražiti kakav utjecaj napredniji okvir za izradu igara ima na uspjeh studenata jer je jako mali dio studenata odabrao napredni okvir za izradu igara.

Također, iznenađujući je i podatak da su neki od studenata izabrali nekakav alternativni, kompleksniji okvir za izradu igara što nam govori da među studentima informatike postoji nekolicina onih koji su već zakoračili u svijet izrade igara.

Ispitivanje opsega projekta pokazalo je zadovoljavajuće rezultate. U prvom slučaju pregledani su svi projekti na temu igara te zabilježen opseg svakog projekta uključujući različite koncepte objektno-orijentirane paradigme. Za test odabran je broj zabilježenih klasa u projektu jer na klasama počivaju cijeli projekti, a unutar tih klasa su sadržani ostali koncepti. Rezultati su prihvatljivi jer sve izrađene igre sadrže otprilike isto različitih objekata, likova i sl. koji su opisani vlastitim klasama. Treba imati na umu kako u ovom slučaju opširnost projekta ne znači količina programskog koda kojom opisujemo ponašanja programa jer bi u tome slučaju možda postojala statistički značajna razlika jer u naprednijim okvirima za izradu igara opisujemo kompleksnija ponašanja posebno ako se radi o 3D igrama.

Za nastavak ispitivanja, u svrhu odgovaranja na pitanje tko se najviše potrudio pri izradi igre, bilo je potrebno zabilježiti porijeklo projekta te modificirati podatke na kojima se ispitivala opširnost projekta. Pod porijeklo projekta smatra se tko je napravio projekt tj. od koga su proizašle klase koje su sadržane u igri. Jedan dio studenata nije sam smislio igru i izrađivao vlastite klase nego izrađivao igre prema tutorskim videima koji su vodili studenta

kroz proces izrade igre. U ovom slučaju je logično da je student bio manje uključen u proces razmišljanja, te uopće nije izrađivao koncepte kako bi igra trebala izgledati i na koji način to izraditi. Oni studenti koji su sami izrađivali igre trebaju u svoj projekt uključiti barem sljedeće korake:

- Definirati kakav tip igre žele izraditi
- Kako će ta igra izgledati i koje će sve likove uključivati
- Pribavljanje materijala za izradu igre (Slike, Zvuk, 3D modeli...)
- Izrada dijagrama klasa ili cijele hijerarhije klasa
- Pisanje programskog kod i testiranje

Ako je student koristio neki napredni okvir za izradu igara onda njegove aktivnosti uključuju još i:

- Upoznavanje s okvirom za izradu igara (funkcionalnosti i korisničko sučelje)
- Upoznavanje s bibliotekama tj. hijerarhijom klasa koje su mu na raspolaganju za izradu igre
- Pronalazak stereotipnih rješenja za probleme s kojima se susreće jer je za programera početnika to nepoznato područje (stereotipna rješenja mogu postojati u dokumentaciji samoga okvira za izradu igara ili na forumima gdje se raspravlja o određenim problemima s kojima se programeri igara susreću)

Iz priloženoga vidimo kako oni studenti koji se odlučuju na praćene takozvanih tutorskih videa ulaze u puno manje aktivnosti od onih koji sami izrađuju igre, a jedan od razloga zašto obavljati sve ove aktivnosti ako možemo samo „prekopirati“ kod iz tutorskih videa je motivacija. Motiviran student će puno više vremena potrošiti savladavanje nečega novoga te se iznova susretati s novim izazovima koje će nastojati savladati dok će nemotiviran student prije izabrati onaj kraći put gdje će preskočiti sve najvažnije aktivnosti te dobiti jednako kompleksan proizvod kao i netko tko je sam gradio program iz vlastitih ideja.

Nakon ponovnog testiranja s nešto izmijenjenim podatcima gdje je u obzir uzeto porijeklo projekta dobijemo nešto drukčiju sliku. Potvrđujemo vlastite pretpostavke kako će studenti koji odaberu okvire za izradu igara uložiti puno više truda u izradu igre te da će veliki ili cijeli dio programskog koda igre sami osmisliti i napisati. Ovakav rezultat je veliki uspjeh ovog istraživanja jer se pokazalo nešto što naizgled može biti neočekivano, studenti koji su odabrali zahtjevnije projekte su sebi stvorili više aktivnosti te krenuli „dužim putem“ kako

bi izradili igru umjesto da pronađu već gotove igre jednake kompleksnosti kojima svatko ima pristup na nekoliko klikova mišem.

Količina projekata baziranih na igrama značajno je porasla u odnosu na prethodnu akademsku godinu što je za ovo istraživanje pozitivan rezultat jer se uspjelo potaknuti studente da budu barem malo kreativniji pri odabiru projekata na kojima će učiti i u konačnici položiti i kolegij. Broj sličnih projekata s najčešćom temom rezervacije i prodaje opada, a to je pozitivan rezultat jer u takvim projektima slabo do izražaja dolazi prava moć programiranja te se može doći do krivog shvaćanja koncepata klase i objekta poistovjećivanjem istih s bazama podataka. Za razliku od prethodne (2013-2014) akademske godine više studenata odabire naprednije ili bilo kakve okvire za izradu igara što je također pomak.

Popularnost „*casual*“ igara pokazala se prisutna i unutar ovog istraživanja gdje skoro sve izrađene igre izgledaju kao da su rađene za mobilne uređaje. „*Casual*“ igra nije nužno 2D igra, ali u ovom istraživanju skoro sve igre dolaze u 2D formatu što potvrđuje rezultate prethodno spomenutih istraživanja. Ako se prisjetimo ta istraživanja preporučuju programiranje 2D igara za programere početnike naspram kompleksnih 3D igara. Ovo također može biti zanimljiv podatak jer se pokazao interes za mobilnim aplikacijama (igrama) što može biti tema nekog budućeg istraživanja ili prijedlog nastavnicima ukoliko žele u vlastiti nastavni plan i program uključiti izradu igara. Slično istraživanjima koja nude statistiku o igrama u svijetu, otprilike jednak broj žena i muškaraca bira projekte bazirane na igrama s tim da je ženska populacija privrženija „*casual*“ igrama.

Također smo mogli vidjeti iz rezultata (Tablica 5.6) kako studenti razmišljaju o organizaciji vlastitog programa te stvaraju smislene hijerarhije klasa te dijele program na funkcionalne module što je u duhu objektne orijentacije. Možda je tome razlog dobra praksa, a isto i sama igra koja zahtjeva necentralizirani sustav klasa kako bi se kvalitetno opisali pojedini dijelovi igre. Naravno, moguće je i unutar igre napraviti centraliziran sustav komunikacije objekata, ali taj slučaj nije primijećen u sagledanim projektima. Imajmo na umu i činjenicu kako je dio projekata preuzet iz tutorskih videa koji opisuju igre izrađene o strane iskusnog programera.

Zaključak

Iz ovog pa tako i svakog drugog istraživanja možemo zaključiti kako niti jedan ishod ne možemo u potpunosti predvidjeti, ali unatoč tomu uvijek možemo pokušavati pronalaziti nove, bolje načine kako poboljšati obrazovanje. Konkretno iz ovog istraživanja možemo izvući poruku kako treba detaljno sagledati trendove igara ako ih želimo uključiti u nastavu. To podrazumijeva istraživanje o svjetskim trendovima i ispitivanje vlastitih studenata što oni žele. Na neki način je ovo istraživanje i krenulo u tom smjeru gdje smo dobili rezultate koji ukazuju na postojanje interesa za igre, ali čak na poseban tip igara tj. 2D „*casual*“ igara. Okviri za izradu igara itekako imaju mjesta u obrazovanju što pokazuju razna istraživanja pa tako i ovo istraživanje. 3D igre su zasigurno previše za programera početnika, a za pripremu materijala se troši previše vremena. Za programere početnije bolje je krenuti na 2D igre i to u nekom od okvira za izradu igara koji je namijenjen u obrazovne svrhe gdje će naglasak biti više na igri kao programerskom proizvodu nego li na samome okviru. Tek u daljnjem obrazovanju možemo nuditi naprednije okvire za izradu igara kao alat za izradu projekata. Projekti igara su dosta raznovrsni tematikom, tipom igre, funkcionalnostima što je dobro jer se pokazuje kreativnost studenata. Igre mogu biti jako kompleksni sustavi te se mogu uključiti u razna područja poput umjetne inteligencije, interakcije čovjeka i računala, računalnih mreža itd. Taj podatak nam govori kako neka buduća istraživanja mogu pokušati integrirati okvire za izradu igara u više kolegija. Ovo istraživanje je samo početak koji otvara vrata drugim istraživanjima koja žele ispitivati vezu između interesa i motivacije studenata kad je u pitanju uporaba okvira za izradu igara.

Literatura

[1] PÄIVI KINNUNEN , LAURI MALMI, Why students drop out CS1 course?, *Proceedings of the second international workshop on Computing education research*, 9.9.2006-10.9.2006, Canterbury, United Kingdom

[2] MARIA FELDGEN, OSVALDO CLÚA, „Games As A Motivation For Freshman Students To Learn Programming“, *34th ASEE/IEEE Frontiers in Education Conference*, 20.10.2004 – 23.10.2004, Savannah, GA

[3] JENKINS, T., "The Motivation of Students of Programming", *SIGCSE Bulletin, Vol. 33 No. 3*, Rujan 2001, pp 53-56.

[4] ENTWISLE, N., "Motivation and Approaches to Learning: Motivating and Conceptions of Teaching", *Motivating Students, Kogan Page*, 1998.

[5] FALLOWS, S., AND AHMET, K., „Inspiring Students: Case Studies in Motivating the Learner.“, *Kogan Page*, 1999

[6] MARI NAKAMURA, „Motivate Students, Accelerate Learning“

[7] CAITLIN KELLEHER, RANDY PAUSCH, „Using Storytelling to motivate programming“, *Communications of the ACM*, Srpanj 2007/Vol. SO, No. 7

[8] SYLVIA BEYER, KRISTINA RYNES, JULIE PERRAULT, KELLY HAY, SUSAN HALLER, „Gender Differences in Computer Science Students“,

[9] PHYLLIS C. BLUMENFELD, ELLIOT SOLOWAY, RONALD W. MARX, JOSEPH. S. KRAJCIK, MARK GUZDIAL, ANNAMARIE PALINCSAR, „Motivating Project-Based Learning: Sustaining the Doing, Supporting the Learning“, *Educational Psychologist*, 26(3 & 4), 369-398

[10] DAVID YUEN, „Project Based Learning“

[11] POLINA CHARTERS, MICHAEL J. LEE, ANDREW J. KO, DASTYNI LOKSA, „Challenging Stereotypes and Changing Attitudes: The Effect of a Brief Programming Encounter on Adults' Attitudes toward Programming“, *Proceedings of the 45th ACM technical symposium on Computer science education*, March 5.3.2014 – 8.3.2014, Atlanta, Georgia, USA

[12] ANNA ECKERDAL, Novice Students Learning of Object-Oriented Programming

[13] ELLIOT SOLOWAY, „Learning to program = learning to construct mechanisms and explanations“, *Communications od the ACM*, Volume 29, Number 9, Rujan 1986

[14] SOLOWAY, E., AND EHRLICH, K., “Empirical studies of programming knowledge“, *IEEE Trans. Softw. Eng. SE-IO. 5* (1984). 595-609.

[15] THEESA, Essential facts about the computer and video game industry, 2015 sales, demographic and usage dana, http://www.theesa.com/wp-content/uploads/2015/04/ESA-Essential-Facts-2015.pdf

[16] JONGHO JEON, KWANWOONG KIM, SOONYOUNG JUNG, A Study on the Game Programming Education Based on Educational Game Engine at School, *Department*

of Computer Science Education, Korea University, Seoul, Korea,Department of Game Development, Chunnam Techno University, Jeollanam-do, Korea

[17] CRISTIANE CAMILO HERNANDEZ, LUCIANO SILVA, RAFAEL ALENCAR SEGURA, JULIANO SCHIMIGUEL, MANUEL FERNÁNDEZ PARADELA LEDÓN, LUIS NAITO MENDES BEZERRA, ISMAR FRANGO SILVEIRA, Teaching Programming Principles through a Game Engine, *Universidade Cruzeiro do Sul*, São Paulo, Brazil

[18] BOJAN KLEMENC, PETER PEER, Introducing Game Development into the University Curriculum, *Faculty of Computer and Information Science, University of Ljubljana, Tržaška 25*, 1000 Ljubljana, Slovenia

[19] COLIN A. DEPRADINE, Using Gaming to Improve Advanced Programming Skills, *Caribbean Teaching Scholar Vol. 1, No. 2,* Studeni 2011, 93–113

[20] SELVARAJAH MOHANARAJAH, SHAN SUTHAHARAN, Learning Algorithm Design Using Casual Games – Motivating Learners by Opening Programming Logic, *Edward Waters College, University of North Carolina at Greensboro,* United States

[21] NIKO MYLLER, ROMAN BEDNARIK, ERKKI SUTINEN, MORDECHAI BEN-ARI, Extending the Engagement Taxonomy: Software Visualization and Collaborative Learning, *ACM Transactions on Computing Education, Vol. 9, No. 1, Article 7,* 2009.

[22] DAVID J. BARNES, Teaching Introductory Java through LEGO MINDSTORMS Models, *The Computing Laboratory, The University Canterbury, Kent. CT2 7NF*, United Kingdom

7. Dodatak 1 – teorijski dio

7.1. Uvod

Kroz ovaj rad razmatrati ćemo osnovne koncepte objektno orijentiranog programiranja i to uz pomoć igara, točnije igre World Of Warcraft. Za početak se možemo zapitati zašto baš objektno orijentirana paradigma i zašto baš ova igra.

Objektno orijentirano programiranje je danas najzastupljenije u softverskoj industriji zbog svih svojih prednosti koje je donijelo sa sobom. OOP je omogućilo da programeri lakše i brže pišu programe koji su jako organizirani. Također, brzina izvođenja programa je jako velika, a kao jedna od najbitnijih prednosti je mogućnost ponovnog korištenja koda (eng. *reusability*). U starim jezicima ako smo htjeli napraviti dvije slične stvari koje su dijelile dosta istih svojstava trebali smo ili mijenjati postojeći kod ili pisati novi koji je skoro pa isti. To upućuje na ponavljanje istog programskog koda na više mjesta. Posljedica toga je nepotrebna nagomilanost programskog koda. U novijim jezicima kao C#, stari kod ne trebamo ponovno pisati, a nadograđivanjem ne utječemo na taj isti stari kod i ne unosimo mogućnost pojavljivanja pogreški (eng. *bugg*) unutar koda koji se do sada točno, bez grešaka, izvodio.

U današnje vrijeme za izradu bilo kakvih aplikacija objektno orijentirano programiranje je neizbježno pa tako i za igre. Za primjer odabrana je igra World of Warcraft jer je to najbolji primjer proizvoda iskonstruiranog unutar objektno orijentirane paradigme. Ovo je igra koja je poznata svakome, i treba biti uzor onima koji žele izrađivati igre. Svoj uspjeh dokazuje ne samo zaradom koja se kreće u milionima mjesečno nego i time što je na vrhu mrežnih igara već desetljeće što još nitko nije uspio postići jednom igrom. Igra je dobar primjer dizajna s bilo kojeg aspekta, bilo to programiranja ili vizualnog dizajna. Također ova kao i mnoge druge igre su ako ne i najbolji predstavnici dobre umjetne inteligencije. Kako je tema ovog rada objektno orijentirano programiranje tako se razmatra taj aspekt dizajna.

Za lekcije u nastavku koristiti ćemo se programskim jezikom zvanim C#. On je jedan od najboljih i najpoznatijih OO jezika uz C++, Python, Java-u, Javascript… U ovom

programskom jeziku je također napisana i igra koja je uzeta kao primjer. Za pisanje programskog koda preporuča se korištenje programa zvanog Monodevelop jer je jako jednostavan i besplatan što je od velike važnosti za početnike.

Za one koji se žele baviti isključivo programiranjem igara ili simulacija preporuča se Unity jer je također besplatan i dobar za početnike, a s njim u paketu dobije se Monodevelop. Unity je program koji služi za izradu igara ili simulacija (eng. *game engine*). *Game engine* je srce ili grubo prevedeno motor koji pokreće igru u kojem je zapisano, pomoću programskog koda, kako će se svaki aspekt igre ponašati. Uz ovaj program imamo, naravno, boljih programa za izradu igara koji se mogu koristiti samo uz mjesečnu pretplatu. Također za pisanje koda možemo koristiti i Visual Studio, ali on nije besplatan.

Ova skripta služi isključivo za programere početnike koji se žele baviti programiranjem igara. Isto tako sva znanja su prenosiva na bilo koja druga područja koja uključuju objektno orijentiranu paradigmu.

7.1.1. Uvod o igri

Azeroth je planeta u svemiru Warcrafta, a dom je čovjeka i njegovih sedam kraljevstva. Također na tom istom svijetu žive i orkovi koji su se naselili s druge planete zvane Draenor. Oni su, iako ne najveći, suparnici ljudima skupa s par drugih rasa koje također nisu prijateljski nastrojene. Ipak, u teškim vremenima i jedni i drugi trebaju surađivati kako bi spasili nekima stari, nekima novi, dom od zajedničkog neprijatelja zvanog Burning Legion (grubo prevedeno goruća legija).

Prvi zadatak pri dizajniranju igara su naravno skice i konceptni crteži kojima kasnije 3D modeleri daju oblik koji će biti korišten za igru. Također tim zadužen za animaciju će napraviti prvi korak ka „oživljavanju“ tih likova. Najveću ulogu u tom takozvanom „oživljavanju“ imaju programeri koji rade na igri, a tu dolazimo do objektno orijentiranog programiranja. OOP ne postoji tek tako, ono je zapravo potreba jer bi drugim načinima programiranja bilo jako teško ostvariti ono što se u današnjim igrama može. Programer je taj koji će spojiti 3D model s animacijom, koji će napisati kako će se pojedini aspekt igre odvijati i kako će se pojedinci u toj virtualnoj stvarnosti ponašati. OOP nam omogućuje dosta realistično preslikavanje pravog svijeta u virtualni što uključuje sve načine interakcije nas s drugim ljudima i stvarima koje nas okružuju.

Kada smo definirali što se radi i tko ima koju ulogu u toj priči, potrebno je odabrati alate za izradu igre, tj. alata za preslikavanje stvarnog svijeta u virtualni. Na raspolaganju su nam prethodno navedeni alati (Unity + Monodevelop). Ovaj dokument je pisan po uzoru na Monodevelop okruženje.

7.2. Tipovi podataka i spremnici

Već smo se na neki način upoznali sa svijetom u kojem se igra WOW odvija, a sada je vrijeme da se pobliže upoznamo s likovima kojima je taj svijet dom.

Za početak nećemo krenuti s osnovnim konceptima objektno orijentiranog programiranja nego s tipovima podataka koji su korišteni u raznim programskim jezicima na koje ćemo lagano nadograđivati te iste koncepte. Imamo dva tipa podataka, a to su oni koji sadrže neku vrijednost (eng. *value type*) i oni koji sadrže referencu (memorijsku adresu) mjesta gdje je neka vrijednost spremljena (eng. *reference type*).

Zamislimo našeg lika koji je jako bogat i ima gomilu zlata, a ima prijatelja koji je jako siromašan i nema ništa zlata. Naš prijatelj može razmišljati na dva načina, prvi je potruditi se i zaraditi isto toliko zlata tako da budemo jednako bogati, a drugi je da on ukrade naše zlato te se time obogati ali i izgubi prijatelja.

Sl. 7.1 Škrinje kao analogija za vrijednosne tipove varijabli

U prvom slučaju bi naše zlato bilo predstavljeno kao vrijednosni tip što omogućuje našem prijatelju „kopiranje“ iznosa zaradom ili posudbom, a spremnici tog zlata ili varijable možemo zamisliti kao dvije različiti škrinje pune zlata (Sl. 7.1).

Sl. 7.2 Mapa s oznakom pozicije kao analogija za referentni tip podatka

U drugom slučaju bi naše zlato bilo predstavljeno kao referentni tip jer bi naš prijatelj znao gdje se nalazi naše zlato kao (u programu je to adresa memorijske lokacije) te bi ga mogao s lakoćom ukrasti, npr. pomoću mape sa slike (Sl. 7.2) koja ima oznaku (u programu memorijska lokacija).

Primijetimo kako u prvome slučaju prijatelj nema pristup našem zlatu nego samo svom i bez obzira što smo postali jednako bogati, daljnjim trošenjem zlata ne mijenja se naša svota, ali ako nam ukrade zlato i potroši, nama se itekako mijenja svota.

Neki vrijednosni tipovi su cijeli brojevi (`int`), decimalni brojevi (`float`, `double`), znakovi (`char`), skup znakova (`string`), istina ili laž (`bool`), skup vrijednosnih tipova (`struct`)…

Referentni tip podatka je objekt koji je zapravo u centru OOP-a po kome je, kako vidimo, OOP i dobio ime.

Ako se na trenutak vratimo na prethodnu sliku i odlomak koji ju opisuje imamo sljedeće: ako u programu želimo kopirati podatak tipa broj, onda ga zapravo i kopiramo, ali nad objektom ne kopiramo vrijednost nego memorijsku adresu, a to znači da i dalje radimo s istim objektom te iste klase samo preko drugog naziva objekta.

Način na koji deklariramo bilo koji tip podatka unutar programa je skoro pa identičan, s malom razlikom kod tipa objekt. Prvo je potrebno definirati tip podatka kojeg želimo

koristiti, zatim naziv varijable/objekta, te na kraju upisati „;“ kako bi završili s tim dijelom koda.

- `tip naziv;` npr. `int mojezlato;` > ovime smo deklarirali varijablu tipa cijeli broj s nazivom broj

Još jedan način na koji možemo deklarirati neku varijablu je sljedeći: prepišemo prethodno te nadodamo operator „=“ te upišemo vrijednost koju želimo da naša varijabla sadrži. To se naziva inicijalizacija varijable. Za prethodni slučaj vrijednost pridodajemo kasnije u kodu upisivanjem: naziv = vrijednost;

Primjer inicijalizacije:

- `tip naziv = vrijednost;` npr. `int mojezlato = 5;` > ovime smo odmah i inicijalizirali varijablu

Slično vrijedi i za:

```
float  MaksimalnaTezinaOklopa= 2.4f;
double MaksimalnaTezinaOruzja= 2.444;
char Spol = 'M';
string Naziv = "Aedal";
bool popustNaOruzje = true;
struct Torba
{
      int kolicinaIstogPredmeta;
      Predmet predmet;
}
```

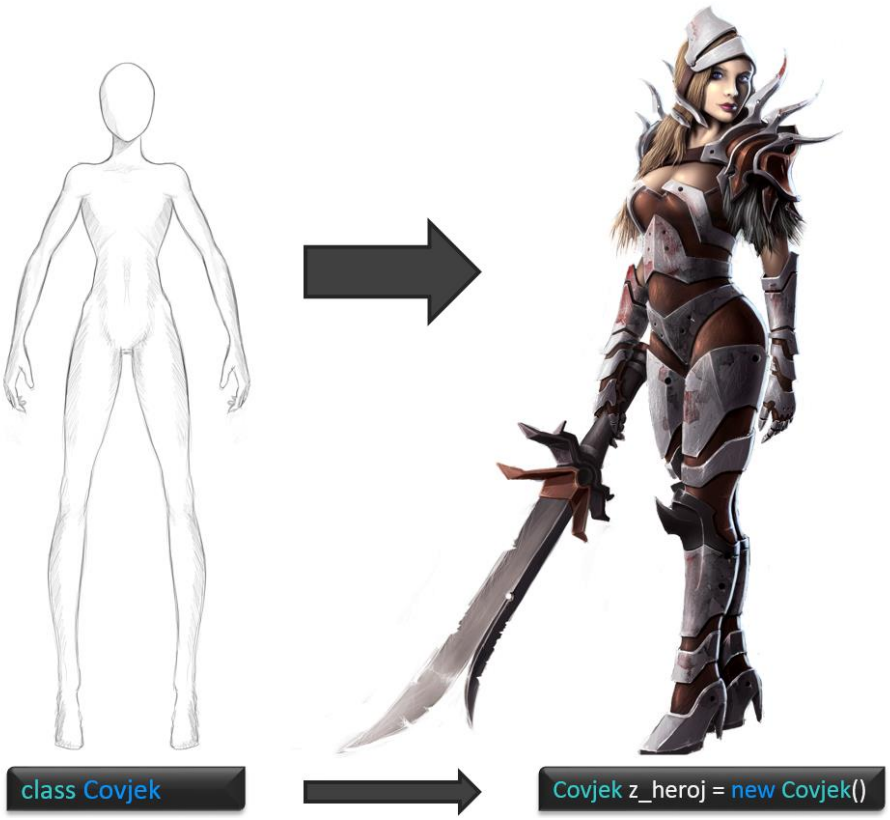
Kod 7.1 Primjer različitih tipova podataka

Za objekte je sličan postupak, a to je obrađeno u poglavlju (7.3).

7.2.1. Klasa i objekt

Vratimo se na trenutak na priču o svijetu Azeroth i herojima koji ga brane. Kao najbolji primjer možemo uzeti čovjeka koji je najviše vezan uz taj svoj dom, a i čovjek nam je već poznat iz stvarnoga svijeta. Sada se možemo zapitati pitanje pa kako ćemo toga čovjeka opisati u programu/igri? Odgovor je jednostavan, uz pomoć objekta i klase. Objekt i klasa su povezani, ali ne i isti koncepti. Klasu možemo zamisliti kao skicu koju crtamo, a ta skica ne definira točan izgled konačnog crteža nego sadrži samo bitne elemente za

prepoznavanje nečega dok objekt možemo zamisliti kao konačan crtež na kojem se jasno vide svi elementi nacrtanoga. Npr. ako pogledamo sliku (Sl. 7.3) možemo primijetiti kako skica čovjeka govori dovoljno da sa sigurnošću možemo reći kako se radi o čovjeku dok tek gotov crtež točno govori o kojem se čovjeku radi i kako on točno izgleda.



Sl. 7.3 Klasa i objekt

Za primjer možemo uzeti ovog ženskog lika sa slike. Prvo se pitamo kako bi opisali čovjeka, znači sve ljude, što im je zajedničko? Odgovor je jasan i mogli bi smo dugo nabrajati sve te opise, ali neki od njih su ime, govor, spol i sl. Primijetimo kako nismo rekli kako se tko zove, kojeg je spola i na kojem jeziku govori. Time možemo reći da smo definirali klasu čovjeka koja opisuje sve ljude. Što je u toj priči onda objekt? Objekt je onda konkretna osoba koja ima svoje ime, koja je određenog spola i priča određenim jezikom.

Ako pogledamo široku sliku tj. cijeli svijet gdje ovi konkretni heroji žive vidimo ne samo njih nego i druga živa i neživa bića. Cijeli taj svijet je opisan uz pomoć klasa, a kada se igra pokrene i kada se na zaslonu našeg ekrana prikaže taj svijet, tada se prema tim klasama stvaraju objekti koji imaju svoju ulogu u svijetu. Na slici poviše gotovog crteža imamo primjer koda kako se „stvori“ novi objekt, a to nazivamo instanciranjem pa je

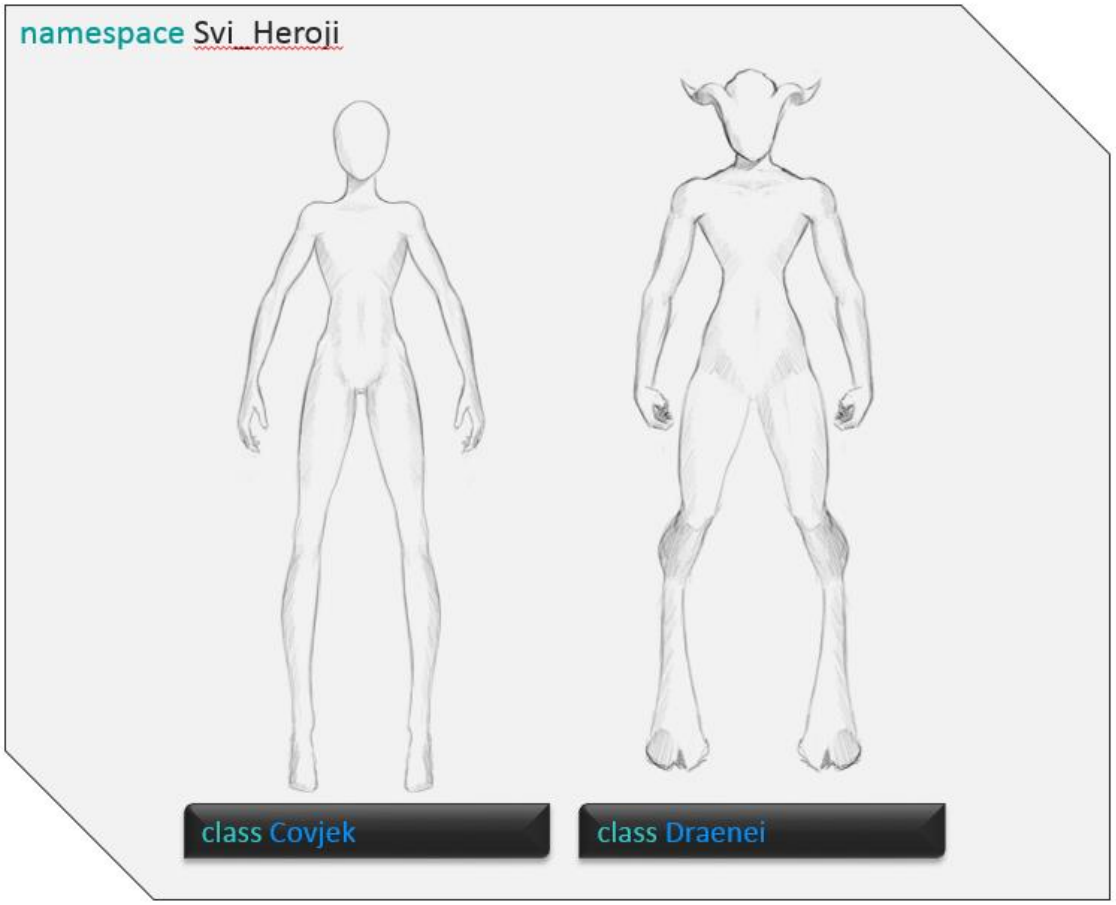
objekt instanca neke klase. Već možemo primijetiti razlike u odnosu na varijable, ali o tome više u poglavlju (7.3).

Sl. 7.4 Dva objekta iste klase

Bitno je za napomenuti kako dva čovjeka različita imena, spola ili jezika predstavljaju dva različita objekta, ali ne i objekte različitih klasa. Različite vrijednosti tih elemenata ne znače i različit element, npr. kada bi jedan od likova imao krila i mogao letjeti, a drugi ne to znači kako onaj prvi više nije pripadnik klase čovjek jer mu možemo pridodati dodatne atribute koje običan čovjek ne posjeduje. Ako npr. muški lik sa slike (Sl. 7.4) može brže trčati do ženskog lika to ne znači da ima dodatne atribute nego samo različitu vrijednost istih, on je samo brži, a obadvoje mogu trčati.

7.2.2. Imenski prostor

Imenski prostor (eng. *namespace*) također je pojam koji se spominje unutar OOP-a. Njega možemo zamisliti kao spremnik za sljedeće: klase (class), sučelja (interface), struct, druge imenske prostore i dr.. Paralelu možemo povući s direktorijima na računalu koje sadrži druge direktorije i datoteke. Nas trenutno zanimaju samo klase i imenski prostori.



Sl. 7.5 Primjer imenskog prostora

Na slici (Sl. 7.5) vidimo kako sve naše različite skice (klase) stoje pohranjene u jednom spremniku. Broj spremnika unutar jednog projekta nije ograničen, a ukoliko imamo neke klase unutar drugog „spremnika“ onda se trebamo pozvati na taj spremnik. Pozivanje vršimo na sljedeći način:

`using Naziv_spremnika;` (potrebna je ključna riječ `using` kako bi se koristili spremnikom)

Svrha korištenja spremnika je bolja organizacija projekta, a svrha upisivanja prethodnog dijela koda na početku skripte je mogućnost izravnog korištenja klasa unutar drugog

spremnika. Svaka klasa mora biti unutar nekog spremnika, a one klase koje su u različitim skriptama (dokumentima) s istim, naznačenim, spremnikom se zapravo spremaju u isti spremnik.

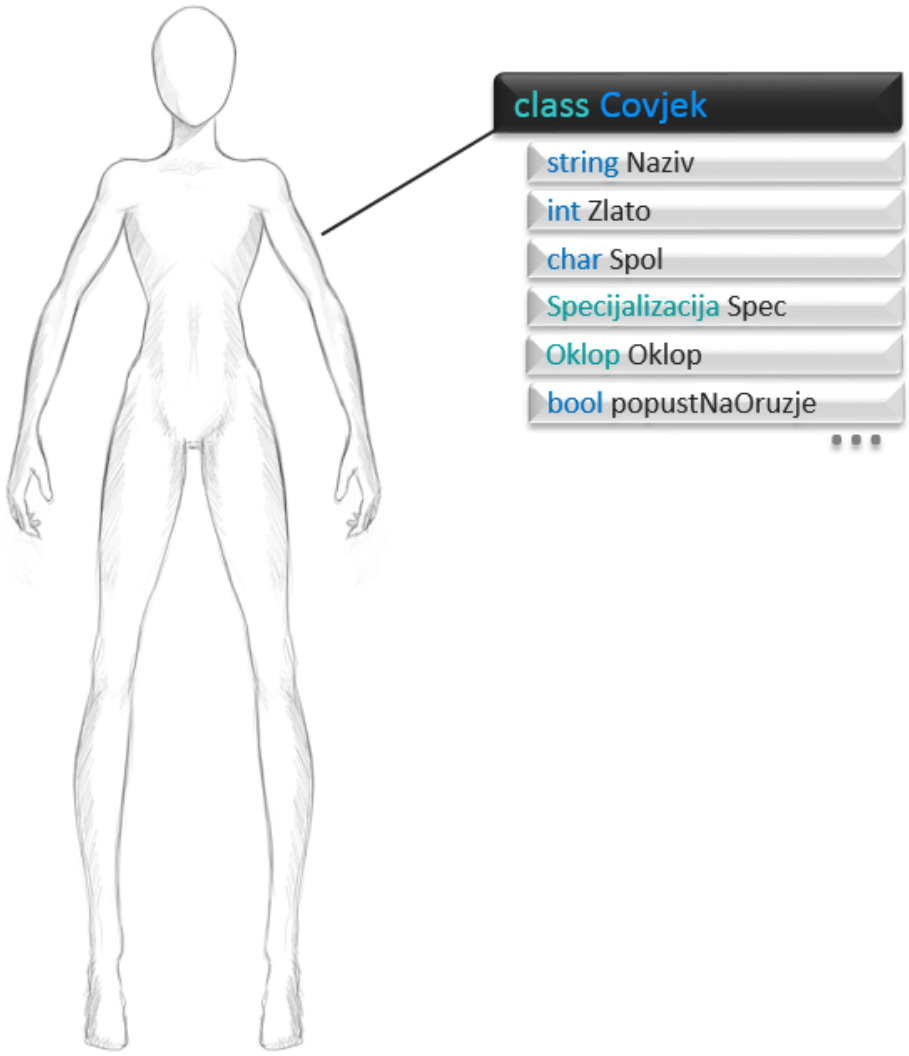
7.3. Polja, metode i konstruktor

Kroz prošlo poglavlje smo definirali što je to objekt, a što klasa te smo opisivali heroje tj. ljude uz pomoć nekih zajedničkih osobina i sl. Kada želimo sve te osobine na neki način klasificirati onda uvodimo pojam polja i metode.

7.3.1. Polje

Klasa se zapravo sastoji od podataka koji su predstavljeni kao polja, a mogu biti brojevi, znakovi, skupovi znakova, druge klase ili bilo koji drugi tip podatka. Te podatke jednim imenom nazivamo atributi klase, a to su neka od onih obilježja naših heroja (ime, spol…).

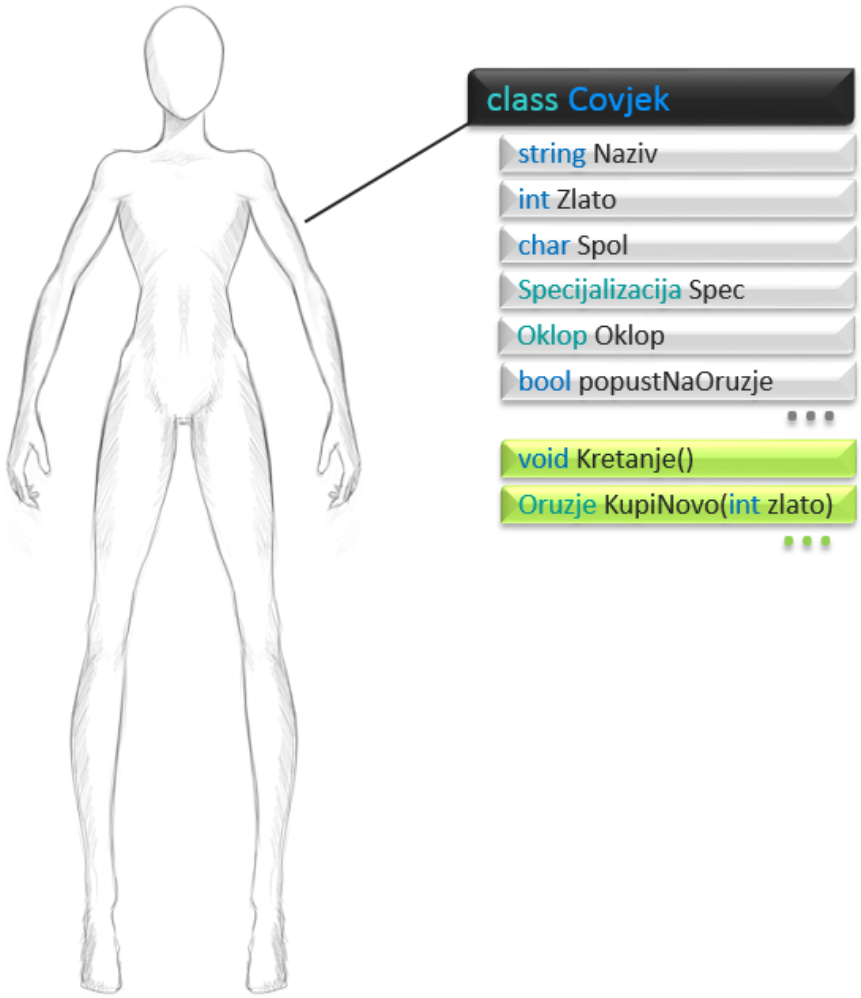
Poljima možemo pridodavati vrijednosti (inicijalizirati varijable) pri samome pisanju klase tako kada napravimo objekt, on će sadržati te vrijednosti. Ukoliko to napravimo kao na slici (Sl. 7.6) (samo deklarirati) onda će ta polja imati vrijednost tek kada ih mi sami odlučimo postaviti. Vrijednosti atributa također mogu biti promjenjive u vremenu.



Sl. 7.6 Primjer klase s poljima

7.3.2. Metoda

Također, klasa se sastoji i od metoda (Sl. 7.7), koje neki nazivaju i funkcijama prema nekim drugim programskim jezicima i paradigmama. Metode su ništa drugo nego one sposobnosti koje heroj/čovjek posjeduje (način na koji govori, hodanje, skakanje, rukovanje mačem…). Ako se prisjetimo onog ponašanja koje programer određuje programskim kodom, za točno to nam služe metode, kako bi opisali ponašanje objekta i način na koji stupa u interakciju s okolinom.



Sl. 7.7 Primjer klase s poljima i metodama

Metode se sastoje od 3 dijela, prvi je tip metode, drugo je naziv metode i treći su argumenti metode. Broj argumenata nije ograničen, može imati 0 ili više.

```
tip Naziv (argumenti tj. ulazni parametri) { ostali programski kod };
```

Prvi i osnovni tip je `void`. To je tip metode koji ne vraća nikakvu vrijednost nego se samo izvršava kod upisan unutar metode. Takvu metodu možemo pozvati samo upisivanjem imena metode i argumenata ako ih uopće metoda prima.

Također, za tip možemo postaviti bilo koji tip podatka (npr. `int`, `string`, `nazivklase`), a to bi značilo kako metoda po završetku izvođenja mora vratiti vrijednost toga istog tipa podatka. To se ostvaruje ključnom riječi `return`. Ime postavljamo proizvoljno, ali bez razmaka. Unutar zagrada upisujemo argumente, a to je zapravo popis varijabla koje želimo da metoda primi. I konačno unutar zagrada“{ }“ pišemo kod koji se treba izvršiti. Primjer koda je ispod:

```
Oruzje Kupi (int zlato ){
      zlato -= oruzje.cijena;
      return oruzje;
}
void Pogledaj (){
      PrikaziOruzje();
}
```

Kod 7.2 Primjer dvije metode

Ako pogledamo sliku (Sl. 7.8), lijevi tip metode, koji vraća neku vrijednost, ne možemo pozvati kao `void` nego trebamo tu „vraćenu“ vrijednost negdje spremiti ili poslati u drugu metodu koja također prima parametar tipa `Oruzje`. Npr. `Oruzje mojeOruzje = Kupi(mojeZlato);` ili `DodajNovoOruzje(Kupi(mojeZlato));` Za void metodu dovoljno je upisati: `Pogledaj ();`.

Sl. 7.8 Metoda s nazivom klase i void metoda

Npr. na slici (Sl. 7.8) vidimo 4 različita oružja. Kada bi metode opisivali preko kupovine i pregleda oružja pregled bi bila metoda tipa `void` koja ne prima nikakve parametre niti vraća parametre kao rezultat. Metoda za kupovinu oružja treba biti tipa `Oruzje` jer to dobijemo kupovinom. Ista metoda treba imati ulazni parametar, tj. novac kojim kupujemo oružje. Unutar te metode bi se izvršavao kod koji bi smanjio količinu našeg zlata te nam

kao vrijednost vratio odabrano oružje. Vrijednosti koje vraćamo kao rezultat metode imaju ključnu riječ `return`.

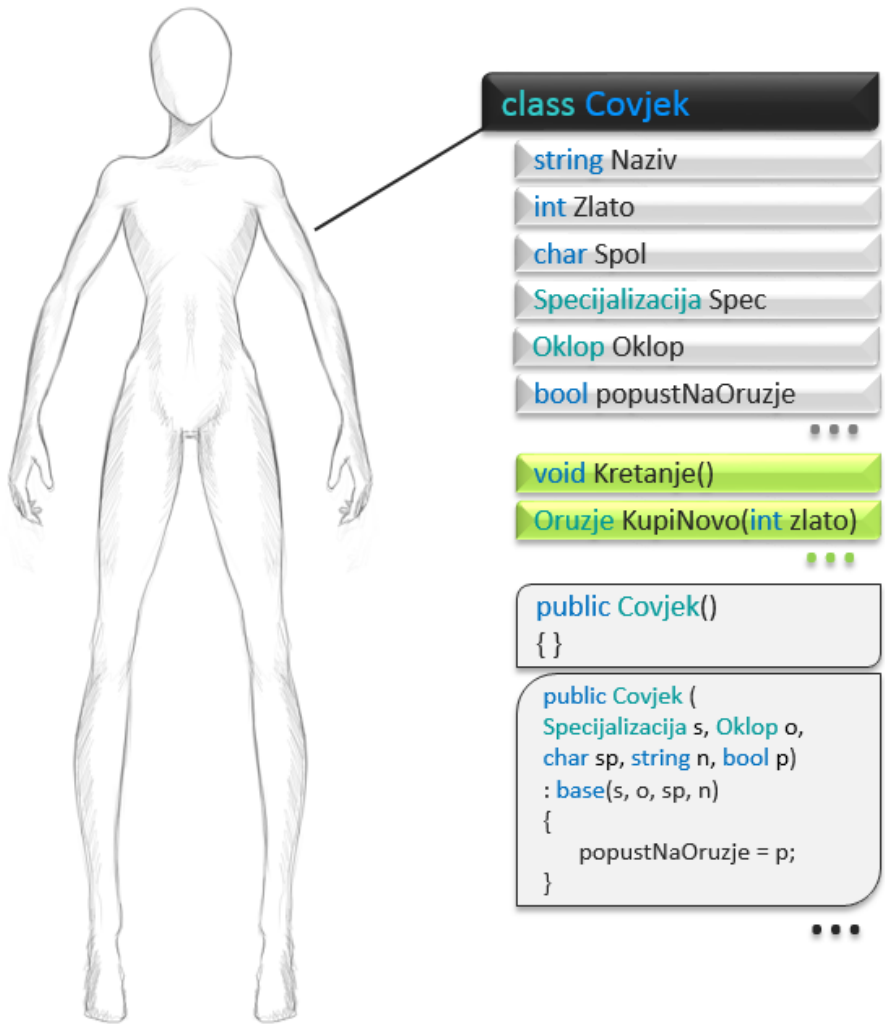
Često se ne razumije način funkcioniranja metoda posebno kada se radi o metodama koje vraćaju neku vrijednost. Tip metode može biti bilo kakav dok god odgovara tipovima varijabli ili objekata koji su nam na raspolaganju (`int`, `string`, `covjek` ili bilo koja druga klasa). Kada metoda vraća neku vrijednost, s tom vrijednošću moramo negdje upraviti bilo to pospremiti je negdje, poslati ju u drugu metodu ili konstruktor kao argument (argumenti su opet varijable). Imamo potpunu slobodu dok se god tipovi podataka poklapaju (rezultat int metode može se spremiti u bilo koju varijablu tipa `int`, ili bilo koji argument tipa `int`). Drugim riječima pozivanje metode koja vraća vrijednost treba biti tretirana kao najobičniji podatak toga istoga tipa.

7.3.3. Konstruktor

Treći element svake klase je konstruktor koji je zapravo poseban oblik metode. Dosta je sličan običnoj metodi s jednom razlikom, uvijek treba sadržati naziv klase kao na slici 4.3, dok metode sadrže svoj vlastiti naziv. Konstruktor je taj ključni element koji je potreban za instanciranje (stvaranje objekta), a to je onaj dio koda koji deklariranje varijabli razlikuje od instanciranja objekta što je spomenuto u poglavlju 3.

Broj konstruktora je neograničen, a u klasi, iako ne vidimo, uvijek postoji jedan konstruktor koji je prazan. Taj konstruktor nazivamo osnovnim konstruktorom (eng. default constructor).

Za izradu novog konstruktora nikako ne smijemo mijenjati naziv. Kako smo rekli da je konstruktor jako sličan metodi, to možemo vidjeti i iz slike (Sl. 7.9). Isto imamo dio koda unutar zagrada „{ }“ te argumente unutar zagrada „( )“. Pridjeljivanje vrijednosti unutar „{ }“ možemo vršiti na bilo koji način što znači kako nismo ograničeni. Za razliku od metode, konstruktoru ne pridodajemo tip kao broj ili znak nego samo naziv klase. Također svaki konstruktor, kao na slici, treba sadržati riječ public što nam je potrebno kako bi mogli pristupiti konstruktoru. To je zapravo pravo pristupa, ali o tome više u poglavlju (7.6).



Sl. 7.9 Primjer klase s poljima, metodama i konstruktorima

Poljima i metodama pristupamo na sljedeći način:

- `Naziv_objekta.nazivpolja;` > pristup polju
- `Naziv_objekta.nazivMetode();` > pristup metodi

Instanciranje se vrši na sljedeći način:

- `NazivKlase Naziv_objekta = new NazivKlase();` > ključna riječ `new` + konstruktor

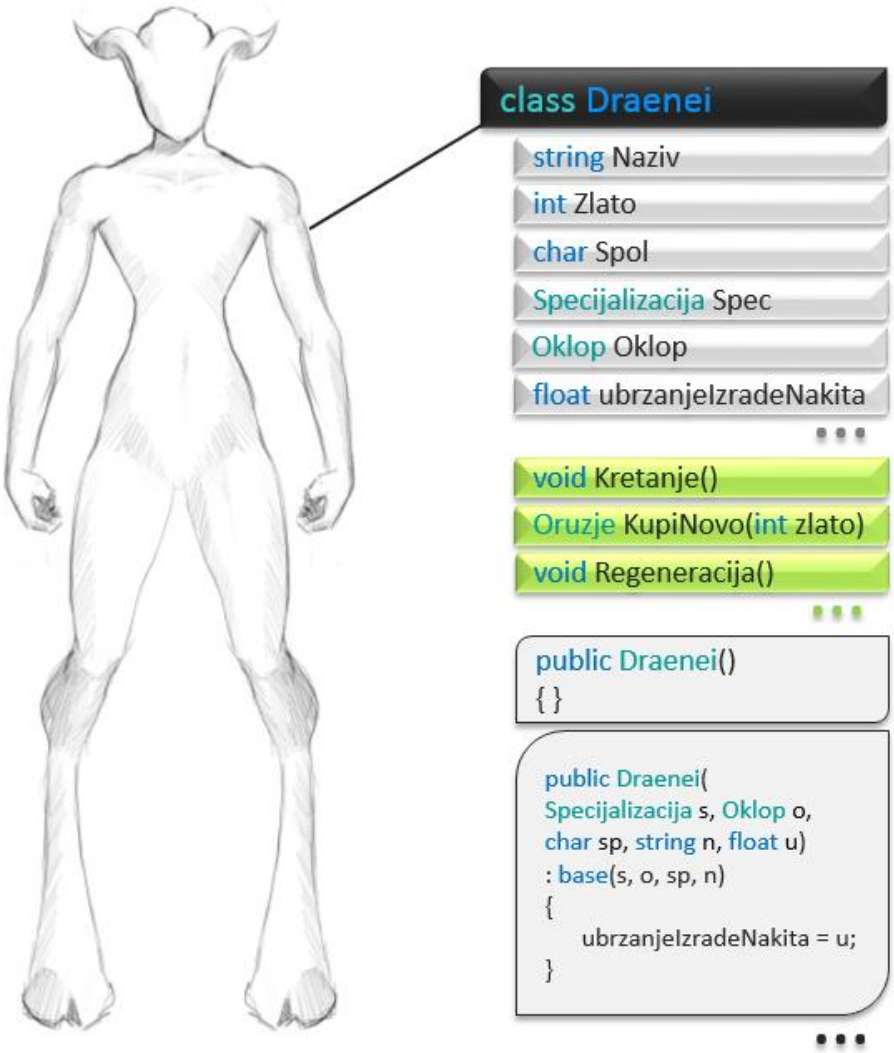
Npr:

```
Covjek z_lik = new Covjek();
Covjek m_lik = new Covjek (objekt_spec, objekt_oklop, "M", "Aedal", true)
```

7.4. Nasljeđivanje

Odmah nakon klase i objekta, nasljeđivanje je jedan od najbitnijih koncepata OOP-a. Kao što sama riječ govori nasljeđivanje je kupljenje osobina roditelja.

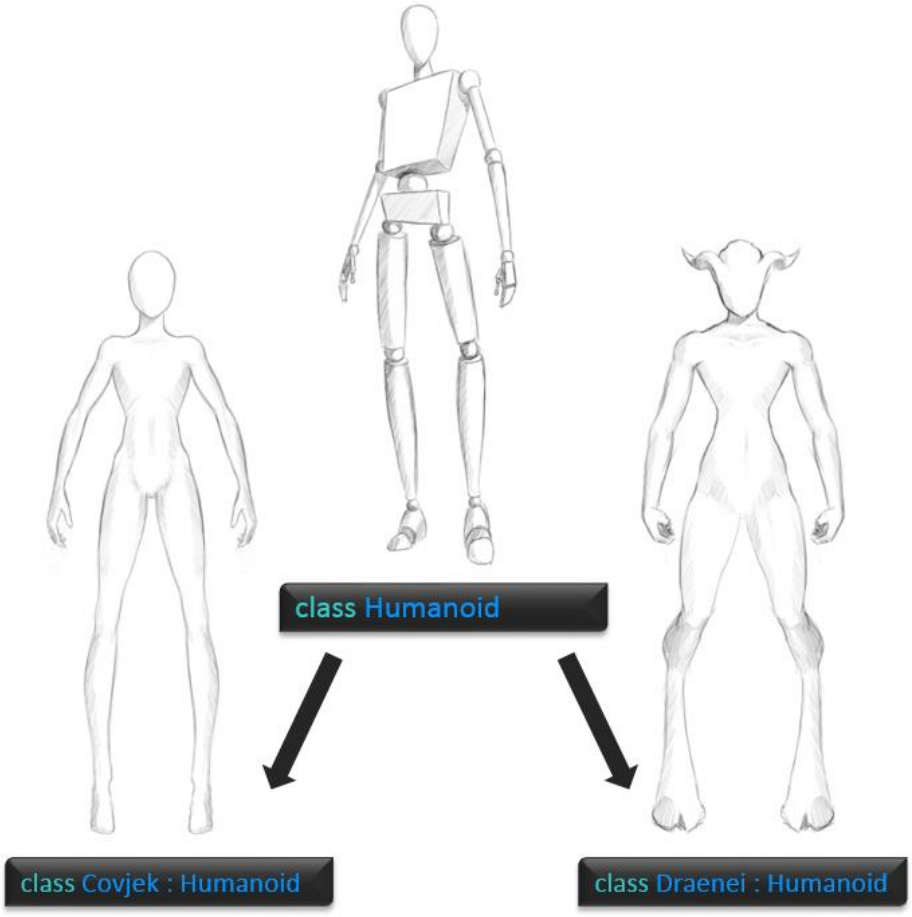
Prvo postavimo jedan problem pa ga pokušajmo riješiti. Već smo uveli klasu čovjeka kojeg smo opisali raznim svojstvima i sposobnostima, a sada uvedimo klasu draenei-a (Sl. 7.10).



Sl. 7.10 Nova klasa

Već iz priloženog vidimo kako draenei i čovjek nisu isti, ali ipak dijele neka zajednička svojstva. Kada bi pošli opisivati draeneia primijetili bi kako se dosta svojstava poklapa s čovjekovim. Problem je što sada imamo dvije slične klase što znači kako imamo viška programskog koda koji se ponavlja. Taj problem lako je riješiti koristeći se nasljeđivanjem, a od tu ta potreba za nasljeđivanjem. Prema tome, definirati ćemo novu klasu koja će sadržati zajednička obilježja čovjeka i draenia te ju nazvati humanoid. Zatim ćemo

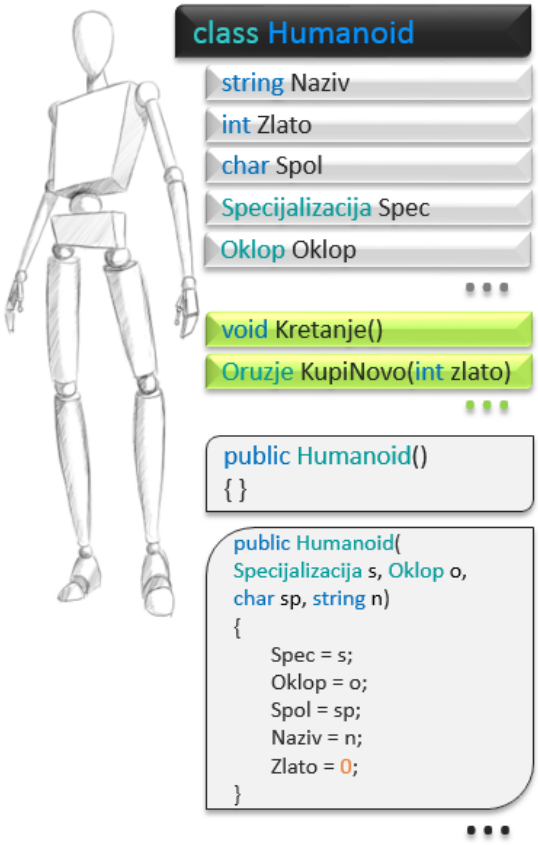
izmijeniti klasu čovjeka te slično čovjeku napraviti klasu draenei. U klasu čovjeka i draeneia dodajemo samo ona obilježja koja su jedinstvena za te rase dok su zajednička svojstva zapisana u klasi humanoid. Kada smo to napravili potrebno je samo kod naziva klase upisati „: Humanoid“ kako bi naznačili nasljeđivanje. Humanoid je sada osnovna klasa koju nazivamo „roditelj“ klase čovjek i draenei, a klase čovjek i draenei su izvedene klase ili „djeca“ humanoida (Sl. 7.11). Kada bi i klasa čovjeka imala djece, onda bi ta djeca za klasu humanoid bila potomak, ali ne i djete.



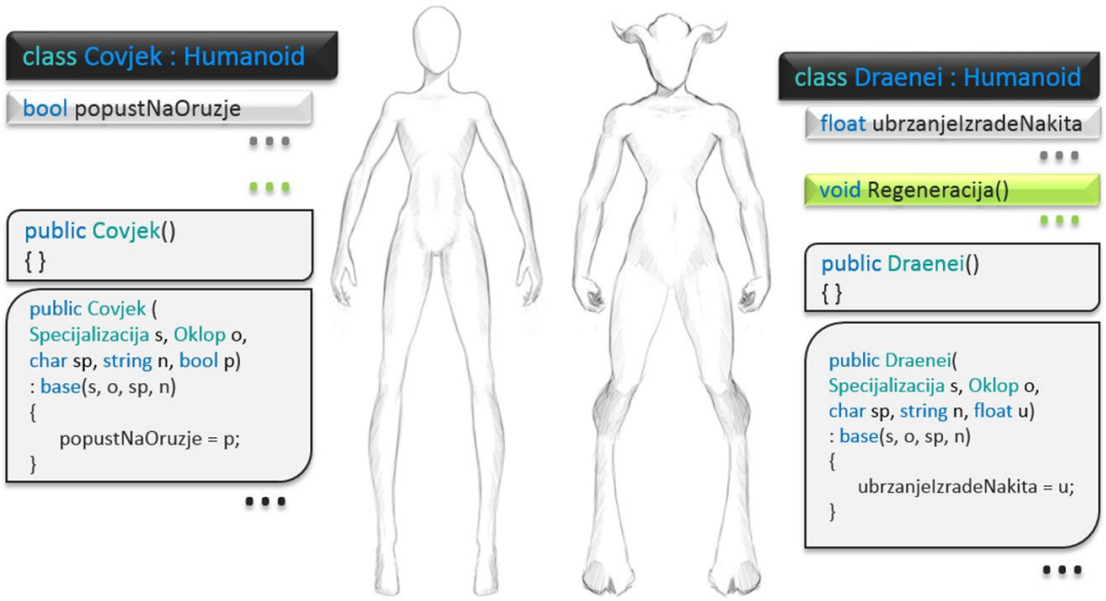
Sl. 7.11 Primjer nasljeđivanja s rasama iz igre WoW

U OOP-u, sve klase, osim jedne, su naslijeđene klase. Svaka klasa potječe od najviše u hijerarhiji, a ona se naziva object (malo zbunjujuće, ali se ne dajmo zavarati, taj object je samo naziv klase). Unutar okruženja Unity, pri stvaranju nove skripte svaka klasa se također postavlja kao dijete od klase Monobehaviour kao bi „pokupila“ sva obilježja potrebna za upravljanje elementima igre koji su već isprogramirani od tima programera koji su izgradili Unity.

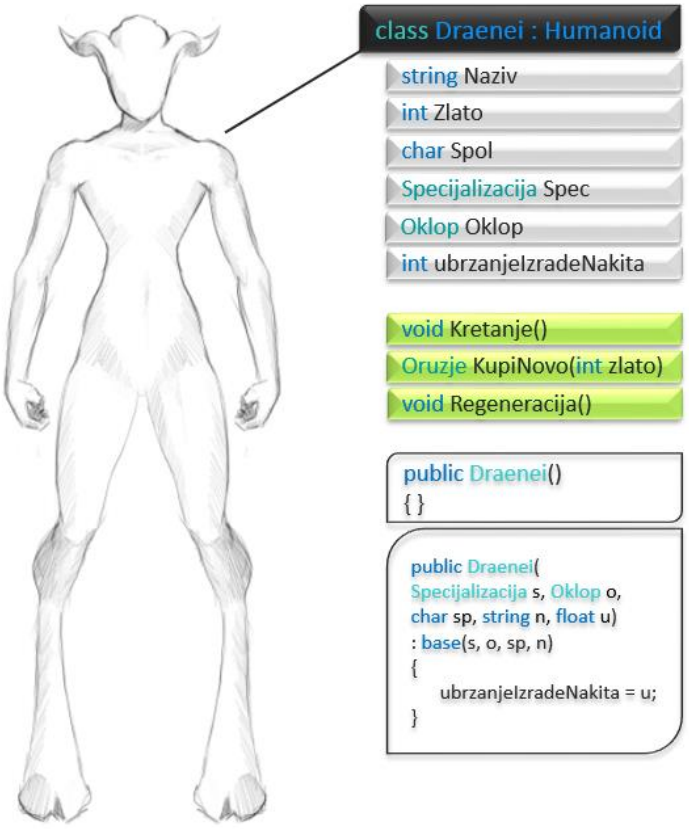
Na slici (Sl. 7.12) vidimo kako sada izgleda izvorna klasa iz koje radimo sve izvedene klase rasa, a na slici (Sl. 7.13) vidimo primjer dviju različitih izvedenih klasa (rasa). Iako na slici 6.4 izgleda kako čovjek ima samo jedan atribut i konstruktor te draenei samo jedan atribut, jednu metodu i jedan konstruktor, ali to nije točno. Oni imaju sve što je zapisano unutar njihove klase i sve što je zapisano unutar klase humanoid (naslijedili su elemente roditelja). Da klasa čovjek ima svoje dijete, ono bi posjedovalo sve što imaju njegovi pretci, znači sve što posjeduje čovjek i humanoid i naravno onaj object i monobehaviour što je slučaj za OOP općenito i Unity.



Sl. 7.12 Izvorna klasa



Sl. 7.13 Izvedene klase



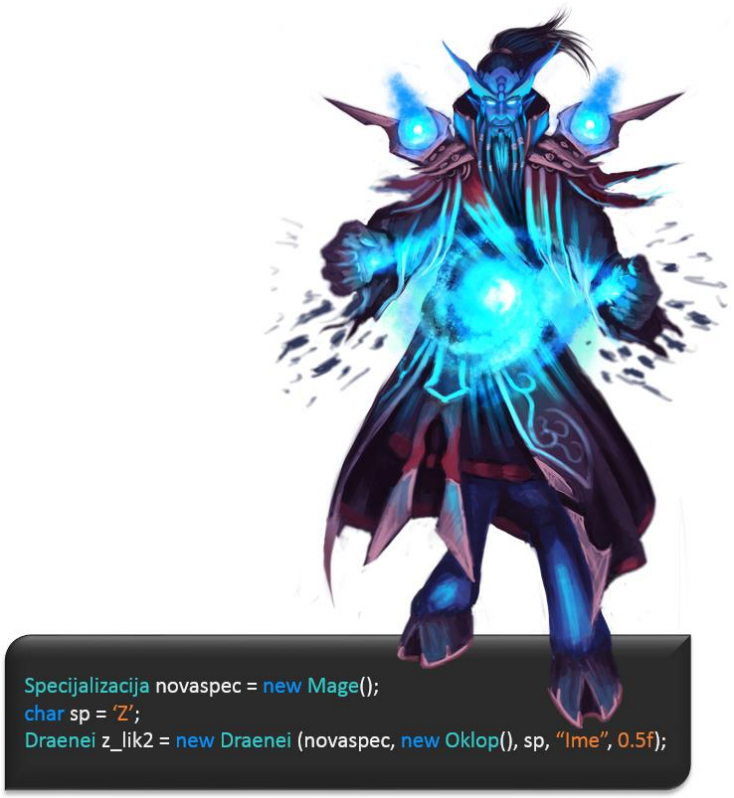
Sl. 7.14 Stvarni sadržaj izvedene klase

Na slici (Sl. 7.14), u odnosu na sliku (Sl. 7.10), možemo primijetiti jedino razliku kod konstruktora izvedene klase. Argumente primamo kao i kod konstruktora osnovne klase, ali oni dio argumenata koji odgovara atributima osnovne klase potrebno je naznačiti da su

za tu osnovnu klasu. Za to koristimo :base() odmah nakon navedenih argumenata gdje upišemo, nazivima, one argumente koji pripadaju osnovnoj klasi. Za stvaranje objekta izvedene klase možemo koristiti samo konstruktore te klase dok se konstruktori izvorne klase pozivaju korištenjem ključne riječi base.

Novu instancu klase možemo „pospremiti“ pod trenutnim imenom klase ili pod imenom izvorne klase kao na slici (Sl. 7.15). Humanoid je izvorna klasa, a Covjek je izvedena klasa kao i što je odnos Humanoid – Draenei. Čovjek jest humanoid, ali humanoid nije čovjek stoga vrijedi:

- `Humanoid z_lik = new Humanoid();` > Valjano
- `Humanoid z_lik = new Covjek();` > Valjano
- `Covjek z_lik = new Covjek();` > Valjano
- `Covjek z_lik = new Humanoid();` > Nevaljano



Sl. 7.15 Objekt

Na slici (Sl. 7.15) možemo primijetiti nekolicinu stvari koje su spomenute kao napomena kod lekcije 4, točnije kod metoda i konstruktora. Primijetimo sve načine na koje konstruktoru šaljemo parametre, prvo kao objekt, drugo kao objekt izravno preko konstruktora, treće kao varijablu, a četvrto i peto kao riječ u navodnicima i broj.

`Specijalziacija novaspec = new Mage()` – ovaj dio koda će napraviti novi objekt kojeg ćemo kasnije poslati kao parametar za Specijalziacija s. Drugi parametar šaljemo također kao objekt. U slučaju novaspec imamo samo liniju koda više, a isti rezultat kao i kod oklopa. Način kojim se koristimo ovisi o programeru i njegovom stilu programiranja. Isto vrijedi i za naziv i spol. Način slanja parametara je nebitan dok god tipovi podataka odgovaraju na obadvije strane.

Važno je još naglasiti kako postoje dvije vrste veza koje se ne smiju miješati, a to su veza posjedovanja (eng. *has relationship*) i veza nasljeđivanja (eng. *is relationship*). Veza nasljeđivanja je prethodno objašnjeno, svaki čovjek je humanoid, a isto tako i svaki ork je humanoid. Svaki čovjek posjeduje vlastitu specijalizaciju, ali to ga ne čini specijalizacijom tj. pojedinac posjeduje sposobnosti ratnika ili lovca, ali ga to ne čini tom sposobnošću.

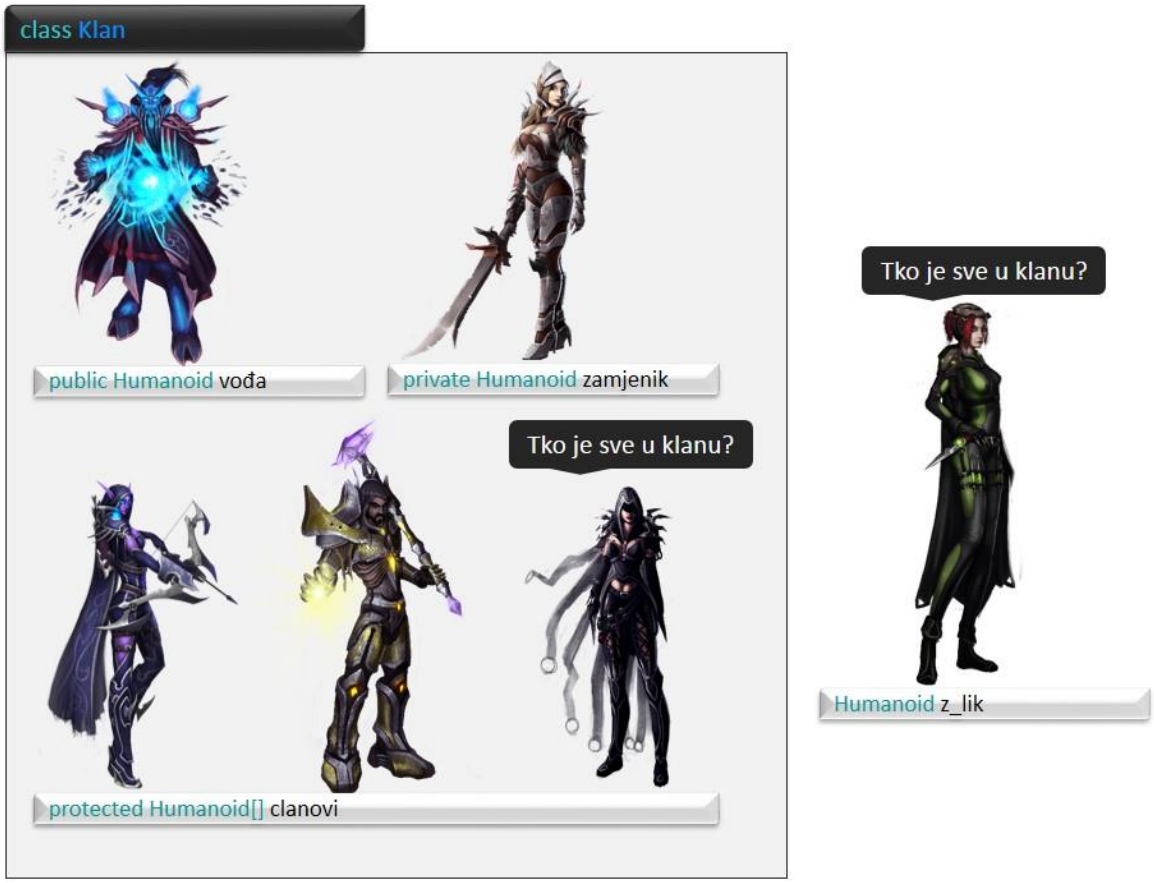
7.5. Prava pristupa i učahurivanje

Kada smo uspješno opisali našeg heroja, od imena pa sve do njegovih sposobnosti i uloge u svijetu, trebamo postaviti prava pristupa određenim obilježjima. Za to imamo par pitanja:

- Tko bi sve trebao znati naše planove?
- Tko bi sve trebao znati sve o nama?
- Tko sam ja i da li želim da drugi utječu na to tko sam ja?
- Kako da se zaštitim od negativnog utjecaja drugih?

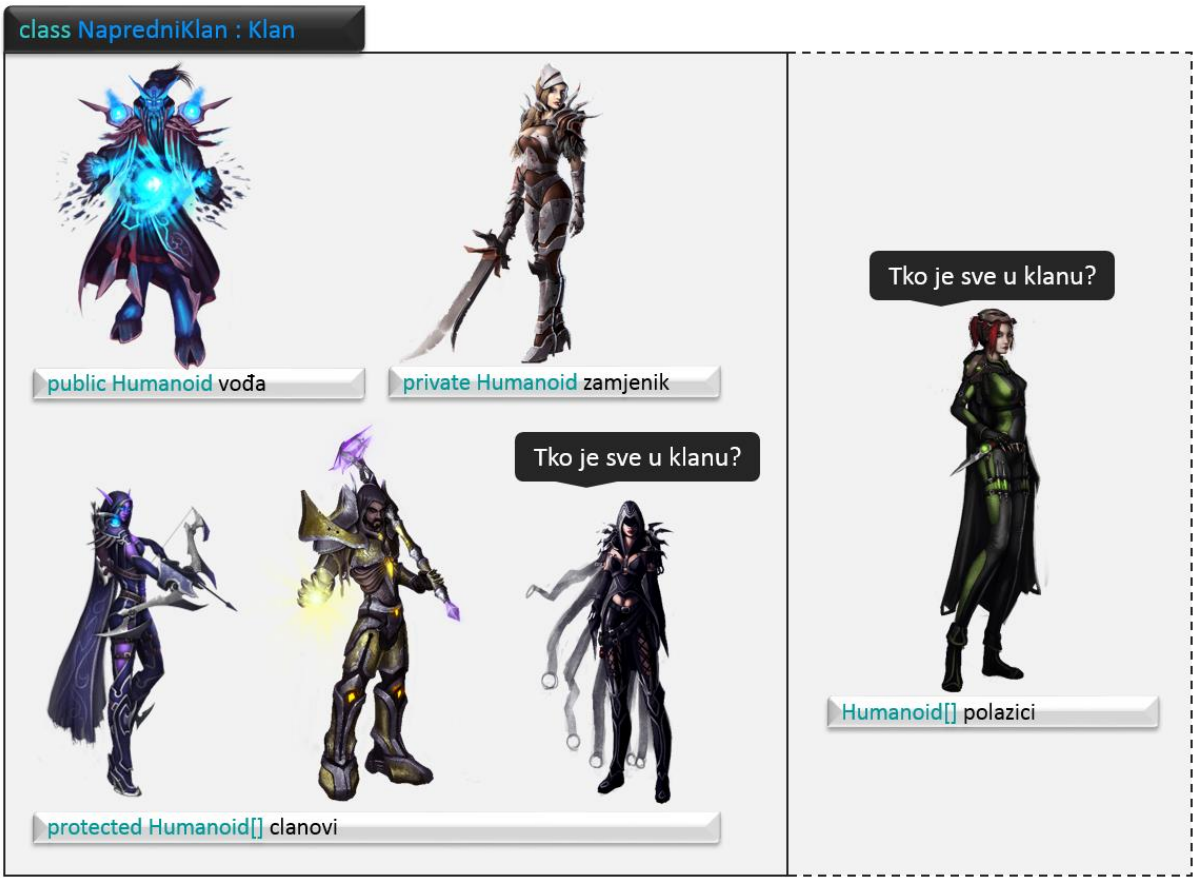
Ujedinili smo se i osnovali klan, bolje ćemo funkcionirati kao skupina nego kao pojedinci. Kada imamo svoj klan trebamo ga zaštiti od drugih i držati neke informacije tajnima. Za to nam, u programiranju, služe prava pristupa. Navesti ćemo samo tri osnovna, a to su: `public`, `private`, `protected`.

Na primjeru slike (Sl. 7.16) s klasom bez nasljeđivanja vidimo kako se dva heroja pitaju tko je sve u klanu. Heroj koji je član klana dobro poznaje sve članove klana dok onaj tko nije član klana može poznavati samo vođu jer je vođa javan (`public`), a informacije o ostalima su privatne (`private`) ili zaštićene (`protected`).



Sl. 7.16 Primjer 1 za prava pristupa

Također, imamo još jedan slučaj po kojem možemo razlikovati `private` i `protected` članove. Ako pogledamo sliku (Sl. 7.17) vidimo kako smo svoj klan malo proširili kako bi omogućili suradnju onih koji nisu članovi klana da to možda jednoga dana i postanu. Sada oni heroji koji nisu članovi mogu postati polaznici. Ti polaznici su na neki način dio klana, ali one verzije naprednog klana i kada se sada zapitaju tko je u klanu dobiti će nešto drugačiji odgovor. Sada će znati javne i zaštićene članove klana, a privatne i dalje neće poznavati. Tek kada polaznik dobije ulogu člana, zamjenika ili vođe, znati će sve one koji su u klanu.



Sl. 7.17 Primjer 2 za prava pristupa

U programu to znači kako `public` atributima i metodama klase možemo pristupati od bilo gdje, dok privatnim možemo samo unutar klase, a `protected` unutar klase i svih klasa koje su potomci te iste klase.

Prava pristupa mogu biti jako korisna za zaštitu podataka klase. Npr. ako se heroji zvan klana ili unutar klana pobune i žele svrgnuti vođu oni će to biti u mogućnosti napraviti, ali zamjenika će moći zamijeniti samo oni unutar klana. U programu to znači, da promjena vrijednosti privatne varijable/objekta može biti određena samo metodom koja je u istoj klasi kao i varijabla/objekt. Isto tako postoji još jedan način, a to preko svojstva.

Svojstvom možemo pristupiti privatnoj varijabli, a za to je potrebno učahurivanje polja (eng. *field encapstulation*).

```
private int mojeZlato;

public int MojeZlato{

      get {return mojeZlato;}

      set { if(value) < mojeZlato ) mojeZlato -=value;}
```

```
}
```

Kod 7.3 Primjer učahurivanja

Prema primjeru gore potrebno je napraviti `public` svojstvo istoga tipa i bilo kojeg naziva (preporuča se koristiti isti naziv s velikim početnim slovom ili svim velikim slovima). Unutar „{ }“ zagrada potrebno je upisati `get{}` i `set{}` metode, to su posebne metode koje služe za dohvaćanje zaštićenog elementa (`get`) i za postavljanje zaštićenog elementa (`set`).

Sl. 7.18 Čuvari

Učahurivanje možemo objasniti kao čuvare koji kontroliraju tko i kakav pristup ima onome tko je iza vrata (Sl. 7.18). Učahurivanje nam omogućuje da postavljamo razne uvjete unutar `get` i `set` metoda kako bi bolje zaštitili privatni element. Npr. unutar set metode možemo postaviti provjeru valjanosti ulazne vrijednosti (`value`) i ako vrijednost nije primjerena možemo ju odbaciti tj. ne učiniti ništa, a u protivnom spremiti.

7.6. Polimorfizam

Do sada smo opisali čovjeka i draeneia te smo našli njihova zajednička, a i nepripadajuća obilježja te smo spominjali nekakve specijalizacije. Specijalizacije dijelimo na ratnike, mađioničare i sl.. Kako se vidi na slici 8.1, naši heroji su sada naoružani i spremni za borbu, a isto tako su različitih specijalizacija. Lako nam je zamisliti jednu metodu koja će predstavljati udarac oružjem ili golim rukama koja je zajednička za sve specijalizacije. Problem se javlja kada za svaku specijalizaciju trebamo ponovno pisati istu metodu za udarac, ali za različite tipove oružja koje koriste različite specijalizacije (Sl. 7.19). Postavlja se zadatak pred nas kako realizirati da taj isti napad ne razdvajamo po svim klasama svih specijalizacija pod različitim imenima nego samo želimo u svakoj verziju specijalizacije prilagođenu metodu za određeno oružje. Za to ćemo se služit pojmom zvani polimorfizam.

Sl. 7.19 Različiti heroji s različitim tipovima oružja (različiti načini napada i obrane)

Polimorfizam nam omogućava da postojeće metode i svojstva izvorne klase napravimo, pod istim nazivom, i u izvedenoj klasi. Postoje dva načina izrade „istih“ metoda i svojstava, prvi je premošćivanje (`override`), a drugi način je korištenjem ključne riječi `new`.

Premošćivanje: u izvornoj klasi trebamo metodu ili svojstvo deklarirati kao virtualnu (`virtual`) metodu, a pri deklariranju iste u izvedenoj klasi koristimo ključnu riječ override.

Korištenjem ključne riječi `new` deklarira se metoda u izvedenoj klasi kao u primjeru.

Primjer:

```
class Humanoid{
      public virtual void Napad();
      public void Obrana();
}
class Covjek : Humanoid{
      public override void Napad();
      public new void Obrana();
}
```

Kod 7.4 Primjer nasljeđivanja

Instanciranje:

- `Humanoid z_lik = new Covjek();` > instanciranje 1. način
- `Covjek m_lik = new Covjek();` > instanciranje 2. način

Korištenje metoda:

- `z_lik.Napad(…);` > pozivamo metodu iz izvedene klase, klase Covjek
- `z_lik.Obrana(…);` > pozivamo metodu iz osnovne klase, klase Humanoid
- `m_lik.Napad(…);` > pozivamo metodu iz izvedene klase, klase Covjek
- `m_lik.Obrana(…);` > pozivamo metodu iz izvedene klase, klase Covjek

Razlikovanje metoda unutar izvedene klase:

- `base.Napad();` > metoda osnovne klase
- `this.Napad();` > metoda izvedene (trenutne) klase

7.7. Uvod u kontrole i događaje

Kontrole su objekti koji su dio grafičkog sučelja, a služe nam za unos i ispis podataka. Neke od kontrola koje su svima poznate su dugme (eng. *button*), tekstualni okviri za unos

(eng. *textbox*), labele za ispis (eng. *label*)… Sve kontrole su također objekti opisani svojim klasama. Metode koje su vezane za te kontrole nazivaju se događajima, npr. metoda za klik na dugme, metoda za prijelaz pokazivačem miša preko dugmeta i sl. Unutar tih metoda pišemo kod koji želimo da se odvija npr. na klik dugmeta. Za primjer ćemo uzeti dugme jer je to najčešći način interakcije igrača i igre, a to je vidljivo na slici (Sl. 7.20).

Sl. 7.20 Sučelje World Of Warcraft igre

Za razliku od običnih metoda, događaji se često mogu napraviti od strane programa u kojem pišemo kod, ali bez obzira na koji način napravili događaj, on ima već predodređen tip i ime. Događaji su uvijek tipa `void` jer po svojoj prirodi ne daju nekakav rezultat nego služe za odrađivanje određenih radnji koje programer zadaje. Kao što je već spomenuto, događaji imaju već predodređeno ime kako bi ih se moglo prepoznati, ali isto tako i razlikovati od drugih metoda bilo to da ih sami pišemo ili da ih je netko drugi napisao.

Događaji su jedan veliki dio igara jer nam oni služe za komunikaciju s igrom i herojima s kojima igramo. Traka s moćima je ništa drugo nego skup dugmadi koja imaju više različitih događaja vezano za sebe, a to su: klik, prelazak mišem preko dugmeta, napuštanje dugmeta i sl. Svaki element grafičkog sučelja posjeduje svoje događaje, ali svi elementi nemaju sve događaje, npr. klikom na energiju heroja ne dobijemo nikakav odgovor jer je to element bez događaja za klik dok prelaskom pokazivača miša preko elementa ispisuje se brojčana vrijednost energije.

Važno je za napomenuti, iako su to već predodređene metode, one su prazne i od programera se zahtjeva punjenje tj. pisanje koda.

Možemo reći kako su događaji zadnji korak pri pokretanju heroja, u početku smo izradili klasu koja je opisivala cijelu rasu, zatim smo stvorili objekt tj. heroja s imenom i jedinstvenim izgledom. Istom tom heroju smo pridodali razne sposobnosti te na samom kraju te njegove sposobnosti povezali na kontrole preko događaja što nam omogućuje sudjelovanje u tom virtualnom svijetu tj. upravljanje herojem koji živi u tome istome svijetu.

8. Dodatak 2 – Unity 3D vježbe

8.1. Razvojno okruženje

Ovo poglavlje sadrži glavne alate koji se koriste za izradu 3D igara. Jedan od alata je Unity game *engine*. *Game engine* je program koji inače dolazi s različitim paketima koji omogućuju stvaranje same igre.

Neki od tih paketa su za grafiku, zvuk, animacije, fiziku i detekciju „sudara“. Paket za grafiku naziva se *renderer*, a on omogućuje stvaranje slika (eng. *frame*) 2D i 3D objekata na ekranu. Paket za zvuk omogućuje upravljanje audio datotekama. Animacije ne spadaju pod fiziku jer su to samo zapisi kretnji koje napravimo u određenim alatima te ne ovise o fizici igre. Programi za razvoj igara nam pružaju jako jednostavan način upravljanja animacijama uz minimalno korištenje programskog koda. Program sam računa animacije prema našim postavkama. Paket za izračunavanje fizike nam pruža simulaciju gravitacije unutar igre dok paket za detekciju sudara služi za sprječavanje preklapanja 2D ili 3D objekata (eng. *clipping*). Svi ti paketi su sadržani unutar gotove biblioteke koje su uključene u sam program.

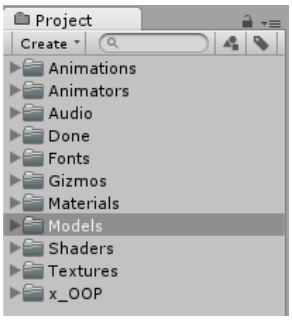
Druga dva alata su Monodevelop i Visual Studio 2012. Oba programa imaju istu svrhu, a to je pisanje programskog koda tj. stvaranje logičkog dijela igre. Unity *engine* podržava više programskih jezika kao što su C# i JavaScript. Ovaj dokument se odnosi isključivo na C#.

8.1.1. Unity

Sada ćemo se upoznati s osnovnim elementima razvojnog okruženja Unity i vidjeti kako je to povezano s objektno orijentiranim programiranjem.

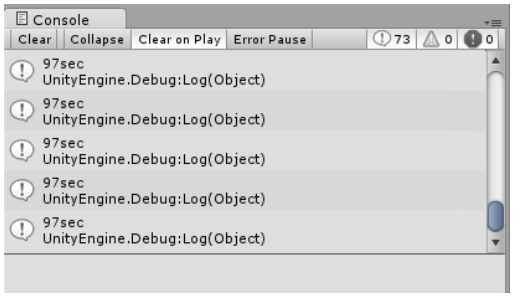
Sučelje

Okvir za organizaciju elemenata igre tj. datoteka i direktorija prikazan je na slici (Sl. 8.1). U ovom okviru možemo pronaći sve datoteke trenutnog projekta (igre) koje Unity podržava. Organizacija podataka je kao i kod operacijskog sustava (stablasta struktura).



Sl. 8.1 Okvir za datoteke i direktorije trenutno otvorenog projekta

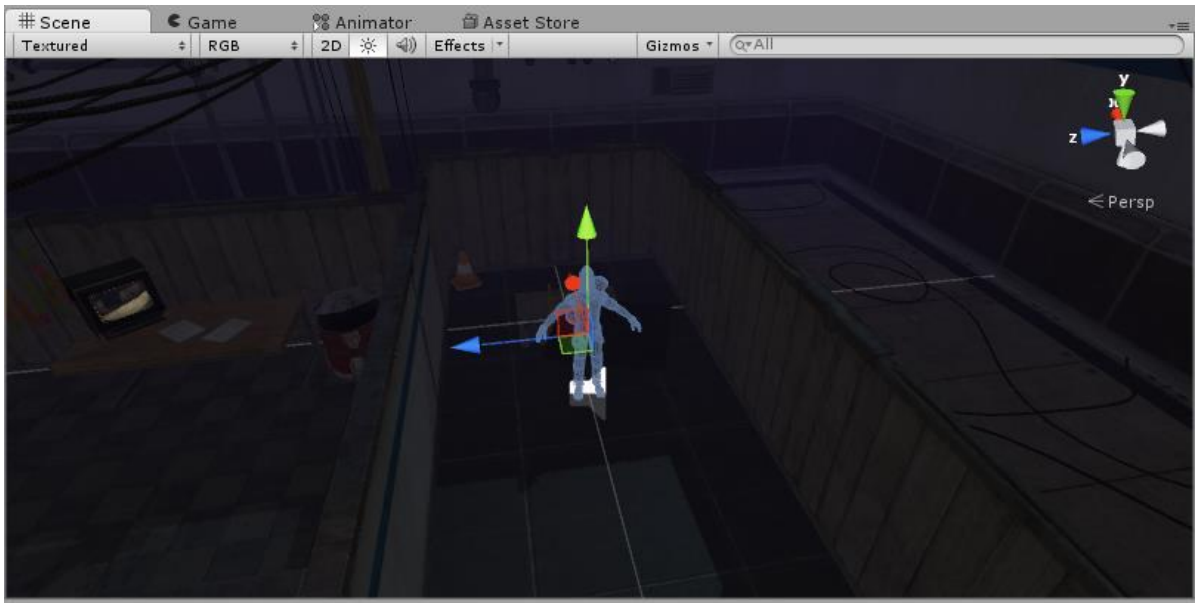
Okvir konzole služi nam za ispis poruka preko skripti igre (Sl. 8.2). Također, na konzolu se ispisuju i sva upozorenja i greške koje imamo u trenutnoj igri. Ispis na konzolu se preporuča koristiti samo za testiranje određenih funkcionalnosti igre. Poruke korisnicima potrebno je pisati na grafičko sučelje igre jer konzola nije vidljiva korisnicima tj. igračima.



Sl. 8.2 Okvir konzole

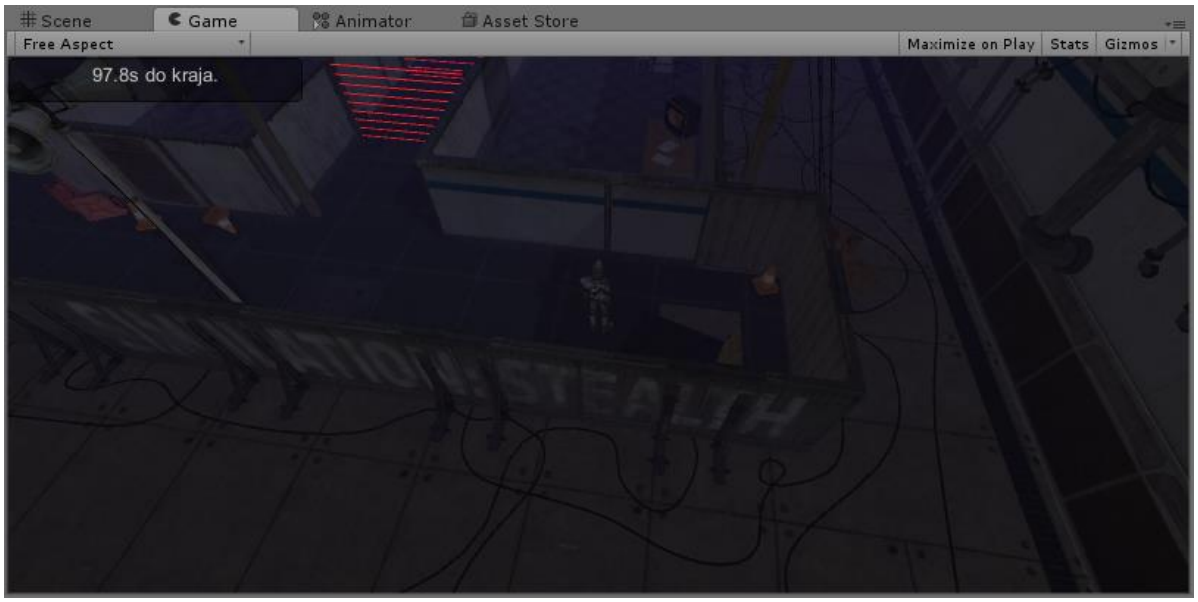
Na slici (Sl. 8.3) prikazan je okvir scene. U ovaj okvir dodajemo sve elemente igre te ih slažemo u 3D prostoru. Pri pokretanju igre svi se elementi stvaraju na mjestu na koje smo ih postavili stoga potrebno je pripaziti na preklapanje 3D objekata. U sceni, elementi igre su nepokretni.

Zbog performansi računala i veličine same igre potrebno je podijeliti igru na manje dijelove tj. scene. Slično radimo i s programskim kodom u OOP. Umjesto da se program sastoji od jednog velikog programskog koda, programski kod dijelimo na dijelove (module) tako da svakom elementu programa (igre) pripadaju vlastite skripte koje sadrže klase što ga opisuju i daju mu funkcionalnost. Tako promjena i greške u jednoj sceni ne utječu na cjelokupnu igru. Isto tako izmjene, dodatci i greške unutar jedne skripte ne utječu na ostatak programskog koda.



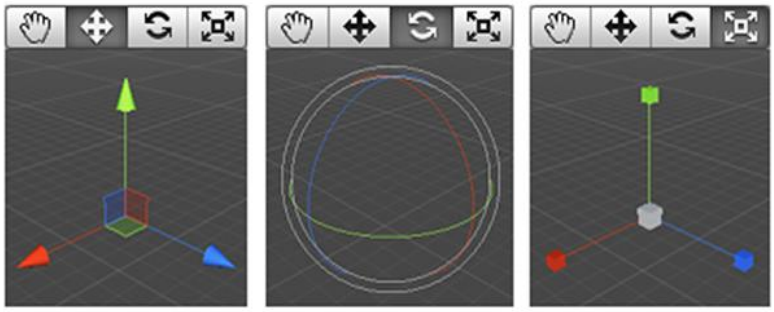
Sl. 8.3 Okvir scene

Okvir igre prikazan je na slici (Sl. 8.4), a služi za pregled igre u pokretu. Za razliku od okvira scene, u ovom okviru ne možemo slobodno pomicati 3D objekte nego samo onda kada je to jedna od funkcionalnosti igre tj. kada napišemo programski kod za taj dio.



Sl. 8.4 Okvir igre

Na slici (Sl. 8.5) prikazana su dugmad alatne trake skupa s prikazom alata u okviru scene (odabrani alati odgovaraju odabranom dugmetu, označeno sivom bojom). Prvi alat služi za pomicanje, drugi za rotiranje, a treći za promjenu mjerila 3D objekta u prostoru.



Sl. 8.5 Dugmad za manipuliranje 3D objektima i perspektiva

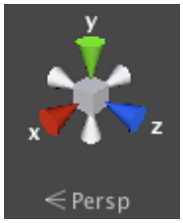
Za razliku od prethodnih alata i dugmadi, dugmad na sljedećoj slici ne služe za manipuliranje 3D objektima nego za manipuliranje pogleda u sceni. Prvo označeno dugme na slici (Sl. 8.6) služi za pomicanje pogleda (lijevi klik mišem), drugo označeno dugme služi za rotiranje pogleda (lijevi klik mišem) i treće označeno dugme služi za zumiranje pogleda (Alt + desni klik mišem).

Sl. 8.6 Dugmad za manipuliranje pogledom

Dugmad na slici (Sl. 8.7) odnose se redom na dugme za pokretanje igre, pauziranje igre te pokretanje igre za samo jednu „sliku“ (eng. Frame). Klikom na „Play“ dugme automatski se prebacujemo s okvira scene na okvir igre.

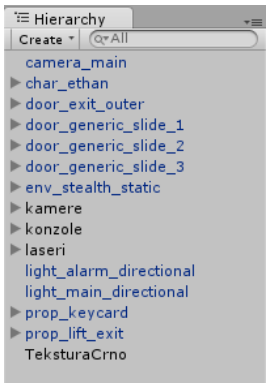
Sl. 8.7 Dugmad za upravljanje igrom

Alat sa slike (Sl. 8.8) ne treba uključivati kao prethodne jer on stoji u gornjem desnom kutu okvira scene i služi za mijenjanje perspektive isključivo unutar scene. Perspektivu mijenjamo klikom na ovaj alat. Pri pisanju koda potrebno je paziti koja os koordinatnog sustava gleda u kojem smjeru. Također potrebno je za napomenuti kako pretvaranje koordinata iz 3D u 2D daje naopačke okrenutu y-os.



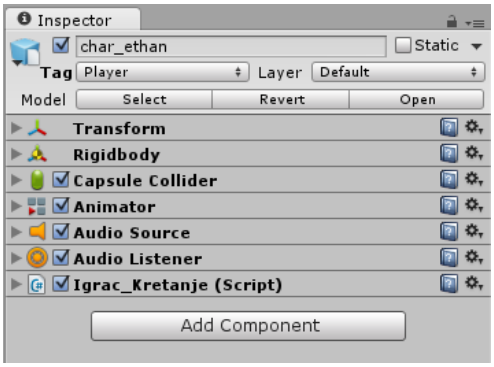
Sl. 8.8 Alat za promjenu perspektive scene

Okvir hijerarhije (eng. *Hierarchy*) odnosi se na trenutno učitanu scenu (Sl. 8.9). U ovaj okvir dodajemo elemente iz okvira „Project“ (Sl. 8.1). Kada dodamo elemente u ovaj okvir oni se automatski prikazuju i u sceni, obrnuto vrijedi ako element prvo dodamo u okvir scene. Klikom na bilo koji element u ovom okviru se prikažu sve komponente vezane na odabrani element unutar okvira zvanog *Inspector* (Sl. 8.9).



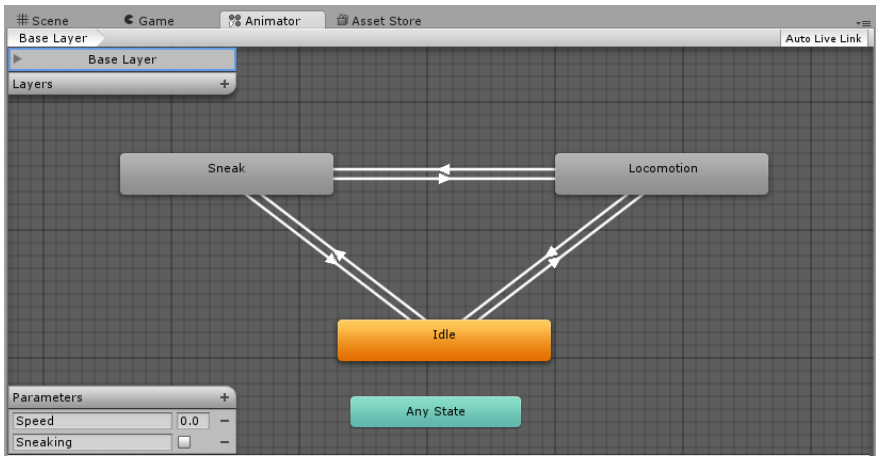
Sl. 8.9 Okvir hijerarhije trenutne scene

Inspector je okvir u kojem možemo vidjeti sva obilježja i komponente koje su vezane na odabrani objekt (Sl. 8.10). Također u ovom okviru otvaraju se i druge postavke kao npr. postavke projekta ili animatora.



Sl. 8.10 Okvir za pregled komponenti vezanih za odabrani objekt

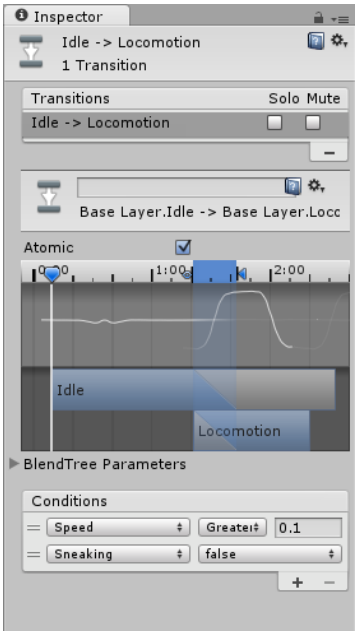
Animator je okvir gdje spajamo animacije u jednu cjelinu. U slučaju sa slike (Sl. 8.11) dodane osnovne animacije lika kojeg želimo pokretati u igri, a to su mirovanje, prikradanje i kretanje koje se sastoji od šetanja i trčanja.



Sl. 8.11 Okvir za podešavanje animatora (upravljanje animacijama)

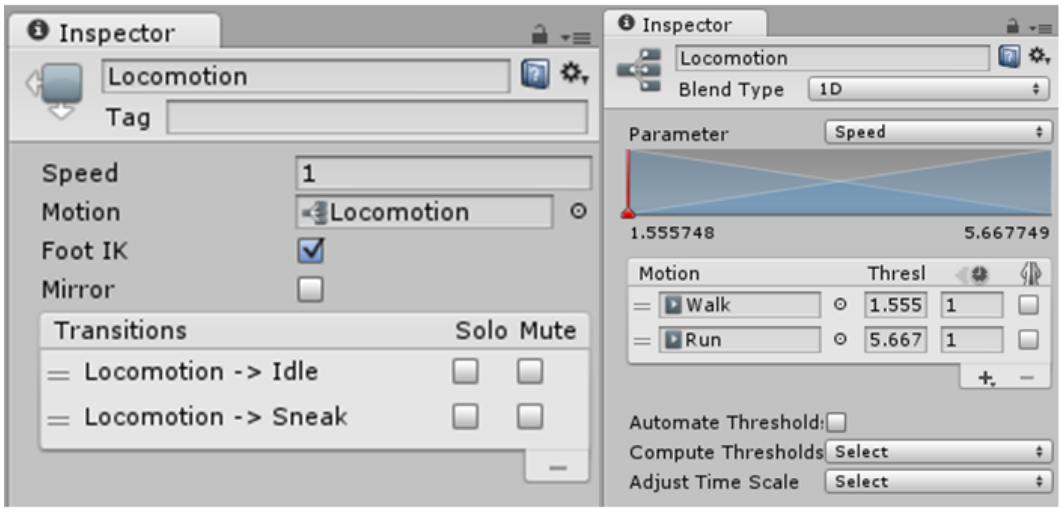
Animator je osmišljen tako da animacije kontroliramo preko parametara, stanja, prijelaza između stanja i slojeva. Za početak potrebno je znati kako funkcionira animator kako bi s lakoćom pisali programski kod. Na slici (Sl. 8.11) imamo 4 stanja za korištenje (Osnovno stanje mirovanja – bez unosa, dva stanja s unosom i stanje koje predstavlja sva ostala stanja). Unutar svakog stanja postavljena je pripadajuća animacija ili stablo animacija. Sva stanja su povezana strelicama (imaju prijelaz) osim stanja zvanog „*Any State*“. Ako na „*Any State*“ povežemo neko drugo stanje, to znači kako u to drugo stanje možemo prijeći iz svih stanja stoga nije potrebno raditi prijelaze sa svih ostali stanja.

Kako bi kontrolirali prijelaze animacija koristimo parametre. U ovom primjeru imamo dva parametra i to brzina (eng. *Speed*) i prikradanje (eng. *Sneaking*). Parametri su ništa drugo nego varijable. Kada kliknemo na strelice prijelaza animacija otvore nam se postavke u *inspector-u*. U postavkama zadajemo koje parametre želimo koristiti za kontrolu prijelaza te pri kojoj vrijednosti parametara (pri kojim uvjetima) će se prijelaz dogoditi (Sl. 8.12). Kada se ne bi koristili animatorom trebali bi sami učitati sve animacije te postavljati uvjete kada pokrenuti, a kada zaustaviti određenu animaciju.



Sl. 8.12 Prikaz postavki prijelaza između dva stanja

Također kod stabla animacija gdje imamo više animacija za isto stanje (u primjeru sa slike (Sl. 8.13) šetanje i trčanje) potrebno je parametrima kontrolirati koju animaciju koristiti. Na slici se jasno vidi koji parametar je odabran te koje su vrijednosti parametra potrebne kako bi se određena animacija pokrenula. Pod „*Threshold*“ vidimo donju i gornju granicu promjene animacije. Za sve vrijednosti između granica Unity računa vrijednost preklapanja animacija pa će lik trčati polagano npr. za Speed=3.



Sl. 8.13 Prikaz postavki stabla animacija (eng. *Blend Tree*)

Postavke animacija potrebno je poznavati za lakše orijentiranje unutar programskog koda. Trebamo poznavati ograničenja i uvjete pod kojima se pokreće željena animacija.

Osnovni elementi i klase igre

Osnovni elementi od kojih se izrađuje igra su 3D modeli, 2D teksture, animacije, audio datoteke, materijali i dr.. Kombinacijom prethodno navedenih elemenata dobijemo manje ili više realističnu reprezentaciju objekata iz svakodnevnog života ili mašte, ali kako bi im „udahnuli život“ potrebno ih je opisati programskim kodom tj. klasama. Za to, kroz ovu skriptu, dan je primjer programskog koda pisanog u C# jeziku.

Za početak potrebno je poznavati osnovne klase iz Unity biblioteke kako bi uspješno manipulirali 3D objektima i iščitavali potrebne informacije sadržane u objektima tih klasa. Današnje programiranje se u većini slučajeva svodi na rukovanje već gotovim bibliotekama. Unity je odličan primjer jer se od nas ne zahtjeva razumijevanje na koji način je izrađen ovaj game engine nego u kojem kontekstu koristiti njegove elemente. To uključuje poznavanje osnovnih koncepata OOP-a.

Osnovna klasa koja se vezuje na sve elemente koje možemo dodati u igru je klasa GameObject. Ta klasa se automatski veže na sve što dodajemo u scenu, npr. kada dodamo 3D model u scenu njega automatski opisuje klasa GameObject. Unutar te klase postoje reference na različite objekte koje možemo, a i ne moramo dodati. Ukoliko ne dodamo referencu na određeni objekt, u programu će nam biti prikazana vrijednost null. Objekti koji se najčešće vežu na objekt klase Gameobject su (Slika 1.10) objekti sljedećih klasa:

- Transform – Klasa kojom opisujemo trenutnu poziciju, rotaciju i mjerilo objekta (ove vrijednosti možemo podešavati unutar skripti ili „ručno“ preko alata sa slike 1.5)
- Rigidbody – Klasa koja služi za omogućavanje simulacije fizike nad objektom
- Collider – Klasa koja služi za detekciju sudara između dvaju ili više objekata igre
- Animation – Klasa koja sadrži informacije vezane za trenutno vezanu animaciju na objekt (ako imamo više od jedne animacije koje želimo vezati na određeni 3D objekt potrebno je koristiti se klasom Animator, klasa Animator je vezana za animator kontroler opisan u prethodnom poglavlju)
- Vlastita klasa – Možemo dodavati vlastite skripte koje sadrže klase s vlastitim opisima i funkcionalnostima objekta. Funkcionalnosti se mogu ostvariti koristeći Unity ili vlastitu biblioteku. Broj skripti na jednom objektu nije ograničen.

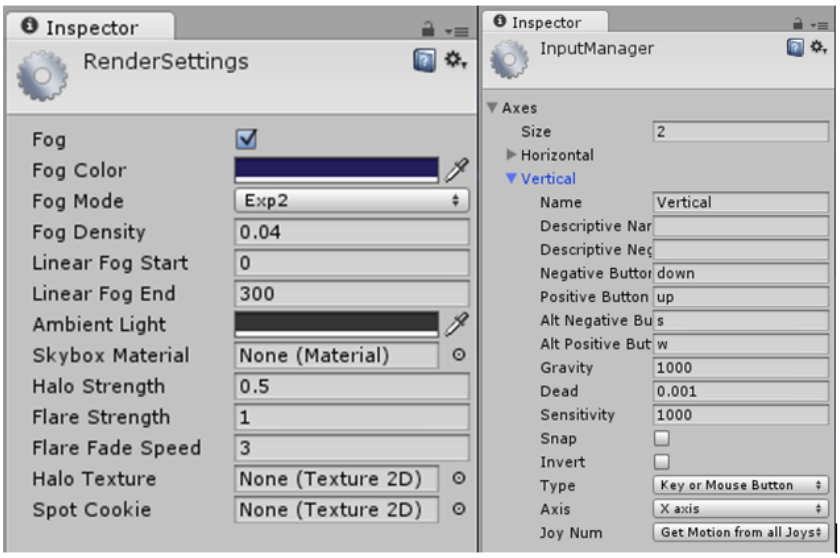
Bitno je za zapamtiti kako u klasi Gameobject nisu rezervirana mjesta za sve klase iz Unity biblioteke. Npr. kada kontroler klase Animator vežemo na neki objekt klase Gameobject, unutar vlastite skripte trebamo implementirati dohvaćanje kontrolera dok za objekte

prethodno navedenih klasa to nije potrebno. Više o gotovim Unity klasama i dohvaćanjem objekata u narednim poglavljima.

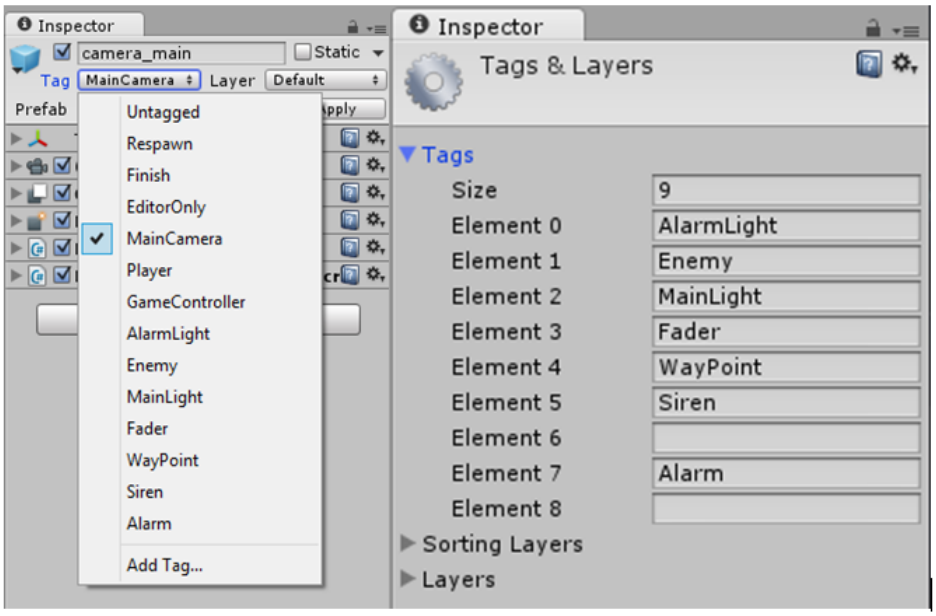
Postavke

Render settings – neke od grafičkih postavki kao zamagljivanje pozadinskih elemenata igre (eng. Fog) ili dodavanje materijala koje predstavlja nebo (eng. Skybox material)

Input settings – postavke za unos. U ovim postavkama možemo dodavati vlastite kontrole za upravljanje igrom te pridijeliti bilo koje kontrole s ulaznih jedinica našeg računala. U primjeru sa slike (Sl. 8.14) je prikazan unos po vertikali (tipke down, up, S, W). Otpuštenu tipku igra čita kao nulu dok pritisnutu tipku čita kao jedinicu. Osjetljivost (eng. Sensitvity) nam određuje brzinu kojom će tipka biti u potpunosti pritisnuta (1) dok gravitacija (eng. Gravity) određuje kojom brzinom će, nakon otpuštanja tipke, vrijednost pritisnute tipke vratiti na 0. Dead označava vrijednost tipke ispod koje će se ista smatrati u potpunosti otpuštenom (0). Kada napravimo vlastite postavke, u programskom kodu možemo pristupiti našim vrijednostima unosa.



Sl. 8.14 Postavke grafike i ulaznih jedinica

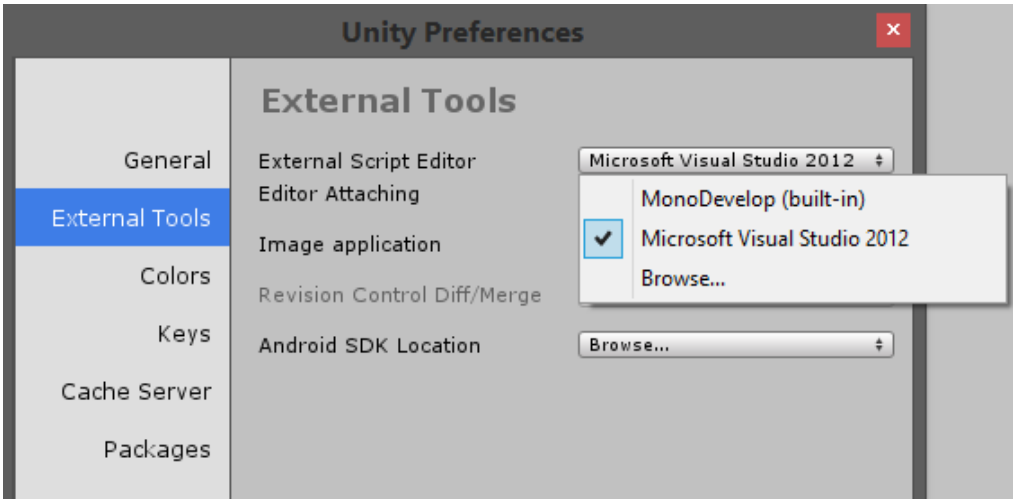


Sl. 8.15 Oznake

Oznake (eng. *Tag*) koristimo radi lakšeg prepoznavanja i lakšeg dohvaćanja objekata u igri. Oznake možemo dodati pri mijenjanju postavki objekta u *Inspector* okviru ili pod postavkama projekta unutar okvira *Tags and Layers* (Sl. 8.15). Oznake mogu biti pridijeljene jednom objektu, ali isto tako i nizu objekata stoga pri pisanju koda potrebno je paziti što dohvaćamo. Ako ne koristimo oznake potrebno je pretraživati objekte po potpunom nazivu objekta u igri. Dohvaćanje objekata je nužno kako bi se ostvarila komunikacija između različitih elemenata igre, a oznake nam u tome olakšaju posao.

8.1.2. Okruženja za programiranje

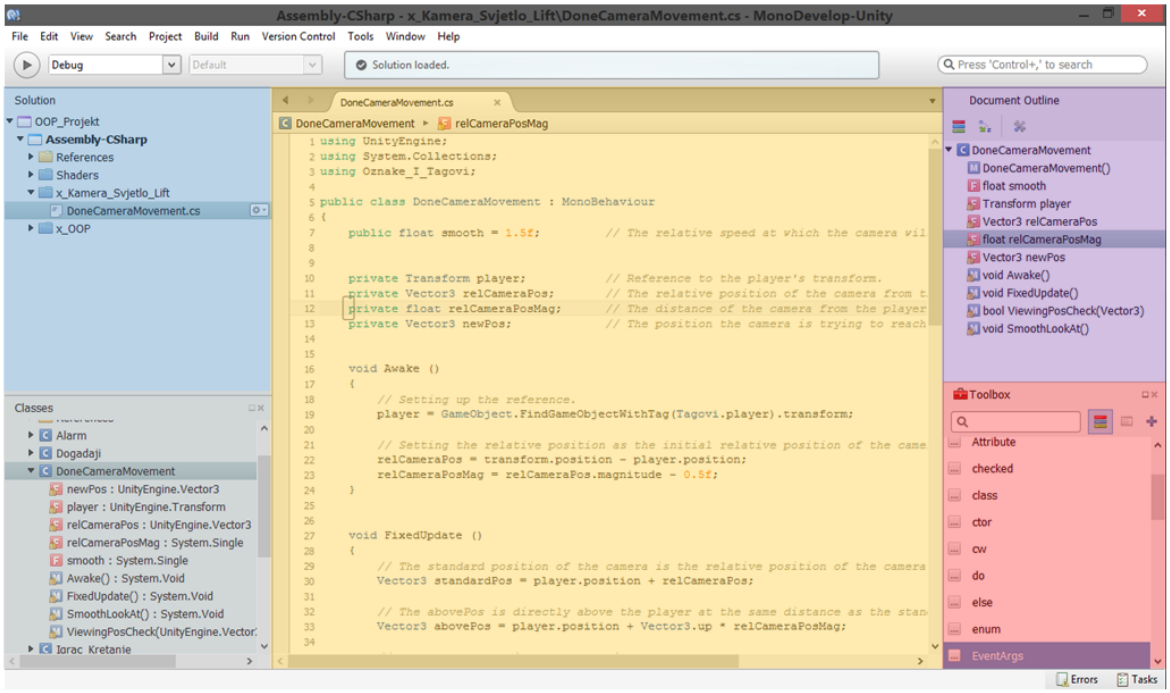
Za pisanje programskog koda možemo koristiti bilo koje okruženje koje podržava iste programske jezike kao i Unity. Samom instalacijom Unity paketa instaliramo i program zvan Monodevelop. Monodevelop je jako jednostavan i brz. Za one koji imaju vlastiti program kojim se svakodnevno koriste mogu ga odabrati u postavkama kao primarni alat za uređivanje skripti. Na slici (Sl. 8.16) dan je primjer odabira Visual Studio 2012 kao primarnog alata. Mijenjanje primarnog alata možemo uraditi unutar postavki projekta (External Tools).



Sl. 8.16 Odabir primarnog alata za pisanje programskog koda

Za uspješno snalaženje unutar Monodevelop programskog okruženja potrebno je poznavati pet osnovnih okvira (Sl. 8.17):

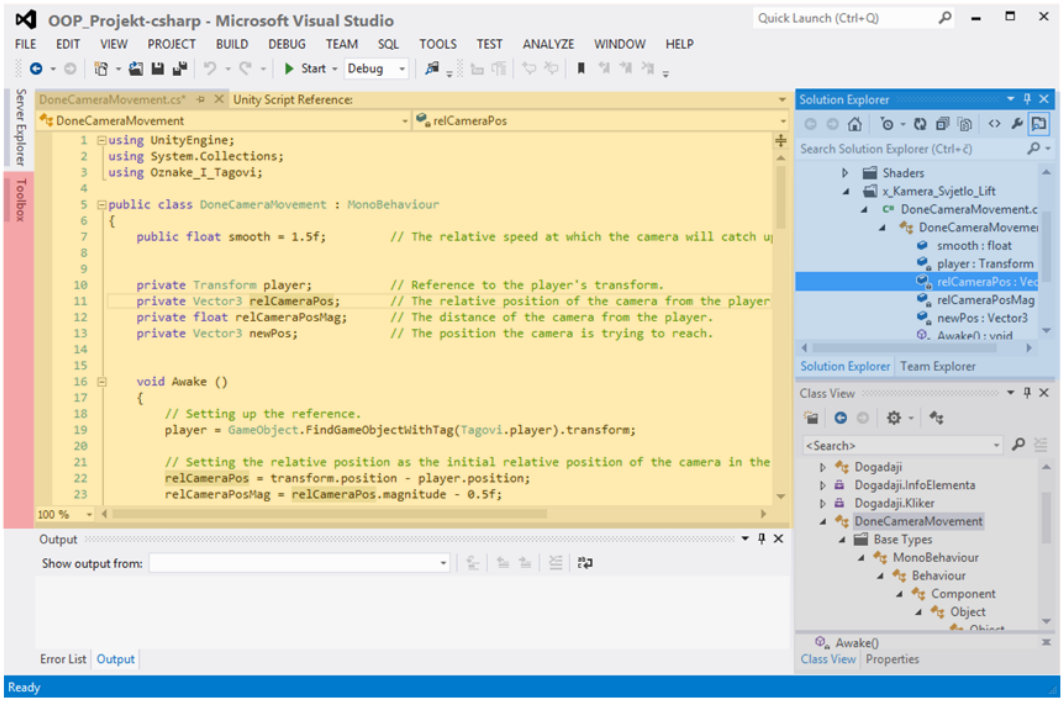
- Solution – okvir za pregled svih skripti projekta
- Classes – hijerarhijski pregled svih klasa i sadržaja unutar tih klasa
- Document outline – okvir za pregled elemenata trenutno otvorene skripte, klikom na element u skripti „skačemo“ na taj dio koda
- Toolbox – popis svih ključnih riječi i naredbi odabranog programskog jezika, dvostrukim klikom na naredbu stvaramo primjer koda u trenutnoj skripti
- Okvir za pisanje koda – u ovom okviru je prikazan sav programski kod odabrane skripte spreman za uređivanje



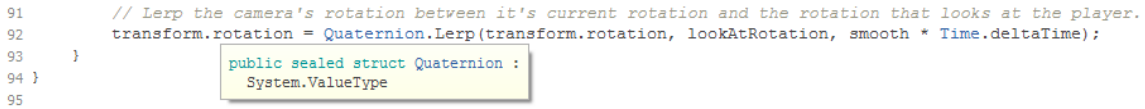
Sl. 8.17 Monodevelop

Za uspješno snalaženje unutar Visual Studio 2012 programskog okruženja potrebno je poznavati četiri osnovna okvira (Sl. 8.18):

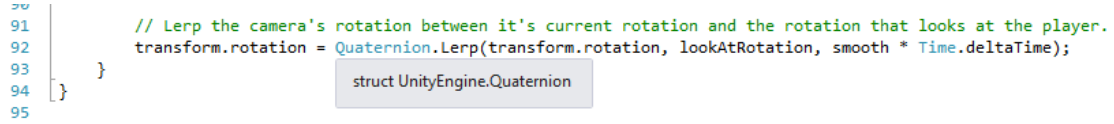
- Solution Explorer – okvir za pregled svih skripti projekta i njihovog sadržaja (klikom na bilo koji od elemenata otvorene skripte program pronalazi sve elemente s tim imenom)
- Class View – pregled svih klasa (klikom na klasu nam se prikažu informacije o nasljeđivanju)
- Toolbox – popis elemenata koje možemo dodavati u skripte (u ovom slučaju prazan, inače bude popis klasa koje predstavljaju Label, TextBox, Button i sl.)
- Okvir za pisanje koda – u ovom okviru je prikazan sav programski kod odabrane skripte spreman za uređivanje



Sl. 8.18 Visual Studio 2012

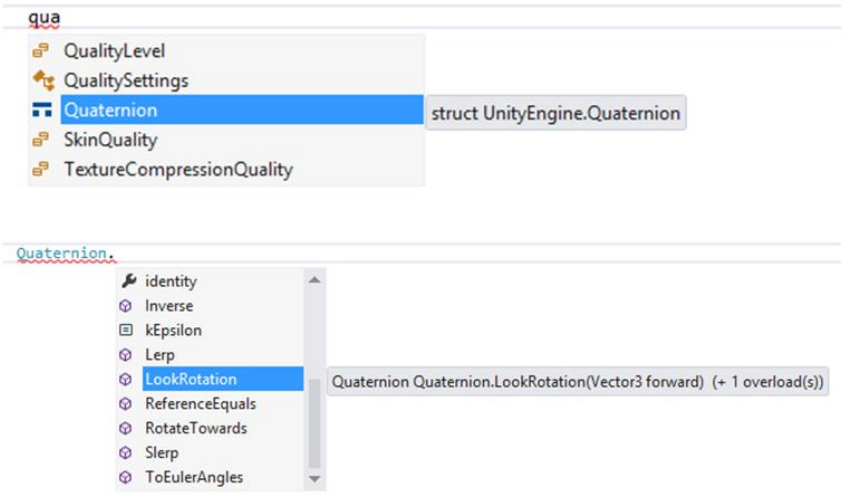


Sl. 8.19 Informacije za napisani kod Mono



Sl. 8.20 Informacije za napisani kod VS2012

Sl. 8.21 Informacije pri pisanju koda, Monodevelop



Sl. 8.22 Informacije pri pisanju koda VS-2012

Na slici (Sl. 8.17) i (Sl. 8.18) se vidi kako postoje različiti okviri za lako i brzo snalaženje po programskom kodu. Isto tako pri pisanju koda nam se nude razne informacije o onome što pišemo. Na slici (Sl. 8.19) i (Sl. 8.20) je prikazano kako prelaskom miša preko neke riječi dobivamo potpunu informaciju o istoj. Slično vrijedi i za pisanje koda (Sl. 8.21 i Sl. 8.22). Dok pišemo program nam sam nudi sve elemente pod tim nazivom uz detaljan opis elementa kao što su tip, pristupačnost, ako se radi o metodi onda dobijemo prikaz traženih parametara.

Ova funkcionalnost je jako korisna jer možemo s lakoćom provjeriti u kojem kontekstu određeni element možemo koristiti.

8.1.3. Zahtjevi korisnika i dijagram klasa

Potrebno je izraditi igru gdje će korisnici upravljati zadanim likom kojeg prati kamera igre. Igra treba sadržavati prepreke za igrače i dinamičan svijet. Igraču treba omogućiti interakciju s okruženjem unutar igre. Cilj igre je pronaći karticu te otvoriti vrata što će omogućiti liku bijeg liftom prije isteka vremena.

Sve komponente igre su pripremljene, a zadatak je sve te komponente spojiti tj. isprogramirati kako bi dobili funkcionalnu igru. U projektu se nalazi samo jedna skripta za kameru koja prati lika po igri, sve ostale skripte potrebno je napraviti.

Igra treba sadržavati:

- Lika kojim igrač upravlja (igrač treba imati torbu u koju će spremiti karticu)
- Glavnu kameru koja prati igrača

- Dvije vrste vrata, automatska i vrata koja se otvaraju karticom
- Karticu za vrata
- Kamere sa senzorom za detektiranje lika
- Dvije vrste lasera sa senzorom za detektiranje lika, s automatskim gašenjem i paljenjem te lasere koji se gase/pale preko konzole
- Konzole preko kojih igrač gasi lasere
- Alarm kojeg je potrebno upaliti kada laseri ili kamere detektiraju lika
- Lift koji predstavlja cilj
- Na grafičkom sučelju potrebno je prikazati:
  - vrijeme prije završetka igre
  - informacije o načinu interakcije s kamerom, konzolom i vratima za koja treba kartica
  - indikator o stanju konzole i kamere (ON/OFF)

Pri samom početku potrebno je sagledati sve elemente od kojih se igra sastoji te nacrtati početni dijagram klasa prema kojem ćemo polagano graditi programski kod (Sl. 8.23).

Sl. 8.23 Dijagram klasa

Klase su podijeljene u 6 skupina prema funkcionalnosti. Za upravljanje likom nam treba jedna klasa kao i za upravljanje glavnom kamerom jer u ovom slučaju imamo samo jednog igrača kojeg prati jedna kamera. Za upravljanje igrom nam također treba jedna skripta jer će ta skripta upravljati učitavanjem i prekidom igre. Elemente grafičkog sučelja smo mogli

podijeliti po klasama za pojedine elemente, ali u ovom primjeru su svi elementi grafičkog sučelja u istoj klasi zbog smanjenja količine i preglednosti programskog koda. Elemente svijeta potrebno je odvojiti u posebne klase i tako osposobljavati svaki element posebno. Kada jedan element funkcionira možemo prijeći na sljedeći, a implementacija sljedećeg ne utječe na funkcionalnost prethodnog osim ako ponašanje jednog ne uvjetuje ponašanje drugog elementa.

8.2. Tipovi podataka

Kroz ovu vježbu upoznati ćemo se s osnovnim tipovima podataka, isključivo s klasama i hijerarhijom istih. Upoznati ćemo se s nekim gotovim klasama i strukturama iz Unity biblioteke. Svako naredno poglavlje, uključujući i ovo, sastoji se od informativnog dijela za one koji nisu dovoljno upoznati s objektno orijentiranim programiranjem te dijela koji je vodič kroz vježbe. Zadatci se nastavljaju na informativni/teorijski dio koji im prethodi.

8.2.1. Klasa i objekt

Tipove podataka dijelimo u dvije skupine, referentni i vrijednosni. Referentni tip je onaj koji sadrži adresu memorijske lokacije podataka dok vrijednosni tip sadrži vrijednost podatka. Najbolji primjer za referentni tip je klasa (eng. *class*) i struktura (eng. *struct*) Ako promotrimo slučaj sa slike (Sl. 8.24) vidimo kako imamo dva tipa podataka (referentni i vrijednosni) koja ćemo zbog demonstracije predstaviti kao klasu i strukturu.

Elementi u sivom okviru (igrac1 i igrac2) u ovom slučaju predstavljaju objekte, a korištenjem znaka jednakosti kopira se samo referenca na objekt što znači da igrac1 i igrac2 pokazuju na iste podatke u memoriji. Elementi igrac3 i igrac4 bi predstavljali varijable što znači da korištenjem znaka jednakosti kopiramo vrijednosti iz igrac3 u igrac4 te sada u memoriji postoje isti podatci na dvije različite lokacije.

Sl. 8.24 Referentni i vrijednosni tip

`Class` i `Struct` su jako slični, a sastoje se od atributa (druge varijable ili objekti – opisni dio), metoda (funkcionalnost) i konstruktora (za stvaranje istih - instanciranje).

Osim Struct-a postoje i drugi vrijednosni tipovi kao cijeli broj (`int`), decimalni broj (`float`), `boolean` vrijednost (`bool`), `enum`… Tekstualni tip podatka (`string`), iako se koristi kao vrijednosni je zapravo referentni tip.

8.2.2. Statički elementi, konstante

Elemente unutar klasa i struktura možemo podijeliti u dvije skupine:

- Elementi za koje nije potrebna instanca (svi elementi koji uključuju ključne riječi static ili const)
  - pristupamo im preko naziva klase tj. `NazivKlase.element` ili `NazivStructa.element`
- Elementi za koje je potrebna instanca (svi elementi koji ne uključuju ključne riječi static ili const)
  - pristupamo im preko naziva objekta tj. `mojObjekt.element`

Ako instanciramo više različitih objekata iste klase tada su statički elementi i konstante isti za sve instance dok su svi ostali elementi različiti za različite instance. Konstante su nakon kompajliranja nepromjenjive dok se statički elementi, tijekom izvođenja programa, mogu mijenjati. Također, i klasa može biti statička što znači da se ne može instancirati i mora sadržati samo konstante ili statičke elemente. Ukoliko želimo instancirati klase trebamo ih učiniti javnima koristeći ključnu riječ `public` s tim da ne smiju sadržati riječ `static` pri deklaraciji.

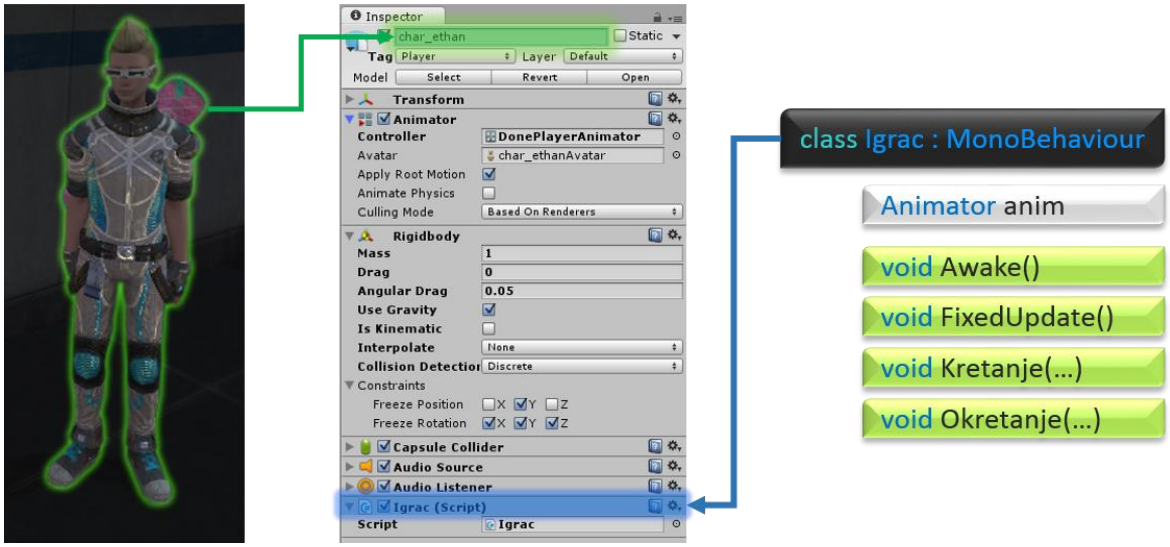
8.2.3. Imenski prostor (*Namespace*)

Imenski prostori su ništa drugo nego spremnici za podatke kao što su `Class`, `Struct`, `Namespace`, `Interface`… Kada otvorimo bilo koji dokument pisan u C# jeziku prvo što piše na samom vrhu skripte su korišteni imenski prostori. Za korištenje svih elemenata imenskog prostora upisujemo ključnu riječ `using` te naziv imenskog prostora. Kada se referenciramo na određeni imenski prostor, možemo koristiti njegov sadržaj upisivanjem naziva određenog elementa.

8.2.4. Vježba br_1

Potrebno je izgraditi klasu `Igrac` prikazanu na dijagramu klasa (Sl. 8.23). Na slici (Sl. 8.25) prikazan je ciljani izgled klase. Prisjetimo se podpoglavlja 1.1.2, rečeno je kako na

trenutnom objektu igre postoje određene komponente koje su također vidljive i na slici 3.2. Jednom dijelu komponenti (poput objekta `transform` klase `Transform`) možemo pristupiti direktno unutar klase `Igrac` (taj objekt je prethodno ugrađen) dok elemente klase Animator trebamo dohvatiti.

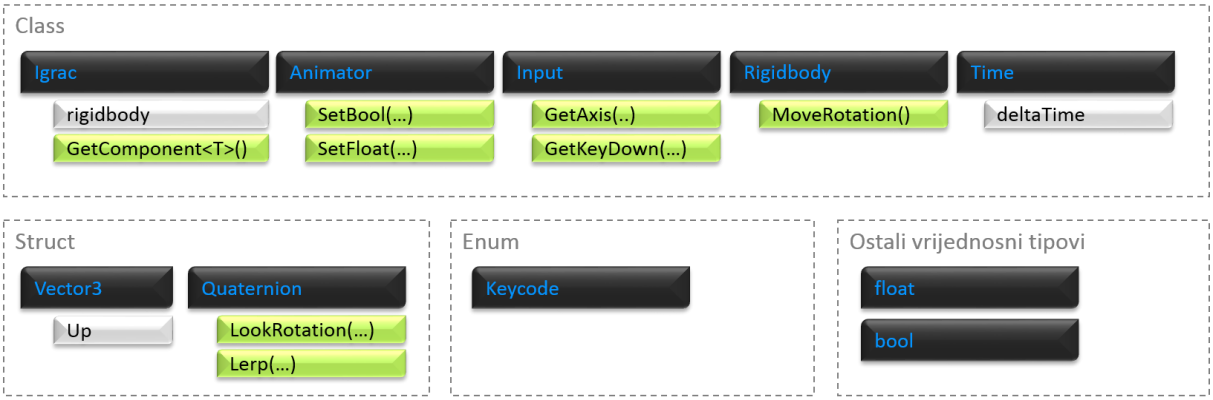


Sl. 8.25 Klasa Igrac i veza s likom iz igre

Metode `Awake()` i `FixedUpdate()` su već ugrađene metode koje možemo, a i ne moramo koristiti. Iste metode moraju biti tipa `void` i ne smiju primati niti jedan parametar. Metoda `Awake()` se pokreće pri osposobljavanju skripte (klase u skripti) dok se metoda `FixedUpdate()` pokreće neposredno prije izračunavanja fizike u igri. Postoji još ugrađenih metoda, ali o tome više u nastavku.

Bitno: izvršavanje programskog koda u skriptama, preko ugrađenih metoda, vrši se više puta u sekundi jer se tako omoguće konstantni izračuni stanja igre. Ovime se ostvaruje pokretljivost igre.

Za pokretanje lika tj. dodavanje funkcionalnosti istome (kretanje i okretanje) koristiti ćemo se Unity bibliotekom. Na slici (Sl. 8.26) prikazani su podatci potrebni za izvršavanje zadatka. Klase i strukture sastoje se od mnogo više metoda, atributa i konstruktora, a na slici su prikazani samo trenutno potrebni.



Sl. 8.26 Podatci potrebni za izvršavanje zadatka

Korak 1: Otvoriti već pripremljenu scenu koja se nalazi među projektnim dokumentima pod nazivom Scena_za_vjezbe

Korak 2: Napraviti novu skriptu pod nazivom Igrac te ju spremiti u direktorij Skripte – Vjezba1

Korak 3: Vezati skriptu na lika u igri (char_ethan) koji se nalazi u okviru Hijerarhije scene

Korak 4: Implementirati sve atribute i metode sa slike (Sl. 8.25) (neka budu prazne i bez parametara)

Korak 5: Unutar metode Awake() treba dohvatiti „Animator Controller“ prikazan na slici (Sl. 8.25) i vezan na igrača skupa s našom skriptom Igrac. Element koji dohvaćamo je klase Animator.

- GetComponent<T>() – generička metoda klase Component. Ovoj metodi možemo pristupati izravno iz vlastite klase jer je ta metoda također već ugrađena (ova metoda je primjer koji je ugrađen u sve objekte klase MonoBehaviour i GameObject). Ovaj tip metode kao parametar prima tip podatka stoga na mjesto „T“ treba upisati naziv Klase elementa kojeg dohvaćamo.
- Paziti u kojem kontekstu se ova metoda treba koristiti (za info pokazivačem miša prijeći preko iste).
- Ako je metoda ispravno korištena parametar „anim“ bi trebao sadržati referencu na kontroler

Korak 6: Unutar metode FixedUpdate() treba učitati unose s tipkovnice za „Axes“ horizontale i vertikale te za tipku LeftShift kojom će se naš lik prikradati

- GetAxis(…) i GetKey(…) su statičke metode iz klase Input.

- Metoda GetAxis(…) vraća float vrijednost trenutno pritisnutih tipki definiranih pod „Axes“
  - Prisjetimo se poglavlja (8.1.1) i što je definirano u postavkama ulaznih jedinica pod „Axes“ te promotrimo koje parametre zahtjeva metoda GetAxis(…)
- Metoda GetKey(…) vraća bool vrijednost o pritisnutoj, traženoj, tipki (tipka koja je zadržana).
  - KeyCode – podatak tipa enum. Keycode sadrži sve tipke tipkovnice i miša. Promotrimo koje parametre zahtjeva metoda GetKey(…)

Korak 7: Urediti metodu Kretanje() tako da prima parametre koji predstavljaju informacije o unosu iz prethodnog koraka. Metoda treba preko „Animator Controller-a“ upravljati parametrima o kojima ovisi izbor animacija. Parametrima lako možemo upravljati uz pomoć if-else uvjeta.

- U koraku 5 smo dohvatili „Animator Controller“ i referencu spremili u „anim“
- Preko metoda SetBool(…) i SetFloat(…), koje posjeduje „anim“, postavljamo vrijednost parametara toga istog animatora uz pomoć naziva parametra (napomena – ovo nisu statičke metode)
  - Nazive parametara i granične vrijednosti istih su postavljene u animatoru (pogledati poglavlje 8.1.1 – Slike Sl. 8.11, Sl. 8.12, Sl. 8.13 ili u Unity-u datoteku DonePlayerAnimator)
  - Provjeriti u kojem kontekstu se koriste metode (vraćaju li parametre ili su tipa void te koje točno parametre primaju)
- Posebno promotriti parametre metode SetFloat(…)
  - dampTime i deltaTime služe za ublažavanje prijelaza između dvije vrijednosti parametra (prethodne i nove vrijednosti). Kako se skripte pokreću više puta u sekundi, to nam omogućuje izračune i simulaciju blagih prijelaza određenih vrijednosti. To znači, što više ublažimo prijelaz parametra to će lik sporije prelaziti iz hoda u trk. dampTime postavimo proizvoljno npr. 0.1f dok za deltaTime koristimo statički parametar Time.deltaTime koji nam daje trenutnu vrijednost vremena u odnosu na prethodno pokretanje skripti
  - za testiranje prijelaza iz šetnje u trk mijenjajte vrijednost za dampTime
- Ako je metoda Kretanje(…) uspješno napravljena potrebno ju je pozvati unutar metode FixedUpdate() neposredno nakon učitavanja unosa (metoda je tipa void – paziti kako ju pozvati), a zatim pokrenuti igru i provjeriti kreće li se lik pri unosu s tipkovnice

Korak 8: Urediti metodu Okretanje() tako da zahtjeva dva parametra (informacije o unosu vertikale i horizontale iz koraka 6). Metoda treba okretati lika preko objekta rigidbody klase Rigidbody

- Objekt rigidbody je sadržan samim stvaranjem skripte ako je naznačeno :

  MonoBehaviour (prisjetimo se, Rigidbody je klasa zadužena za izračun fizike)

  – Metoda MoveRotation(…) će okrenuti 3D objekt u smjeru koji je zadan parametrom metode (ukoliko ima vezan rigidbody na sebe)
  – rigidbody.rotation – je javni atribut koji nam daje informacije o trenutnoj poziciji rotacije
- Kako rigidbody.MoveRotation() nema mogućnost ublažavanja prijelaza između dva smjera rotacija potrebno je svakim izvršavanjem skripte:

  – učitati ulazne vrijednosti i odrediti smjer unosa
  – zatim prema tim ulaznim vrijednostima napraviti novu, željenu, rotaciju
  – nakon toga u odnosu na trenutnu rotaciju računati vrijednost nove rotacije uzimajući u obzir vrijeme
  – u konačnici novu vrijednost rotacije treba postaviti na rigidbody objekt.
  – Prethodne četiri točke predstavljaju svaki korak (liniju) metode Okretanje() kojeg je potrebno opisati programskim kodom
- Vector3 je struct u kojeg možemo pospremiti informacije o poziciji bilo kojeg objekta igre

  – Novi vektor možemo napraviti koristeći jedan od konstruktora (new Vector3(…))
  – Cilj je dobiti smjer u igri koji odgovara unosu horizontale i vertikale (horizontalu promatramo kao os-x, a vertikalu kao os-z u svijetu, lik ne može letjeti stoga je os-y = 0)
  – Vector3.up – statički parametar koji predstavlja jedinični vektor u smjeru osi y tj. predstavlja os-y
- Quaternion je struct u kojeg možemo pospremiti informacije o rotaciji bilo kojeg objekta igre

  – LookRotation(Vector3 forward, Vector3 upwards) je statička metoda iz struct-a Quaternion koja, kao rezultat, vraća novu varijablu tipa Quaternion (forward – smjer u prema kojem će se rotacija izvršiti, upwards – os oko koje želimo vršiti rotaciju)
  – Lerp(Quaternion from, Quaternion to, float t) je također statička metoda koja vraća novu varijablu tipa Quaternion tj. vraća novu vrijednost rotacije koja se računa prema parametrima (from – od koje rotacije želimo krenuti, to – do koje rotacije se treba zakrenuti, t – koliki će pomak biti u jedinici vremena npr. Time.deltaTime * 15f)
- Instanciranje vršimo preko konstruktora koristeći ključnu riječ new i naziv klase ili strukture

  – npr. Vector3 mojVektor = new Vector3(…parametri ili bez parametara…)

```
using UnityEngine;
using System.Collections;
```

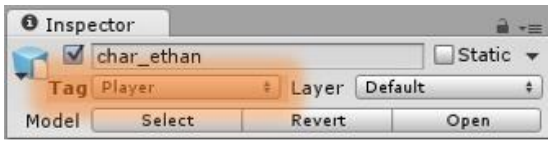
```
using namOznake;
public class Igrac : MonoBehaviour {
	Animator anim;
	public Torba mojatorba;
	void Awake () {
		anim = GetComponent<Animator>();
		mojatorba = new Torba();
	}
	void FixedUpdate (){
		float h = Input.GetAxis("Horizontal");
		float v = Input.GetAxis("Vertical");
		bool prikradanje =
		Input.GetKeyDown(KeyCode.LeftShift);
		Kretanje(h, v, prikradanje);
	}
	void Kretanje (float h, float v, bool p){
		anim.SetBool(Anim_Oznake.bool_prikradanje, p);
		if(h != 0f || v != 0f){
			Okretanje(h, v);
			anim.SetFloat(
				Anim_Oznake.float_brzina,
				5.5f, 0.1f, Time.deltaTime);
		}
		else anim.SetFloat(Anim_Oznake.float_brzina, 0);
	}
	void Okretanje (float h, float v){
		Vector3 smjer_unosa = new Vector3(h, 0f, v);
		Quaternion zeljeni_smjer =
		Quaternion.LookRotation(smjer_unosa, Vector3.up);
		Quaternion novi_smjer = Quaternion.Lerp(
			rigidbody.rotation,
			zeljeni_smjer,
			15f * Time.deltaTime);
		rigidbody.MoveRotation(novi_smjer);
	}

}
```

Kod 8.1 Konačan izgled klase Igrac

Za razumijevanja programiranja igara potrebno je poznavanje OOP koncepata te načina rada programa u kojem izrađujemo igru. Jedna od miskoncepcija je način izvršavanja programskog koda sadržanog u skriptama. Npr. ostvarivanje blagih pokreta u igri ne možemo ostvariti petljama jer se izvršavanje petlje odvija unutar jednog pokretanja skripte koje traje jako kratko vrijeme. Potrebno je uvesti vremensku dimenziju koja će taj jedan pokreti podijeliti na manje dijelove koji će biti raspoređeni u zadanoj jedinici vremena te ćemo tako dobiti realistične kretnje.

8.2.5. Vježba br_2

Kada radimo s velikim brojem objekata u igri bilo da su različite ili iste klase, potreban nam je sistem označavanja. Označavanjem objekata igre olakšavamo postupak pronalaženja tih istih objekata. Isto tako te objekte iznova dohvaćamo te uvijek trebamo pamtiti i upisivati nazive tih istih oznaka. U prethodnoj vježbi koristili smo se parametrima za animatore koji su bili tipa `string`. Elementima igre, kojima se često koristimo, pridodajemo oznake (eng. *Tags*) kao na slici (Sl. 8.27).



Sl. 8.27 Oznake

Kako bi se lakše koristili parametrima za animatore i oznakama objekata u sceni, sve oznake ćemo tekstualne oznake pospremiti na jedno mjesto gdje ćemo ih iščitavati kada nam je to potrebno. Ovako ćemo umanjiti vjerojatnost pogreški pri pisanju tekstualnih parametara. Za to će nam poslužiti statičke klase. Klase ćemo pospremiti u vlastiti imenski prostor.

Klasa Anim_Oznake neka sadrži tekstualne (`string`) vrijednosti parametara animatora (na sličan način možemo spremiti i hash vrijednosti parametara (kao ID parametra) korištenjem statičke metode `Animator.StringToHash("nazivParametra")`. Klasa Tagovi neka sadrži string vrijednosti oznaka koje su obrađene u poglavlju (8.1.1).

Korak 1: Napraviti novu skriptu pod nazivom Oznake te ju pospremiti u direktorij Vjezba2

Korak 2: Umjesto klase u skriptu napraviti novi imenski prostor naziva namOznake (koristiti ključnu riječ namespace)

Korak 3: Unutar imenskog prostora napraviti dvije statičke klase (Anim_Oznake i Tagovi)

- Prisjetimo se kako statičke klase ne mogu biti instancirane
- Statičke klase mogu sadržati samo statičke elemente i konstante

Korak 4: Klasa Anim_Oznake neka sadrži 3 javna statička stringa (“Speed“, “Sneaking“ i “Open“), a klasa Tagovi konstante tipa string (“Player“, “AlarmLight“, “GameController“, “MainLight“ i “Fader“)

- Statičkim elementima možemo mijenjati vrijednosti tijekom izvođenja programa
- Konstantama, nakon kompajliranja, ne možemo mijenjati vrijednosti
- Za javne elemente, pri deklariranju, koristiti ključnu riječ public

Korak 5: Izrađene klase potrebno je koristiti u ustalim skriptama igre (za sada u Igrac)

- Koristeći ključnu riječ using naglašavamo koji imenski prostor želimo koristiti u skripti (Sl. 8.28)

```
1 using UnityEngine;
2 using System.Collections;
3 using namOznake;
```

Sl. 8.28 Imenski prostori za korištenje

- Koristiti oznake iz statičke klase Anim_Oznake kao parametre u metodama SetBool(…) i SetFloat(…) (Sl. 8.29)

```
anim.SetBool("Sneaking", p);  ➡  anim.SetBool(Anim_Oznake.bool_prikradanje, p);
```

Sl. 8.29 Primjer korištenja statičkih elemenata statičke klase

Primjetimo kako skriptu sa statičkim klasama nismo trebali dodavati u scenu igre na neki od objekata igre. Razlog je jednostavan, to su klase koje se ne mogu instancirati, dovoljno se referencirati na njihove elemente. U programu može postojati proizvoljan broj različitih objekata iste klase (obične klase) s različitim vrijednostima atributa. Ako su atributi statički kao bool_prikradanje iz klase Anim_Oznake (Slika 3.8) onda u programu postoji samo jedna varijabla bool_prikradanje s jednom jedinom vrijednosti koja je promjenjiva.

```
using UnityEngine;
using System.Collections;
namespace namOznake{
      public static class Anim_Oznake{
            public static string float_brzina = "Speed";
```

```
            public static string bool_prikradanje =
                  "Sneaking";
            public static string bool_vrata = "Open";
      }
      public static class Tagovi{
            public const string player = "Player";
            public const string alarm = "AlarmLight";
            public const string gameController =
                  "GameController";
            public const string mainLight = "MainLight";
            public const string fader = "Fader";
      }
}
```

Kod 8.2 Konačan izgled skripte Oznake

Klasa i objekt su glavni koncepti objektno orijentiranog programiranja stoga ih je potrebno razumjeti u samome startu. Nerazumijevanje osnovnih tipova klasa često dovodi do nerazumijevanja koda i pitanja „zašto meni ovo ne radi?“. Kada deklariramo klasu možemo odrediti njenu pristupačnost u različitim dijelovima programa. Kada želimo koristiti klase u različitim dijelovima programa činimo ih javnima (eng. *public*). Ovisno o načinu upotrebe klasa možemo ih deklarirati kao obične (`class`), statičke (`static class`) te apstraktne (`abstract class`). Obične klase opisuju objekte u programu. Statičke se često opisuju kao spremnici za određena polja i metode. Metode statičkih klasa često imaju funkcionalnost koja ne pripada nekom određenom objektu nego je tu kao dodatna funkcionalnost koja može poslužiti u različitim dijelovima programa. Apstraktne klase služe kao podloga za neke druge klase, malo više o tome u poglavlju o nasljeđivanju.

8.3. Interakcija objekata u igri (metode, polja, događaji)

Već smo spominjali kako nam objektno orijentirani pristup omogućuje podjelu programskog koda na manje dijelove. Svaki, manji, dio koda odnosi se na jedan element igre, tj. na njegova svojstva i funkcionalnost koju programer ostvaruje. Sljedeći korak bi bio omogućiti nekakav oblik komunikacije između tih elemenata igre, a to ostvarujemo uz pomoć metoda i polja. Metode igraju najveću ulogu kod tog oblika komunikacije. Kroz prethodno poglavlje osposobljen je naš lik iz igre da se kreće po svijetu, a sada je potrebno omogućiti interakciju lika s okolinom. To možemo napraviti na više načina. Jedan od

načina je opisati svaki element iz okoliša tako da prepoznaje igrača te mu nudi određene opcije interakcije. Za početak se nećemo koristiti grafičkim sučeljem jer je to zasebno poglavlje.

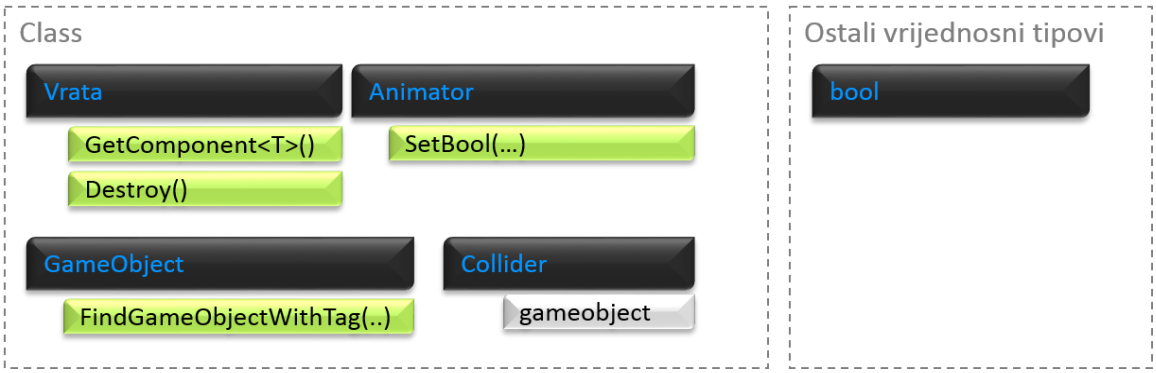
Kroz prethodno poglavlje smo se upoznali s pojmovima klasa, objekt, metoda, atribut i konstruktor. Kroz ovo poglavlje napravljen je naglasak na pojmove metode i atributa. Također, uvodimo dva nova pojma, a to su događaj (eng. event) i okidač (eng. trigger). Događaj možemo shvatiti kao reakciju na neku promjenu u programu, a okidač je ta promjena koja pokreće događaj. U samoj konačnici reakcija je zapravo pokretanje određene metode. Primjer događaje je klik mišem. Kada se dogodi promjena s nije kliknut na kliknut je, pokreće se događaj.

Također, uvesti ćemo pojam veze posjedovanja (eng. has relationship). To je jedan tip veze unutar OO paradigme. Možemo zaključiti kako za bilo kakav tip veze trebamo imati barem dvije klase. Ovime smo se koristili čak i u prethodnom poglavlju. Primjer iz prethodnog poglavlja su klase Igrac i Animator. To su dvije različite klase koje su povezane preko veze posjedovanja, tj. Igrac posjeduje Animator kao jedan od atributa.

8.3.1. Vježba br_3

Ova vježba sastoji se od dva dijela. Prvi dio je izrada klasa Torba i Vrata sa slike (Sl. 8.23), a drugi dio vježbe je izrada klase Kartica. Izrada klase Vrata prikazana je detaljno, dok klasu Kartica treba izraditi po uzoru na klasu Vrata.

Za uspješno obavljanje dijela vježbe br_3 dovoljno se koristiti elementima sa slike Sl. 8.30. Slika (Sl. 8.30) odnosi se samo na izrađivanje klase Vrata.



Sl. 8.30 Podatci potrebni za izvršavanje zadatka

Za početak potrebno je izraditi javnu klasu pod nazivom Torba. Ova klasa će nam simulirati, kao što sam naziv kaže, torbu. Na ovaj način možemo bilo što pospremiti u torbu. Npr. u igri imamo klasu predmet i te predmete želimo spremiti u torbu igrača. Dovoljno je napraviti niz ili listu predmeta (s ograničenjem) te u taj niz ili listu spremati reference na predmete. U našoj igri postoji jedna kartica koja ima jednu ulogu stoga je dovoljno u torbu pospremiti samo boolean vrijednost koja će nam dati informaciju je li igrač pronašao karticu. Klasa torbe i reference na torbu unutar klase Igrac prikazana je na u kodu (Kod 8.3).

Korak 1: Napraviti novu skriptu pod nazivom Torba te ju pospremiti u direktorij Vjezba3

Korak 2: Klasa torba neka sadrži javni boolean atribut naziva imakljuc

Korak 3: Klasa torba neka sadrži jedan konstrukutor bez ulaznih parametara

- Konstruktor bez ulaznih parametara naziva se zadani konstruktor (eng. default constructor)
- Vrijednosti atributa klase pri instanciranju tj. korištenju konstruktora možemo, a i ne moramo postaviti.
- Atribute klase možemo postaviti unutar konstruktora preko parametara ili odmah upisanih vrijednosti kao u kodu (Kod 8.3)

Korak 4: Klasa torba neka sadrži jednu javnu metodu pod nazivom Uzmi_Ili_Baci_Kljuc() koja mijenja vrijednost ključa u suprotnu

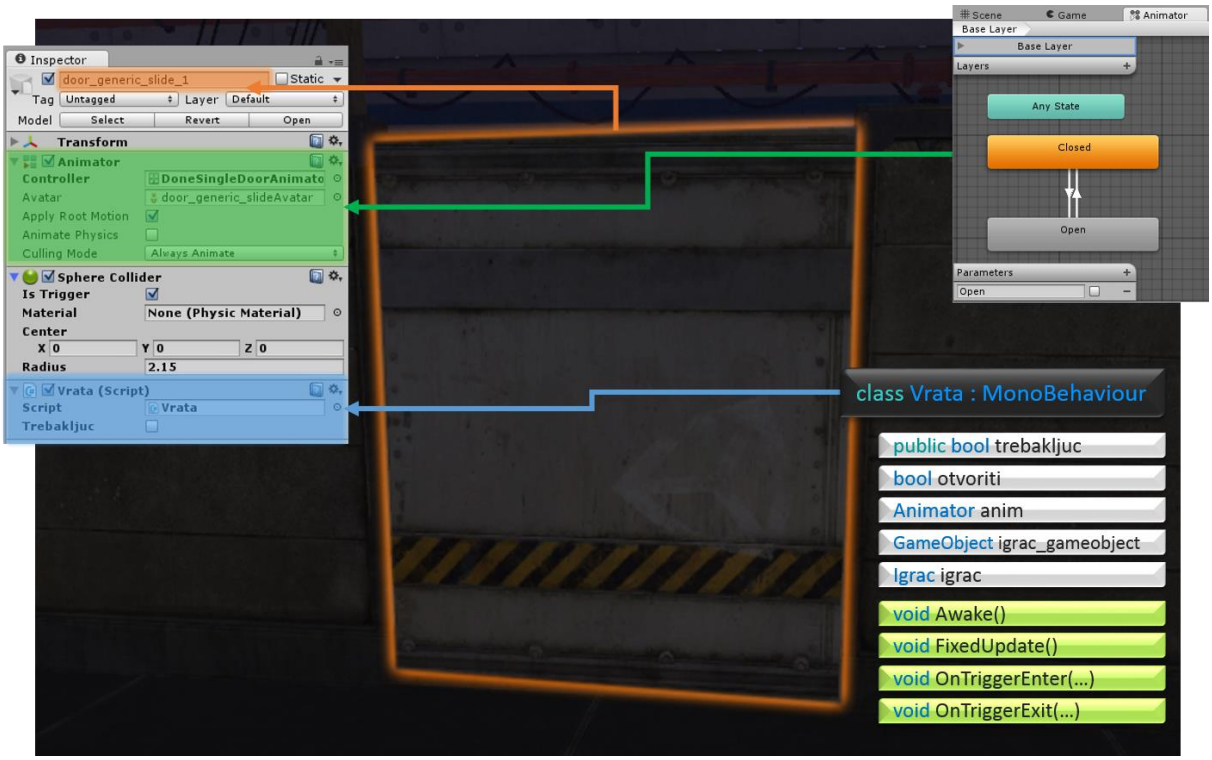
Korak 5: Neka klasu Igrac opisuje atribut klase Torba (veza posjedovanja)

- Veza posjedovanja je zapravo stvaranje novog atributa koji je objekt neke klase

```
using System;
using UnityEngine;
using System.Collections;
public class Torba{
            public bool imakljuc;
            public Torba (){
                  imakljuc = false;
            }
            public void Uzmi_Ili_Baci_Kljuc(){
                  imakljuc = !imakljuc;
            }
```

```
}
public class Igrac : MonoBehaviour {
	Animator anim;
	public Torba mojatorba;
	void Awake () {
		anim = GetComponent<Animator>();
	  mojatorba = new Torba();
	}
	/*...*/
}
```

Kod 8.3 Prikaz klase Torba te veza posjedovanja (Torba u Igrac)



Sl. 8.31 Klasa Vrata i veza s 3D modelom iz igre

Na slici (Sl. 8.31) prikazan je ciljani izgled klase vrata. Također, na slici, možemo vidjeti sve elemente vezane na objekt igre naših vrata. Zelenom bojom označen je animator kontroler sličan kao i kod igrača. Plavom bojom označena je skripta koja sadrži klasu Vrata u koju pišemo programski kod.

Još jedna bitna komponenta sa slike je sferni detektor sudara (eng. *Sphere Collider*). Detektor sudara možemo koristiti na dva načina. Prvi način je za sprječavanje preklapanja dvaju 3D objekata, a drugi kao okidač. U ovom slučaju koristimo ga kao okidač (označiti

„*Is Trigger*“) što znači da ovaj detektor sudara pokreće nekoliko događaja, za nekoliko slučajeva. Nas najviše zanimaju tri slučaja, kada detektor sudara našeg lika uđe, napusti ili stoji preklopljen s detektorom sudara vrata.

Korak 1: Napraviti novu skriptu pod nazivom Vrata te ju pospremiti u direktorij Vjezba3

Korak 2: Implementirati sve atribute i metode sa slike 4.3

- Metode OnTriggerEnter(…) i OnTriggerExit(…) sadrže po jedan ulazni parametar naziva other i klase Collider
  - Metoda OnTriggerEnter(…) pokreće se kada se bilo koji drugi detektor sudara preklopi s detektorom sudara koji je vezan na isti objekt igre kao i skripta u kojoj se nalazi ova metoda
  - Metoda OnTriggerExit(…) Pokreće se kada se bilo koji drugi detektor sudara prestane preklapati s detektorom sudara koji je vezan na isti objekt igre kao i skripta u kojoj se nalazi ova metoda
  - Metoda OnTriggerStay(…) Pokreće se dok se drugi detektor sudara preklapa s detektorom sudara koji je vezan na isti objekt igre kao i skripta u kojoj se nalazi ova metoda
- Varijabla „trebakljuc“ neka bude javna (public). Ovo će nam omogućiti da preko okvira Inspector postavimo taj isti parametar ovisno o tipu vrata. Za tip vrata sa slike (Sl. 8.31) ne treba ključ dok za tip vrata sa slike (Sl. 8.32) treba ključ.

Sl. 8.32 Tip vrata za koja treba ključ

- Varijabla „otvoriti“ nam koristi za kontrolu vrata (otvoriti – zatvoriti) tj. služiti će nam kao parametar „Open“ za animator (pogledati sliku Sl. 8.30). Istu varijablu potrebno je pri kompajliranju postaviti na false tako da vrata na samom početku igre budu zatvorena

Korak 3: Unutar metode Awake() dohvatiti referencu na objekte klasa Animator, GameObject i Igrac. Objekt klase Animator nalazi se na objektu na kojem se nalazi i trenutna skripta dok su ostali objekti vezani za našeg lika

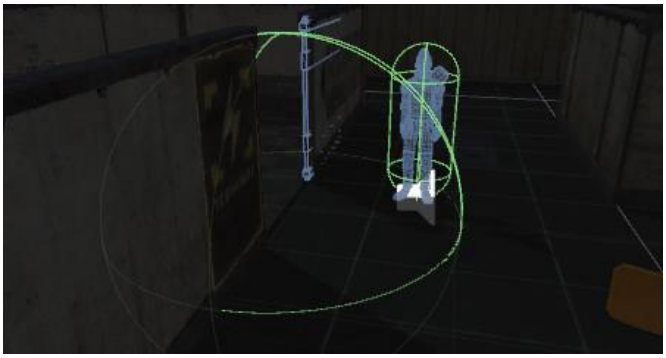
- Prisjetimo se: GetComponent<T>() – generička metoda klase Component. Ovoj metodi možemo pristupati izravno iz vlastite klase jer je ta metoda također već ugrađena
- Klasa GameObject posjeduje statičke metode za pronalazak objekata u igri prema različitim parametrima. Jedna od takvih metoda je FindGameObjectWithTag("naziv_oznake")
  - Prisjetimo se: u vježbi 2 smo napravili statičku klasu Tagovi koja sadrži sve potrebne oznake
- Napomena: kada dohvatimo igračev objekt klase GameObject, svim klasama, uključujući i vlastite, možemo pristupiti preko generičke metode GetComponent<T>(). U ovom slučaju istu metodu, ali iz drugog objekta koristimo za dohvaćanje objekt klase Igrac.

Korak 4: Unutar metode Update() potrebno je postaviti parametar „Open" animatora kojeg smo dohvatili u prethodnom koraku

- Prisjetimo se: metoda SetBool(…) nam služi za postavljanje parametara tipa bool
- Metode Awake() i Update() su zapravo metode pokretane događajima. Okidači koji pokreću ove događaje su vremenski postavljeni

Korak 5: Potrebno je napisati dio koda unutar „OnTrigger" metoda kojim ćemo saznati je li igrač taj koji je došao do vrata. Ako je igrač došao do vrata potrebno je otvoriti vrata

- Jedan način interakcije je kombinacija koraka 4 (dohvati igrača) i koraka 6 (provjeri je li traženi objekt igrač – učini nešto, reakcija na promjenu)
- Collider other – other je objekt tipa Collider (detektor sudara) koji je ujedno i parametar svih „Trigger" metoda. Ovaj objekt sadrži informacije o detektoru sudara koji je u interakciji s detektorom sudara objekta na kojeg je vezana trenutno pisana klasa (u ovom slučaju vrata).
  - Ako igrač dođe ispred vrata, sadržaj objekta „other" biti će detektor sudara igrača (Sl. 8.33).



Sl. 8.33 Detektori sudara

  - Iz objekta „other“, preko njegovih atributa, možemo dobiti informaciju o gameObject-u, a to nam omogućuje najjednostavniju usporedbu s gameObject-om igrača iz koraka 4
- Prisjetimo se: metoda Update() se poziva oko 60ak puta u sekundi, a parametre animatora postavljamo unutar iste metode zbog konstantne kontrole vrata
- Varijablu „otvoriti“ postavljamo ovisno o poziciji igrača u odnosu na vrata
  - Pripaziti na ograničenja: je li igrač detektiran, treba li ključ, ima li igrač ključ

Korak 6: Pokrenuti igru i testirati sva vrata u sceni

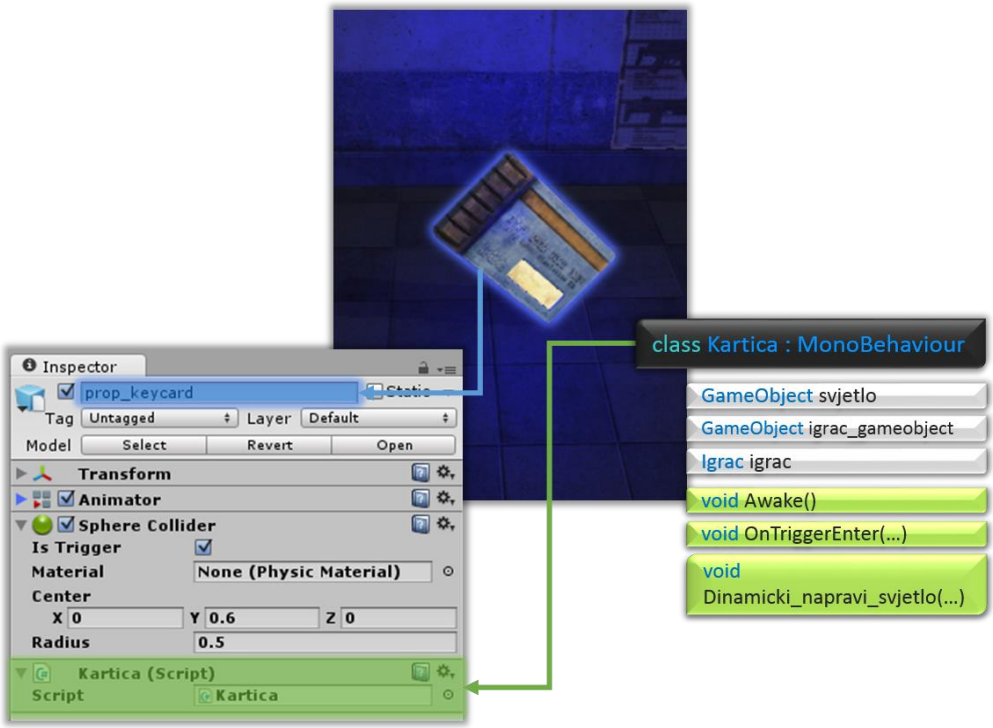
- Ukoliko je postavljanje parametara animatora vršeno na pravome mjestu te ako su implementacije metoda OnTriggerEnter(…) i OnTriggerExit(…) pravilne vrata bi se trebala otvoriti kada se igrač približi te zatvoriti kada se igrač udalji

```
using UnityEngine;
using System.Collections;
using namOznake;

public class Vrata : MonoBehaviour {
      public bool trebakljuc;
      private bool otvoriti = false;
      private Animator anim;
      private GameObject igrac_gameobject;
      private Igrac igrac;

      void Awake () {
            anim = transform.GetComponent<Animator> ();
            igrac_gameobject =
            GameObject.FindGameObjectWithTag(Tagovi.player);
            igrac = igrac_gameobject.GetComponent<Igrac>();
      }
      void Update () {
```

```
            anim.SetBool (Anim_Oznake.bool_vrata, otvoriti);
        }
        void OnTriggerEnter (Collider other){
          if (other.gameObject == igrac_gameobject
              && trebakljuc)
              Dogadjaji.DodajKliker += Kliker_Vrata;
          if (other.gameObject == igrac_gameobject
              && !trebakljuc)
              otvoriti = true;
          else if (other.gameObject == igrac_gameobject
              && igrac.mojatorba.imakljuc)
              otvoriti = true;
          else
              otvoriti = false;
        }
        void OnTriggerExit(Collider other){
          if (other.gameObject == igrac_gameobject
              && trebakljuc)
              Dogadjaji.DodajKliker -= Kliker_Vrata;
          if (other.gameObject == igrac_gameobject)
              otvoriti = false;
        }
}
```

Kod 8.4 Konačan izgled klase Vrata

Na slici (Sl. 8.34) prikazan je ciljani izgled klase Kartica koju je potrebno vezati za 3D model kartice iz scene. Ova klasa je dosta slična prethodnoj stoga je izostavljen dio s detaljnim objašnjenjima iako je ponuđen programski kod (8.5).



Sl. 8.34 Ciljani izgled klase Kartica

Korak 1: Po uzoru na klasu Vrata napraviti klasu Kartica te ju spremiti u direktorij Vjezba3

- Napomena: uzeti u obzir kako naznačiti da je igrač pokupio karticu
- Pomoć: za uništavanje objekta tijekom izvođenja igre koristiti metodu Destroy(…) koja prima objekt igre što želimo uništiti

Korak 2: Kao opcionalni zadatak probajte izraditi svjetlo („light“ objekt) te ga dodati pri samom kompajliranu ili u bilo kojem trenutku igre

- Pomoć: za izradu svjetla koristiti konstruktor Gameobject() koji prima parametar “The Light“
- Pomoć: koristiti već ugrađenu metodu za dodavanje komponente na karticu
- Pomoć: podesiti boju i intenzitet svjetla preko atributa objekta naziva light koji se nalazi u novoizrađenom objektu igre (GameObject)
- Pomoć: podesiti poziciju prema poziciji kartice

```
using UnityEngine;
using System.Collections;
using namOznake;

public class Kartica : MonoBehaviour {
```

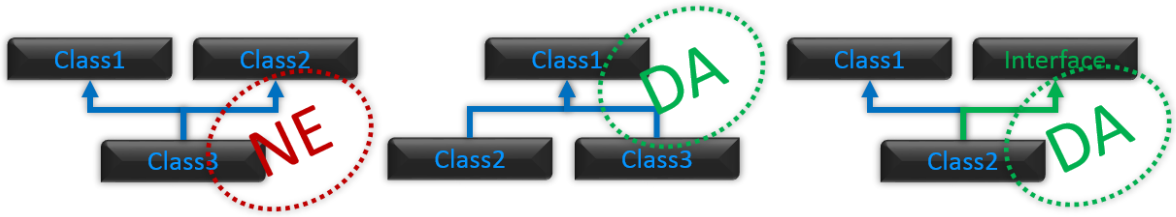
```
    private GameObject igrac_go;
    private Igrac igrac;
    GameObject svjetlo;

    void Awake () {
          igrac_go =
          GameObject.FindGameObjectWithTag (Tagovi.player);
          igrac = igrac_go.GetComponent<Igrac>();
          Dinamicki_napravi_svjetlo();
    }

    void OnTriggerEnter (Collider other){
          if (other.gameObject == igrac_go) {
                igrac.mojatorba.imakljuc = true;
                Destroy(svjetlo);
                Destroy(gameObject);
          }
    }

  private void Dinamicki_napravi_svjetlo(){
      svjetlo = new GameObject("The Light");
      svjetlo.AddComponent<Light>();
      svjetlo.light.color = new Color(0, 0, 1f, 0.01f);
      svjetlo.light.intensity = 0.4f;
      svjetlo.transform.position = transform.position
          + new Vector3(0, 0.5f, 0);
  }
}
```

Kod 8.5 Konačan izgled klase Kartica

8.4. Nasljeđivanje

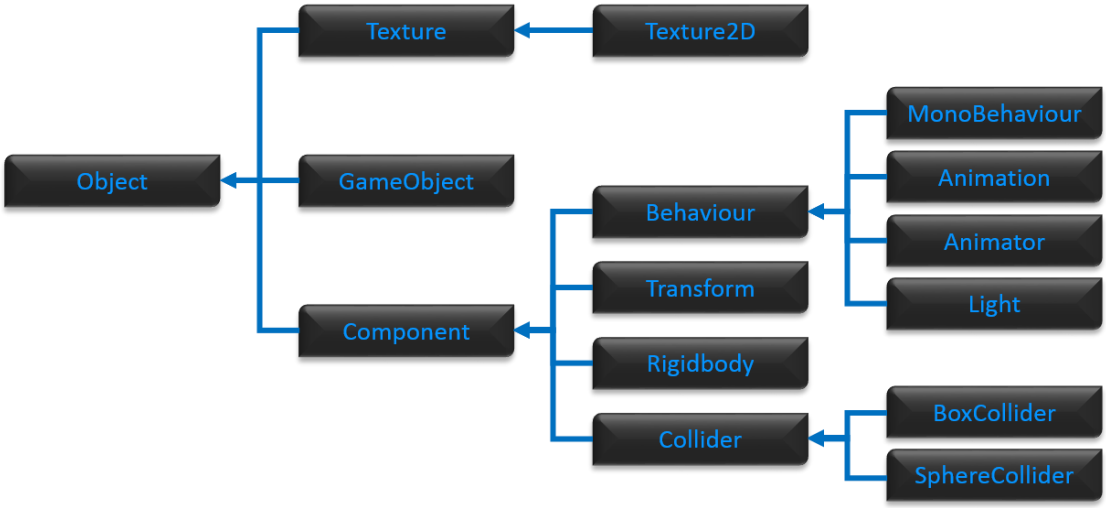
Kroz prethodna poglavlja naučili smo što su to objekt i klasa, od čega se oni sastoje i koja je uloga svakog elementa unutar klase. Također smo se upoznali s različitim vrstama podataka te kako i kada ih koristiti. Iako su svi ti koncepti do određene dubine detalja objašnjeni, postoji još neodgovorenih pitanja. Npr. zašto za neke klase trebamo pisati : MonoBehaviour i koja je svrha toga, kako su to neke metode i parametri već „ugrađeni“. Za sve to postoji jedan jednostavan odgovor, a to je nasljeđivanje (eng. Inheritance). Kada

upišemo „MojaKlasa : MonoBehaviour“ želimo naznačiti kako će naša klasa naslijediti (poprimiti) sva obilježja klase MonoBehavior. Klasa MonoBehaviour je jedna od gotovih klasa Unity biblioteke. Znači, nasljeđivanje nam omogućuje da naša klasa, uz vlastita obilježja, ima obilježja klase MonoBehaviour tj. sva polja (atribute), metode i konstruktore sadržane u istoj. Također, ako je i nad klasom MonoBehaviour izvršeno nasljeđivanje, naša klasa poprima i ta obilježja. Metode Awake(), Start(), Update(), OnTriggerEnter(Collider other) i dr. su metode ugrađene kroz sustav nasljeđivanja stoga ih možemo koristiti ukoliko naglasimo nasljeđivanje nad klasom MonoBehaviour. Važno je napomenuti kako u C# jeziku svaka klasa zasigurno nasljeđuje barem jednu klasu, a to je osnovna klasa naziva Object. C# jezik ne dopušta višestruko nasljeđivanje klasa, iznimka postoji kada istovremeno nasljeđujemo sadržaj jedne klase i proizvoljnog broja sučelja (Sl. 8.35).



Sl. 8.35 Primjeri nasljeđivanja

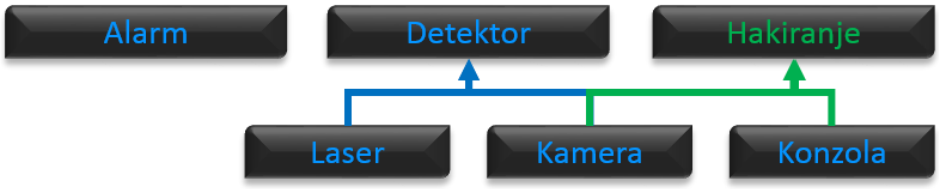
Na slici (Sl. 8.36) prikazan je dio hijerarhije nasljeđivanja unutar Unity biblioteke. Prisjetimo se metoda i atributa koje smo koristili kroz prethodne vježbe. GetComponent<T>() je metoda iz klase Component, atribut rigidbody klase Rigidbody je atribut sadržan u klasi MonoBehaviour, metoda Destroy(…) je zapravo metoda iz klase Object. Primijetimo kako neke metode sežu sve do osnovne klase C# jezika.



Sl. 8.36 Dio hijerarhije klasa Unity biblioteke

Nasljeđivanje je jedan od najbitnijih mehanizama OO paradigme. Nasljeđivanje nam pruža mogućnost smanjenja pisanog programskog koda i bolju organizaciju klasa unutar programa. Kada se koristimo nasljeđivanjem, izbjegavamo dupliciranje programskog koda te kada želimo mijenjati sadržaj zajedničkih elemenata ne trebamo to raditi na više mjesta. Kada izrađujemo dijagram klasa za program trebamo sagledati koja su obilježja zajednička, a koja nisu kako bi postavili dobre temelje za izgradnju programa..

U nastavku ćemo uvesti nova dva pojma, a to su prava pristupa i polimorfizam. Za svaki pojam, u kombinaciji s nasljeđivanjem postoji pripadajuća vježba.



Sl. 8.37 Ciljana hijerarhija klasa za vježbe 4 i 5

Promotrimo sliku (Sl. 8.37) i veze između klasa i sučelja (eng. *interface*). Kamera i laser imaju dosta sličnih obilježja, a to su detektiranje igrača te aktiviranje alarma stoga ćemo napraviti zajedničku klasu Detektor koja će se moći primijeniti na različita dva 3D objekta u igri. Također kamera i konzola imaju neka slična svojstva kao npr. *hakiranje* uređaja za prekidanje alarma. Pošto nam ovaj programski jezik ne dopušta višestruko nasljeđivanje klasa koristiti ćemo se sučeljem. Sučelje možemo zamisliti kao nekakav spremnik za funkcionalnosti koje želimo pridijeliti različitim klasama ili strukturama. Sučelje sadrži samo definicije metoda ili svojstva (eng. *property*). Ukoliko pokušamo izraditi neku

metodu s kodom unutar iste, program će nam javljati grešku. Klasa Alarm nam nije povezana na ostale klase uz pomoć nasljeđivanja nego samo kao sadržaj drugih klasa. Veza Detektor – Laser je veza nasljeđivanja (eng. *is relationship*) dok za vezu Detektor – Alarm, pošto detektor upravlja alarmom, veza posjedovanja (eng. *has relationship*). Time želimo reći kako je laser ujedno i detektor dok alarm nije nikako detektor nego detektor posjeduje referencu na alarm kako bi upravljao istim.

8.4.1. Prava pristupa

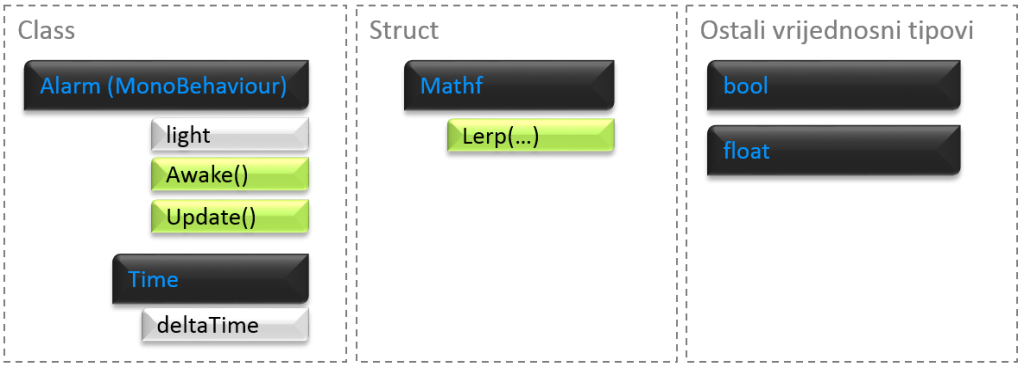
Sva prava pristupa elementima klasa nisu direktno vezana za pojam nasljeđivanja. Razumijevanje nasljeđivanja nam pomaže pri shvaćanju prava pristupa. Neka od prava pristupa su javna (eng. *public*), privatna (eng. *private*) i zaštićena (eng. *protected*). Javnim elementima neke klase možemo pristupati iz bilo kojeg dijela programa. Privatnim elementima neke klase možemo pristupiti samo unutar klase u kojoj su deklarirani dok zaštićenim elementima možemo pristupiti samo unutar izvorne i svih izvedenih klasa (tamo gdje smo se koristili nasljeđivanjem). Ukoliko, pri deklaraciji, ne upišemo ključnu riječ za pravo pristupa element automatski postaje privatan.

Ako pogledamo sliku (Sl. 8.37) i zamislimo da klasa Detektor posjeduje barem tri elementa sa svim različitim pravima pristupa onda:

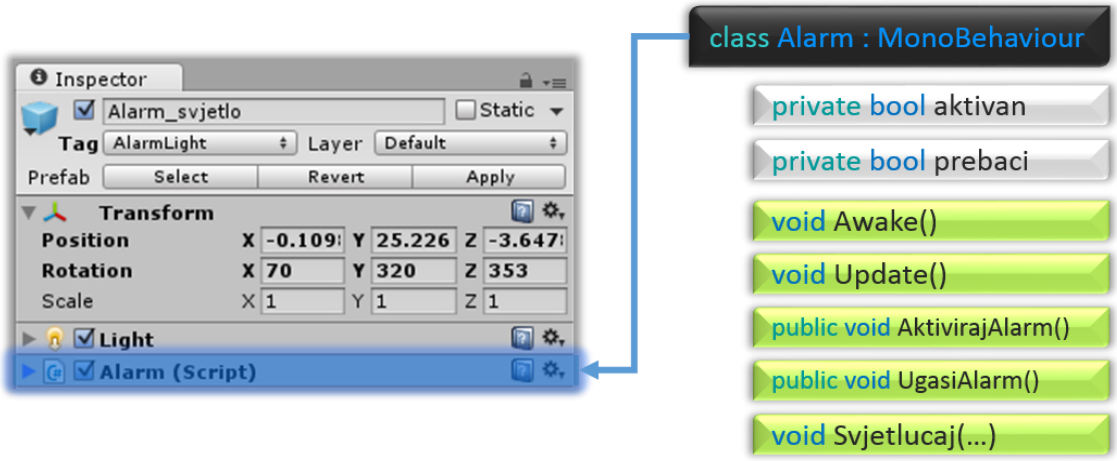
- Javnim elementima možemo pristupati iz svih ostalih klasa
- Privatnim elementima možemo pristupati samo unutar klase Detektor
- Zaštićenim elementima možemo pristupati unutar klasa Detektor, Laser i Kamera

8.4.2. Vjezba br_4

Na slici 6.3 uz objašnjenja smo već promotrili odnos klasa Kamera, Laser, Konzola, Alarm i Detektor. Za ovu vježbu prvo ćemo početi s izradom klasa Alarm i Detektor. Klasa alarm nam služi za kontrolu svjetla iz igre naziva „Alarm_svjetlo“ koje ćemo uključivati i isključivati ovisno o tome je li igrač detektiran (Sl. 8.39). Za izvršavanje vježbe dovoljno se koristiti elementima sa slike (Sl. 8.38).



Sl. 8.38 Podatci potrebni za izvršavanje zadatka



Sl. 8.39 Ciljani izgled klase Alarm

Korak 1: Napraviti novu skriptu pod nazivom Alarm te ju spremiti u direktorij Vjezba4

Korak 2: Implementirati sve atribute i metode sa slike (Sl. 8.39)

- Metode Awake() i Update() su metode nasljeđene od klase MonoBehaviour
- Metode AktivirajAlarm() i UgasiAlarm() trebaju biti javne kako bi im pristupali iz drugih klasa kao npr. klase Detektor. Detektor treba paliti/gasiti alarm prego ovih javnih metoda.

Korak 3: Metoda Awake() treba resetirati intenzitet svjetla alarma na 0 te postaviti aktivnost alarma na false

- Objektu svjetla (light) pristupamo izravno iz vlastite klase jer je to jedan od atributa klase MonoBehaviour (MonoBehaviour sadrži reference na objekt igre i neke od njegovih komponenti, ovo se odnosi na objekt igre na kojeg je vezana i skripta s klasom koja nasljeđuje klasu MonoBehaviour)
- Objekt light posjeduje atribut za intenzitet svjetla koji je podesiv tj. nije samo za čitanje (eng. read only)

- Upozorenje: light je jedan od objekata vezan na sve objekte klase GameObject, za uspješan rad sa svjetlom trebamo prvo provjeriti je li svjetlo vezano za naš objekt igre (Sl. 8.39) jer će nam u protivnom light objekt biti prazan tj. null

Korak 4: Metoda Update() treba, ako je alarm aktivan, neprestano pozivati metodu kojom dobijamo funkcionalnost svjetlucanja, a u protivnom vraćati intenzitet svjetla na 0

- Prisjetimo se: Update() metoda se poziva neposredno prije stvaranja svake slike (frame) što je oko 60-ak puta u sekundi.

Korak 5: Metode za aktivaciju i deaktivaciju alarma trebaju mijenjati vrijednost varijable aktivan

Korak 6: Metoda Svjetlucaj() kao Awake() i Update() trebaju biti privatne jer ne želimo vanjsku kontrolu nad ovom metodom

- Prisjetimo se: privatne metode su one koje su deklarirane bez ključne riječi za pravo pristupa ili kada su deklarirane uz ključnu riječ private
- Svjetlucanje ćemo ostvariti na sličan način kao i okretanje lika iz vježbe br_1, statičkom metodom Lerp(...) klase Mathf.
- Lerp() metoda se nalazi u više struktura kao npr. Mathf i Quaternion
- Ideja je, za svaki frame, pozivati metodu Svjetlucaj() kako bi postepeno mijenjali intenzitet svjetla s vrijednosti 0.5f na 2f i obrnuto. Lerp() metoda će nam omogućiti da u određenoj jedinici vremena (npr. 2f*Time.deltaTime) prelazimo iz trenutne vrijednosti intenziteta u krajnju vrijednost. Kada postignemo krajnju vrijednost 0.5f ili 2f trebamo promijeniti ciljanu krajnju vrijednost. Npr. prvo ćemo ići od trenutna=0.5f do krajna=2f, a zatim od trenutna=2f do krajna=0.5f.

Korak 7: Provjeriti ispravnost klase Alarm

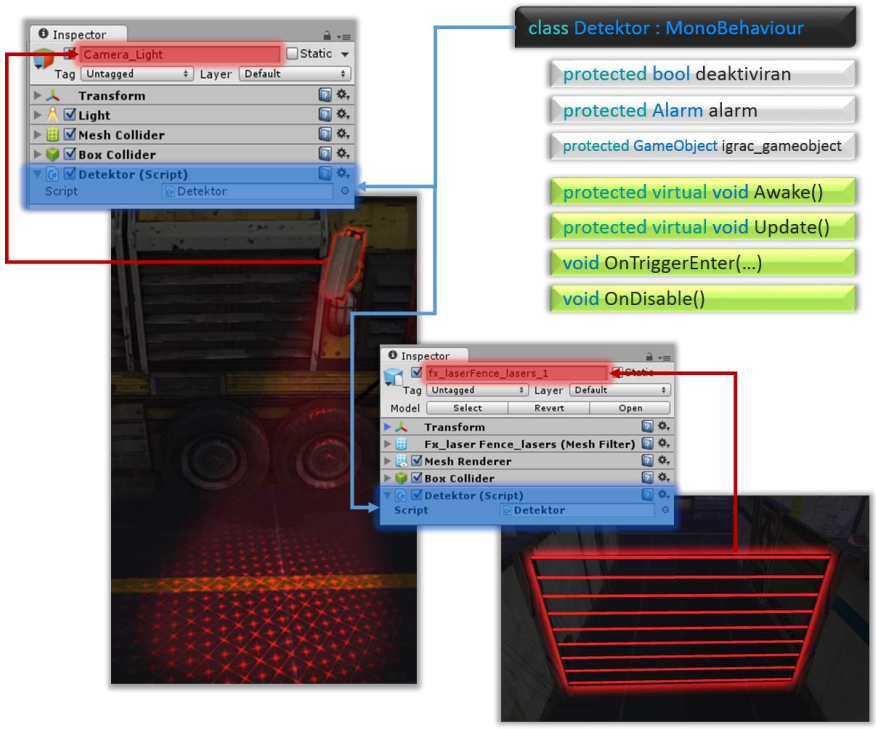
- Ispravnost možemo provjeriti tako da metodu svjetlucaj pozivamo bez ikakvih uvjeta unutar metode Update()

```
using UnityEngine;
using System.Collections;

public class Alarm : MonoBehaviour
{
    private bool aktivan;
    private bool prebaci;
```

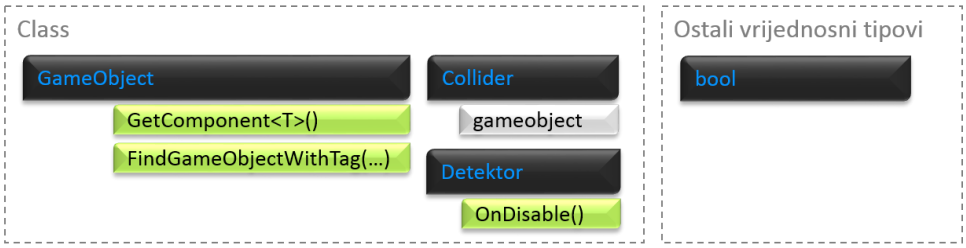
```
    void Awake(){
        light.intensity = 0f;
        aktivan = false;
    }
    void Update(){
        if (aktivan) Svjetlucaj();
        else light.intensity = 0f;
    }
    public void AktivirajAlarm(){
        if (!aktivan){
            aktivan = true;
            Kontrola_igre.PovecajBrzinuBrojaca();
        }
    }
    public void UgasiAlarm(){
        if (aktivan){
            aktivan = false;
            Kontrola_igre.ResetirajBrzinuBrojaca();
        }
    }
    void Svjetlucaj(){
        if (prebaci)
            light.intensity = Mathf.Lerp(
                  light.intensity,
                  2f,
                  2f * Time.deltaTime);
        else
            light.intensity = Mathf.Lerp(
                  light.intensity,
                  0.5f,
                  2f * Time.deltaTime);
        if (light.intensity >= 1.95f
            || light.intensity <= 0.55f)
            prebaci = !prebaci;
    }
}
```

Kod 8.6 Konačan izgled klase Alarm

U nastavku ćemo krenuti s izrađivanjem klase Detektor. Već smo ustanovili kako laseri i kamere imaju neka zajednička obilježja koja ćemo opisati ovom klasom. Skriptu je potrebno vezati za sve objekte igre tj. sve kamere i lasere kao na slici (Sl. 8.40). Na slici (Sl. 8.41) je prikazan minimalan broj elemenata potreban za izvršavanje zadatka.



Sl. 8.40 Ciljani izgled klase Detektor



Sl. 8.41 Podatci potrebni za izvršavanje zadatka

Korak 1: Napraviti novu skriptu naziva Detektor te ju spremiti u direktorij Vjezba4

Korak 2: Implementirati sva polja i metode sa slike (Sl. 8.40)

Korak 3: Polja trebaju biti pristupačna samo u izvedenim klasama i trenutnoj klasi

- Prisjetimo se: zaštićeni (protected) elementi su pristupačni u izvornoj i svim izvedenim klasama

Korak 4: Metode Awake() i Update() će imati različitu implementaciju nakon nasljeđivanja stoga je potrebno omogućiti pristup istima samo još u izvedenim klasama

- Prisjetimo se: zaštićeni (protected) elementi su pristupačni u izvornoj i svim izvedenim klasama

Korak 5: Metoda Awake() neka dohvaća objekte klase Alarm i GameObject (objekt klase GameObject je cijelokupni objekt igre igrača). Također neka ista postavlja varijablu detektiran na laž

- Kao i u prethodno koristimo se GetComponent<T>() i FindObjectWithTag(...) metodama
- Prisjetimo se: metoda GetComponent<T>() je metoda iz klase Component, a klase GameObject i MonoBehaviour ju nasljeđuju stoga ju možemo koristiti unutar tih dviju klasa
- Prisjetimo se: metoda FindObjectWithTag(...) je statička metoda klase GameObject te joj pristupamo izravno preko naziva klase npr. GameObject. FindObjectWithTag(Tagovi.player)

Korak 6: Metoda Update() neka aktivira i deaktivira alarm ovisno o varijabli detektiran i aktivnosti objekta igre „gameobject“ trenutne klase

- Pomoć: potrebno je postaviti uvjete koji će ovisno o detektiranosti igrača paliti alarm preko javne metode AktivirajAlarm() iz klase Alarm (koristimo se objektom alarm u kojeg smo spremili referencu na objekt koji upravlja intenzitetom svjetla)
- Pošto detektor nema izravnu kontrolu nad gašenjem alarma, unutar metode Update() potrebno je samo paliti alarm. Konstantna provjera detektiranosti je potrebna zato što imamo više detektora u igri, ako samo jednim detektorom uključimo alarm, a drugi detektor ga isključi dok je još prvi aktivan imamo logičku pogrešku u našem kodu

Korak 7: Metoda OnTriggerEnter(...) neka postavlja varijablu detektiran postavlja na istinu samo u slučaju detektiranja igrača preko detektora sudara

- Detektore sudara služimo samo za početak preklapanja jer nas ne zanima situacija kada igrač napusti detektor jer ta situacija u igri ne isključuje alarm.

Korak 8: Metoda OnDisable() neka gasi alarm i postavlja varijablu detektiran na laž

- Detektor nema izravnu kontrolu nad isključivanjem alarma, ta funkcionalnost pripada konzoli i kameri, ali je detektor na neki način uključen u proces isključivanja. Ideja je da igrač može interakcijom s konzolom ili kamerom isključiti detektor, a kada je detektor isključen prekidamo kontakt s alarmom (UgasiAlarm()).

- Metoda OnDisable() je nasljeđena od klase MonoBehaviour te se poziva na okidač neposedno prije deaktivacije objekta igre na koji je trenutno vezana skripta u koju pišemo kod

Korak 9: Provjeriti ispravnost klase Detektor

- Ukoliko je cijela klasa ispravno implementirana alarm bi se trebao paliti prolaskom kroz lasere ili svjetlo kamere

```
using UnityEngine;
using System.Collections;
using namOznake;

public class Detektor : MonoBehaviour {

      protected bool detektiran;
      protected Alarm alarm;
      protected GameObject igrac_gameobject;

      protected virtual void Awake () {
        alarm = GameObject.FindGameObjectWithTag(
            Tagovi.alarm).GetComponent<Alarm>();
        igrac_gameobject = GameObject.FindGameObjectWithTag(
            Tagovi.player);
        detektiran = false;
      }

    protected virtual void Update() {
        if (detektiran)
                  alarm.AktivirajAlarm();
        else if (!gameObject.activeSelf)
                  alarm.UgasiAlarm();
      }
    void OnTriggerEnter(Collider other){
        if (other.gameObject == igrac_gameobject
                  && gameObject.activeSelf)
                  detektiran = true;
    }
    void OnDisable(){
        alarm.UgasiAlarm();
        detektiran = false;
```

```
    }
}
```

Kod 8.7 Konačan izgled klase Detektor

8.5. Polimorfizam i učahurivanje

Polimorfizam nam omogućava da postojeće metode i svojstva izvorne klase napravimo, pod istim nazivom, i u izvedenoj klasi. Postoje dva načina izrade „istih“ metoda i svojstava, prvi je premošćivanje (`override`), a drugi način je korištenjem ključne riječi `new`.

Premošćivanje: u izvornoj klasi trebamo metodu ili svojstvo deklarirati kao virtualnu (`virtual`) metodu, a pri deklariranju iste u izvedenoj klasi koristimo ključnu riječ override.Korištenjem ključne riječi `new` deklarira se metoda u izvedenoj klasi kao u primjeru.

Za pristup metodama, ovisno o klasi, koristimo ključne riječi base i `this`. Koristeći `base.Metoda()` se pozivamo na metodu iz osnovne klase dok korištenjem `this.Metoda()` se pozivamo na metodu iz trenutne klase u kojoj pišemo kod. Da bi se mogli koristiti s obadvije implementacije iste metode, trebamo ih deklarirati kao zaštićene (`protected`).

```
class Osnovna : MonoBehaviour{
      protected virtual void Awake(){}
      protected void Update(){}
      private void Start(){}
}
class Izvedena : Osnovna{
      protected override void Awake(){}
      protected new void Update(){}
      protected void Start(){}
}
```

Kod 8.8 Primjer prava pristupa

Instanciranje:

- `Osnovna objekt1 = new Izvedena();` //instanciranje 1. način
- `Izvedena objekt2 = new Izvedena();` //instanciranje 2. način

- `Izvedena objekt3 = new Osnovna();` //nije dopušteno

Korištenje metoda:

- `objekt1.Awake();` //pozivamo metodu iz izvedene klase
- `objekt1.Update();` //pozivamo metodu iz osnovne klase
- `objekt1.Start();` //pozivamo metodu iz izvedene klase
- `objekt2.Awake();` //pozivamo metodu iz izvedene klase
- `objekt2.Update();` //pozivamo metodu iz izvedene klase
- `objekt2.Start();` //pozivamo metodu iz izvedene klase

Razlikovanje metoda unutar izvedene klase:

- `base.Awake();` //metoda osnovne klase
- `this.Awake();` //metoda izvedene (trenutne) klase
- base.Start(); //nije valjan izraz jer je to privatna metoda

Učahurivanje (eng. *Encapsulation*) je također jedan od koncepata OOP-a. Učahurivanje koristimo kada imamo privatne atribute u klasi, ali ipak želimo pristupati istima uz određena ograničenja. Učahurivanje vršimo preko svojstava. Svojstva su poseban oblik polja koja sadrže dvije posebne metode, `get` i `set`. Metoda `get` dohvaća vrijednost privatnog elementa u izvornom ili izmijenjenom obliku (npr. pretvoriti sate u minute). Metoda set postavlja vrijednost privatnog elementa bez ili s uvjetima. Ključna riječ `value` zapravo predstavlja vrijednost koju šaljemo preko svojstva npr. `UdaljenostM = 2000`, u ovom slučaju je `value = 2000`. Korištenje svojstva je nepotrebno ukoliko `get` metodom dohvaćamo izvorni oblik privatnog polja i kada set metodom postavljamo vrijednost privatnog polja bez uvjeta. Razlog tomu je što bez ikakvih uvjeta možemo privatno polje napraviti javnim i dobiti isti rezultat.

```
class Udaljenost{
      private int udaljenostKm;
      public int UdaljenostM{
            get{
                  return udaljenostKm*1000;
            }
            set{
                  if (value > 0 && value < udaljenostKm*1000)
                        udaljenostKm = value/1000;
```

```
                }
            }
        }
```

Kod 8.9 Primjer get i set metoda

8.5.1. Vježba br_5

Na samom početku vježbi smo dobili gotove elemente igre za programirati, a sada je jedan od programera dostavio dvije skripte. Jedna skripta sadrži sučelje dok druga sadži izrađenu klasu Kamera. Zadatak je prvo proučiti što je taj programer napravio te po uzoru na njegovu skriptu izraditi vlastitu klasu Konzola koja nasljeđuje elemente sučelja Hakiranje te klasu Laser koja nasljeđuje elemente klase Detektor.

Korak 1: Promotrimo sučelje u kodu (Kod 8.10)

- Za deklariranje sučelja koristimo ključnu riječ interface
- Sučelje može sadržati samo definicije metoda i svojstava
- Svi elementi sučelja su uvijek javni, dok su elementi klasa automatski privatni. Pri implementaciji elemenata sučelja u klasu trebamo elemente deklarirati kao javne.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Sucelja{
    interface Hakiranje{
        float hakirajNa {get; set;}
        float Brojac { get; set; }
        void Hakiraj();
    }
}
```

Kod 8.10 Sučelje

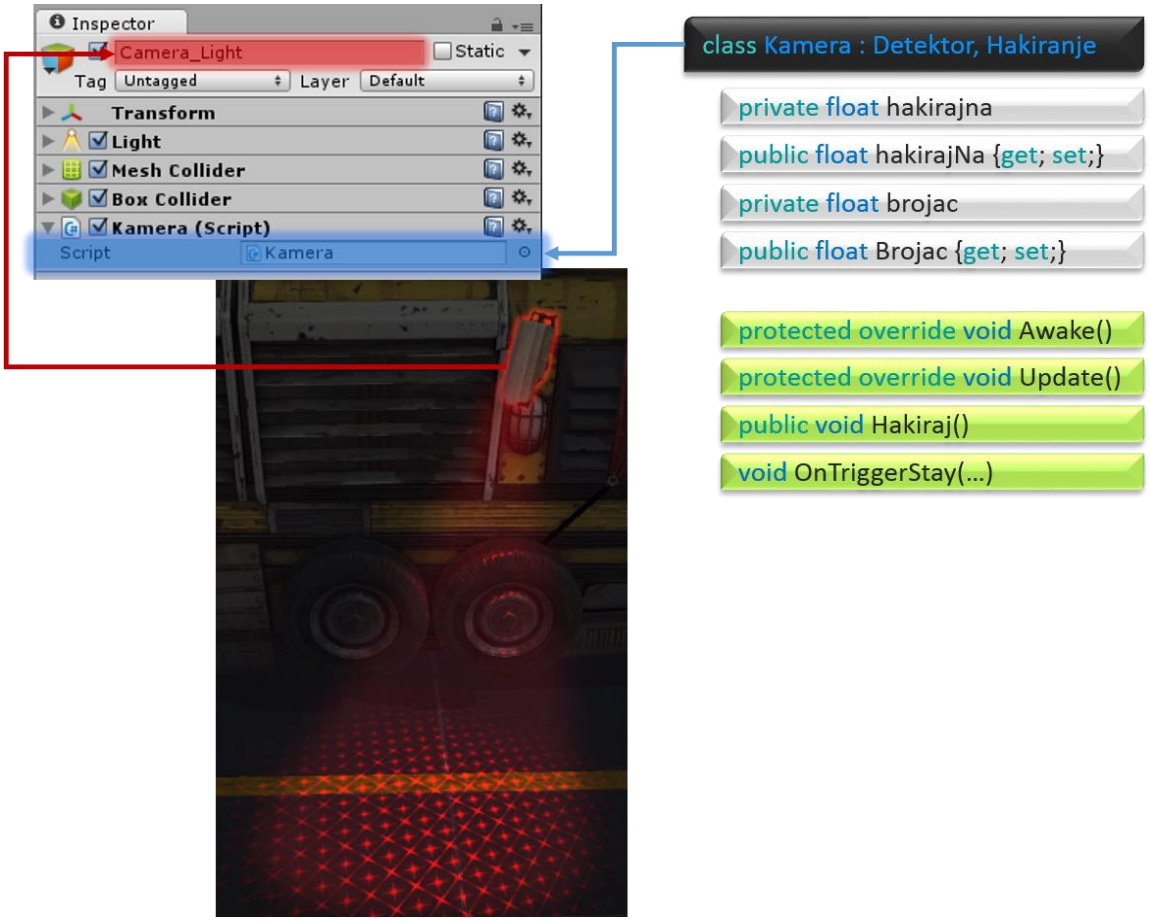
Korak 2: Promotrimo sliku (Sl. 8.42)

- Ukoliko jedan od elemenata sučelja pri implementaciji izostavimo program javlja grešku već pri kompajliranju

Sl. 8.42 Greška ako ne implementiramo element sučelja

Korak 3: Promotrimo sliku (Sl. 8.43)

- Kada imamo izrađenu klasu Kamera koja je naslijedila sva obilježja klase Detektor, možemo te dvije klase zamijeniti na elementu igre pod nazivom Camera_Light
- Klasa Kamera će se dijelom ponašati kao i klasa Detektor, a dijelom će imati novu, vlastitu, funkcionalnost koja ne pripada klasi Detektor
- Također, klasa Kamera nasljeđuje i elemente sučelja Hakiranje. Te iste elemente će nasljeđivati i klasa Konzola jer im je to zajednička funkcionalnost. Implementacije elemenata sučelja Hakiranje će biti slične, ali ipak različite za pojedinu klasu
- Metode Awake() i Update() premoštavamo ključnom riječi override tako da se koristimo novim implementacijama istih metoda
- Uz pomoć metoda Hakiraj() i OnTriggerStay(...) ćemo kontrolirati unose s tipkovnice i gašenje detektora



Sl. 8.43 Ciljani izgled klase Kamera

Korak 4: Promotrimo sliku 5.13

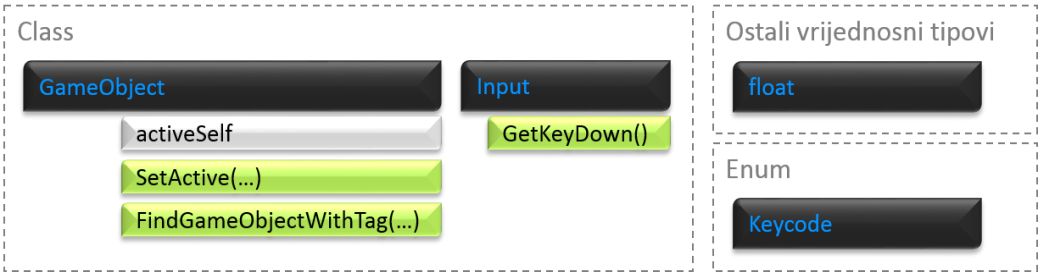
- Primjetimo način na koji su implementirana sučelja te kako kontroliramo vrijednost učahurenih varijabli
- Premoštenim metodama ne smijemo mijenjati pravo pristupa
- Primjetimo kako i dalje želimo funkcionalnost metoda Awake() i Update() iz osnovne klase te način na koji ih pozivamo (base.Awake() i base.Update()). Na ovaj način se prvo izvršava programski kod metoda osnovne klase, a zatim izvedene.
- Statička metoda Input.GetKey(Keycode.E) nam daje istinu dok je god tražena tipka pritisnuta. Ako je igrač pokraj kamere i ako je pritisnuta tipka E onda želimo hakirati kameru, a ukoliko uvjeti nisu ispunjeni potrebno je resetirati brojač na 0.
- Metodom Hakiraj() kontroliramo vrijednost brojača te aktivnost kamere. Ako korisimo samo += Time.deltaTime, dobijemo realno vrijeme. Objekt igre deaktiviramo tako da preko metode SetActive(...) postavimo vrijednost istine ili laži. Metoda SetActive(...) nalazi se unutar klase GameObject stoga joj možemo pristupiti preko atributa gameobject. Taj atribut je nasljeđen od klase MonoBehaviour pa mu možemo pristupiti izravno iz svih izvedenih klasa. Isto tako jedan od atributa klase GameObject je activeSelf, on nam pokazuje je li objekt aktiviran ili deaktiviran.
  - Prisjetimo se: kada se detektor isključi, isključi se i alarm. Kako je ovo izvedena klasa iz klase Detektor i kamera će pri deaktivaciji isključiti alarm.

```
using UnityEngine;
using System.Collections;
using Sucelja;

public class Kamera : Detektor, Hakiranje {
    private float hakirajna;
    public float hakirajNa{
        get { return hakirajna; }
        set {
            if (value >= 1f) hakirajna = value;
            else hakirajna = 1f;
        }
    }
    private float brojac;
    public float Brojac{
        get { return brojac; }
```

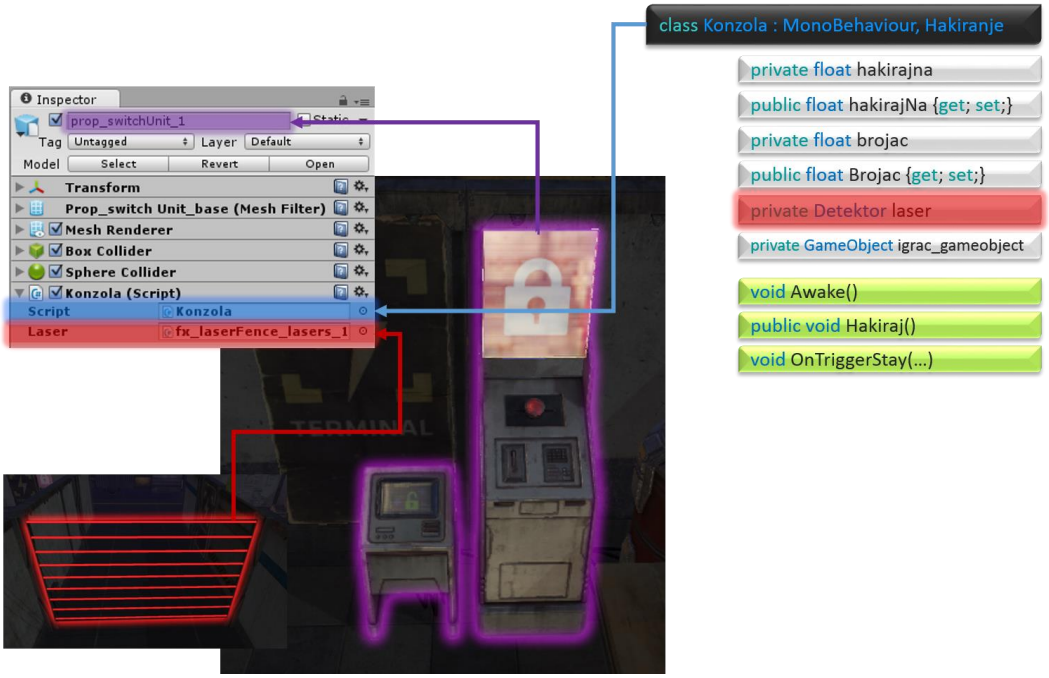
```
            set {
                if (value < 0) brojac = 0;
                else brojac = value;
            }
        }
        protected override void Awake(){
            base.Awake();
            hakirajNa = 4f;
            Brojac = 0f;
        }
        protected override void Update(){
            base.Update();
            Debug.Log(brojac + "/" + hakirajNa);
        }
        public void Hakiraj(){
            Brojac += Time.deltaTime;
            if (brojac >= hakirajNa){
                Brojac = 0;
                gameObject.SetActive(!gameObject.activeSelf);
                base.detektiran = false;
            }
        }
        void OnTriggerStay(Collider other){
            if (Input.GetKey(KeyCode.E)
                && other.gameObject == igrac_gameobject)
            {
                Hakiraj();
            }
            else Brojac = 0f;
        }
    }
```

Kod 8.11 Konačan izgled klase Kamera

Za sljedeći zadatak potrebno je, po uzoru na prethodni zadatak izraditi klase Konzola i Laser koje dijelom izrađuju na sličan način kao i klasa Kamera. Minimalan broj elemenata potrebnih za izradu klase Konzola prikazan je na slici (Sl. 8.44). Ciljani izgled klase Konzola prikazan je na slici (Sl. 8.45). Također, klasa Konzola posjeduje javni atribut klase Detektor. Cilj je kontrolirati elemente u igri s klasom Detektor. U okviru „*Inspector*“ potrebno je dodati elemente iz scene koje želimo kontrolirati.



Sl. 8.44 Podatci potrebni za izvršavanje zadatka



Sl. 8.45 Ciljani izgled klase Konzola

Korak 1: Napraviti novu skriptu naziva Konzola te ju spremiti u direktorij Vjezba5

Korak 2: Klasa Konzola neka nasljeđuje sve elemente sučelja Hakiranje

Korak 3: Implementirati sve elemente sučelja hakiranje po uzoru na klasu Kamera

- Ograničenja: hakirajna ne smije biti različit od 1f
- Ograničenja: brojac ne smije biti manji od 0
- Metoda Hakiranje() neka izravno prebacuje brojač s 0 na 1f
- Metoda Hakrianje(), za razliku one u klasi Kamera, neka isključuje laser kojeg smo dodali kao na slici 5.15. Za uključivanje i isključivanje objekata koristimo se metodom SetActive(...) koja je sadržana u klasi GameObject. Objektu igre lasera pristupamo preko njegovog atributa naziva gameobject kojeg je nasljedio od klase MonoDevelop (laser.gameobject).

- Napomena: ova klasa ne nasljeđuje Detektor stoga ne možemo pristupati elementima klase Detektor bez instance te iste klase

Korak 4: Unutar metode Awake() dohvatiti objekt igre našeg lika (Tagovi.player) te postaviti vrijednosti varijabli hakirajNa i Brojac preko svojstva

- Preko svojstva trebamo postaviti hakirajna = 1f i brojac = 0f ako je unos (value) krive vrijednosti
- Objekt igre našeg lika dohvaćamo preko statičke metode FindGameObjectWithTag(...) koja je dio klase GameObject

Korak 5: Implementirati metodu OnTriggerStay(Collider other) tako da se hakira konzola kada je igrač unutar detektora sudara i kada je pritisnuta tipa E (razlikovati pritiskanje tipke i držanje tipke)

- Input.GetKey(...) nam vraća traženu tipku kada dok je držimo pritisnutom
- Input.GetKeyDown(...) nam vraća traženu tipku kada se tipka pritisne (klik tipke prema dolje)

Korak 6: Testirati ispravnost konzole

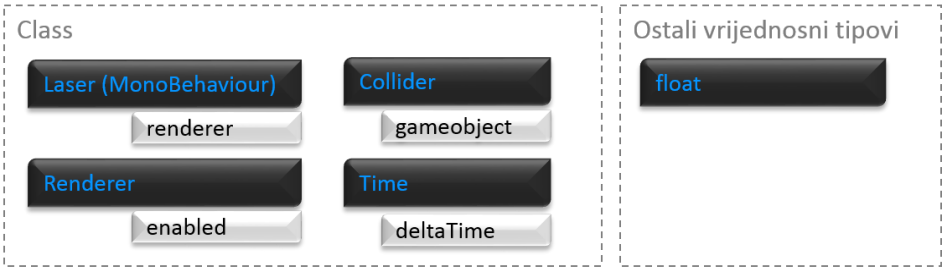
- Ukoliko je klasa pravilno implementirana klikom na tipku E bi se trebao isključiti laser

```
using UnityEngine;
using System.Collections;
using Sucelja;
using namOznake;

public class Konzola : MonoBehaviour, Hakiranje {
    private float hakirajna;
    public float hakirajNa{
        get { return hakirajna; }
        set{
            if (value != 1f) hakirajna = 1f;
        }
    }
    private float brojac;
    public float Brojac{
        get { return brojac; }
        set{
            if (value < 0) brojac = 0;
```

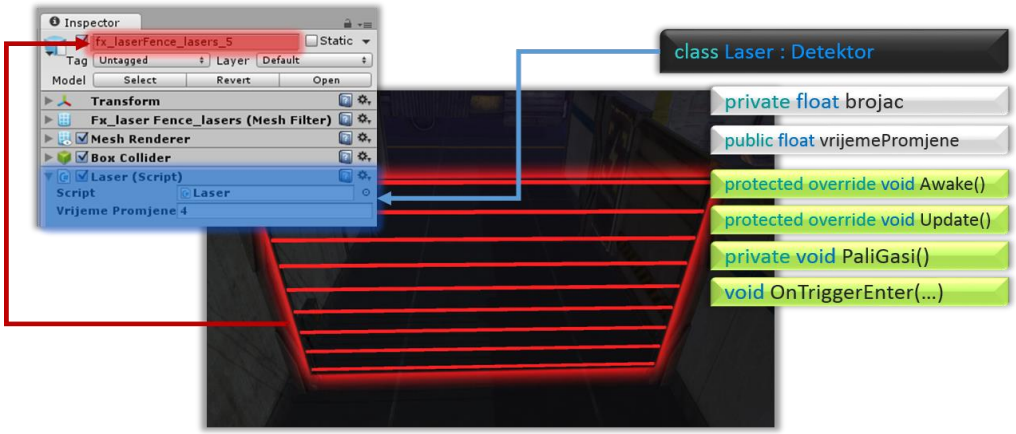
```
            else brojac = value;
        }
    }
    public Detektor laser;
    private GameObject igrac_gameobject;
    void Awake(){
        hakirajNa = 1f;
        Brojac = 0f;
        igrac_gameobject = GameObject.FindGameObjectWithTag(
            Tagovi.player);
    }
    public void Hakiraj(){
        Brojac = 1f;
        if (brojac >= hakirajNa){
            Brojac = 0;
            laser.gameObject.SetActive(
                  !laser.gameObject.activeSelf);
        }
    }
    void OnTriggerStay(Collider other){
        if (Input.GetKeyDown(KeyCode.E)
            && (other.gameObject == igrac_gameobject))
        {
            Hakiraj();
        }
        else Brojac = 0f;
    }
}
```

Kod 8.12 Konačan izgled klase Konzola

Kao što je već rečeno, potrebno je izraditi i klasu Laser. Za uspješnu izradu klase Laser dovoljno se koristiti elementima sa slike (Sl. 8.46). Ciljani izgled klase prikazan je na slici (Sl. 8.47). Važno je za napomenuti kako klasu Detektor ostavljamo na svim laserima za koje želimo da se ne pale-gase dok novoizrađenu klasu postavljamo na lasere koji imaju dodatne funkcionalnosti.



Sl. 8.46 Podatci potrebni za izvršavanje zadatka



Sl. 8.47 Ciljani izgled klase Laser

Korak 1: Napraviti novu skriptu naziva Laser te ju spremiti u direktorij Vjezba5

Korak 2: Klasa Laser neka nasljeđuje sve elemente klase Detektor

- Dovoljno je upisati Laser : Detektor kako bi naslijedili sve elemente klase Detektor

Korak 3: Implementirati sva polja i metode sa slike (Sl. 8.47)

- Metode Awake() i Update() premoštavamo, metoda PaliGasi() je nova funkcionalnost dok je OnTriggerEnter(…) novija verzija stare funkcionalnosti

Korak 4: Metoda Awake() treba premoštavati metodu Awake() iz osnovne klase, ali ujedno imati funkcionalnost metode iz osnovne klase + postavljanje brojača na 0

- Za premoštavanje virtual metoda koristimo ključnu riječ override
- Prava pristupa moraju ostati ista kao i kod osnovne klase
- Za pristup metodi iz osnovne klase koristimo se ključnom riječi base (npr. base.Metoda())

Korak 5: Metoda Update() treba premštavati metodu Update() iz osnovne klase, ali ujedno imati funkcionalnost metode iz osnovne klase + treba pokretati metodu za paljenje i gašenje

- Metoda PaliGasi() je tipa void, pripaziti kako ju pozvati

Korak 6: Metoda PaliGasi() treba, u odnosu na neku vrijednost brojača, paliti gasiti laser

- Brojač ćemo povećavati u realnom vremenu uz pomoć statičkog elementa deltaTime iz klase Time
- Kada brojač dostigne ili premaši vrijeme promjene potrebno je isključiti/uključiti laser (vizualno) te resetirati brojač za sljedeći ciklus
  - 3D objekte možemo „vizalno“ isključiti uz pomoć atributa renderer koji je vezan za objekt igre i sadržan u klasi MonoBehaviour
  - renderer je objekt klase Renderer. Prisjetimo se početka ovog dokumenta i svih „engine-a“ koje sadržu okruženja za izradu igara, jedan od „engine-a“ je render engine, tj. sustav koji je zadužen za crtanje 3D i 2D objekata na zaslon našeg ekrana. Prema tome, ako isključimo renderer nad nekim 3D objektom, on se neće crtati na zaslon našeg ekrana
  - renderer posjeduje atribut „enabled“ preko kojeg možemo uključiti-isključiti renderer
  - Napomena: ako je igrač detektiran prije vizualnog isključivanja 3D objekta, alarm se neće isključivati ako je renderer isključen jer smo postavili da se alarm isključi kada se objekt igre onesposobi (to smo napravili metodom OnDisable() unutar klase Detektor). Ovo nam ide u prilog jer mi želimo da se alarm isključuje samo kada je laser onesposobljen, nikako dok je njegova funkcionalnost pali-gasi zrake lasera

Korak 7: Metoda OnTriggerEnter(Collider other) treba biti nova implementacija iste metode iz klase Detektor koja će detektirati igrača pod uvjetom da je objekt igre aktivan ali vizualno skriven

- Napomena: Ova metoda ne smije imati istu funkcionalnost kao metoda iz klase Detektor
- Kako je metoda iz osnovne klase, pod istim nazivom, samo privatna ne možemo se koristiti premoštavanjem, a korištenje ključne riječi new nije potrebno za Unity. (Napomena: korištenje ključne riječi new može biti potrebno kod drugih okruženja)
- Program će preko detektora sudara uvijek pozivati metodu OnTriggerEnter(…) iz izvedene klase, ako ne implementiramo ovu metodu u izvedenoj klasi onda će program uvijek pozivati metodu iz osnovne klase (npr. Kamera nema novu implementaciju metode OnTriggerEnter(…) pa se detektiranje igrača ponaša kao kod klase Detektor)
- Koristimo se zaštićenom boolean varijablom „detektiran“ iz osnovne klase

Korak 8: Provjeriti ispravnost klase Laser

- Kada igrač prođe kroz laser dok je on vizualno skriven, alarm se ne smije aktivirati
- Razlika između gameobject.activeSelf i renderer.enabled:
  - u prvom slučaju objekt igre se isključuje skupa sa svim svojim komponentama uključujući i skripte što znači kako se skripta više ne izvršava,
  - u drugom slučaju se sve skripte normalno izvršavaju dok objekt igre nije vidljiv u igri.

```
using UnityEngine;
using System.Collections;

public class Laser : Detektor {
    private float brojac;
    public float vrijemePromjene = 4f;
    protected override void Awake(){
        base.Awake();
        brojac = 0f;
    }
    protected override void Update(){
        base.Update();
        PaliGasi();
    }
    private void PaliGasi(){
        brojac += Time.deltaTime;
        if (brojac >= vrijemePromjene){
            renderer.enabled = !renderer.enabled;
            brojac = 0f;
        }
    }
    void OnTriggerEnter(Collider other){
        if (other.gameObject == igrac_gameobject
            && renderer.enabled)
            detektiran = true;
    }
}
```

Kod 8.13 Konačan izgled klase Laser

8.6. Upravljanje događajima

Događaji su danas neizbježni u programima pisanim OO jezikom. Događaji nam omogućuju izradu programa s jako moćnim i kompliciranim funkcionalnostima. Događaje

možemo zamisliti kao nekakav odašiljač signala koji obavještava o nekoj zanimljivosti koja se dogodila. Obavijesti primaju samo oni koji su se zabilježili da žele primati obavijesti.

Za primjer ćemo uzeti analogiju prodavaonice računalne opreme. Na web stranici te iste prodavaonice kupujemo njihove proizvode te nam se ponudi mogućnost da zapišemo svoju e-mail adresu kako bi primali najnovije obavijesti o novim proizvodima. Analogija za naš odabir da želimo primati obavijesti bi bio slučaj kada se klasa upiše na event (eng. *subscribe*). Kada u prodavaonicu dođe najnoviji proizvod, to bi u slučaju programa bila ona zanimljivost koja se dogodila. Kada mi na svoj e-mail dobijemo obavijest o najnovijem proizvodu to bi u kontekstu programa značilo da objekt obavještavač (eng. *publisher*) obavještava klasu o toj zanimljivosti. Kada mi vidimo da imamo novu poštu, na nama je da tu poštu otvorimo i učinimo što želimo bilo to samo pročitati, otići i kupiti proizvod, kontaktirati prodavaonicu itd.. U kontekstu programa klasa mora svojom metodom odgovoriti na obavijest.

Ukratko, u programu postoji objekt koji sadrži u sebi tip podatka naznačen kao događaj (eng. *event*). Druge klase mogu se upisati da žele primiti obavijest o nekom događaju. Upisivanje se vrši preko neke metode iz klase koja se želi upisati. Kada se dogodi zanimljivost ili promjena koju klasa prati preko upravitelja događajima (eng. *event handler*), poziva se ona metoda koja je dio klase koja se upisala.

```
// #1 klasa Dogadjaji
public class Dogadjaji : MonoBehaviour {
      // #2 delegate tip
      public delegate void Handler(int broj);
      // #3 deklaracija mojHandler
      public Handler mojHandler;
      // #4 deklaracija mojEventHandler
      public static event Handler mojEventHandler;
      // #5 deklaracija mojObjekt
      public Dogadjaji mojObjekt;
      void Update(){
            // #6 provjera
            If(mojEventHandler != null)
                  mojEventHandler();// #7 obavijesti klase
      }
}
```

```
public class MojaKlasa : MonoBehaviour {
	void OnEnable(){
		// #8 dodjeli  metodu  eventu (subscribe)
		Dogadjaji.mojEventHandler += IspisBroja;
	}
	void OnDisable(){
		// #9 ukloni metodu iz eventa (unsubscribe)
		Dogadjaji.mojEventHandler -= IspisBroja;
	}
	// #10 obična metoda iz klase MojaKlasa
	public void IspisBroja(int broj) {
		print("Broj: " + broj);
	}
}
```

Kod 8.14 Događaji i delegati

Prije nastavka o događajima, upoznati ćemo se s pojmom delegate. Ovo je također jedna vrsta podatka i to referentni tip podatka. U ovaj tip podatka spremamo reference na metode. Pri deklariranu delegate tipa koristimo ključnu riječ delegate te u nastavku zapis metode kao u primjeru programskog koda gore. Primijetimo kako je linija #1 ima isto značenje kao i linija #2 samo se radi o dva različita tipa (`class` i `delegate`). Linije #3, #4 i #5 imaju isto značenje, a to je deklariranje.

Događaji se zapravo rade preko delegate jer su događaji ništa drugo nego specijalizirani delegate. Linija #3 i #4 zapravo rade istu stvar te se koriste na isti način. Možemo se onda zapitati zašto uopće koristimo event kada imamo obični delegate. Razlog korištenja su ograničenja koja posjeduje event, event dopušta samo dodjelu i uklanjanje metoda neke klase dok delegate nema ta ograničenja što može dovesti do neočekivanog ponašanja programa nepravilnim rukovanjem.

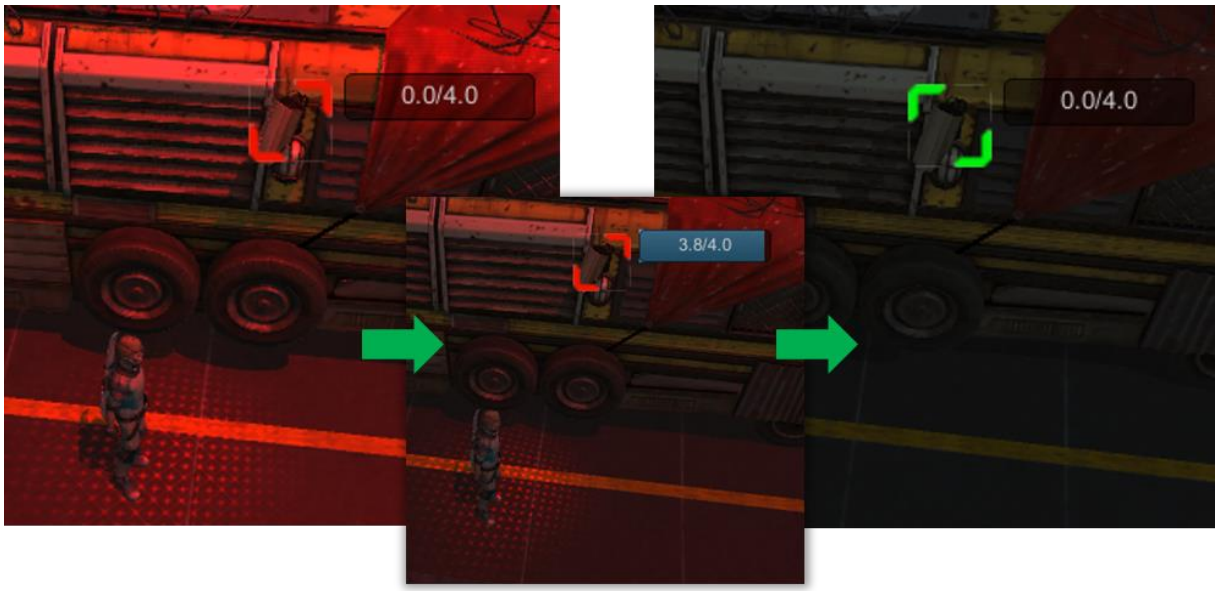
Objekt klase Dogadjaji je objekt koji „čeka“ promjenu koja aktivira događaje. To realiziramo na najjednostavniji način, npr. u Unity-u možemo koristiti metodu Update() u koju postavimo uvjet pod kojim želimo obavijestiti klase o događaju. Kad je taj postavljeni uvjet ispunjen znači da se dogodila ta promjena. Ovo nam pruža priliku da se događaji izvršavaju kada god mi poželimo, čak i bez uvjeta.

Važno je napomenuti da će program javljati grešku ukoliko pozivamo event u kojeg se niti jedna klasa nije upisala stoga je obavezna provjera je li event jedna null. Isto tako potrebno

je zapamtiti pravilo da za svako upisivanje u event treba postojati ispisivanje iz istoga jer može doći do otjecanja memorije (eng. *memory leak*). Npr. ako smo pri osposobljavanju objekta upisali, pri onesposobljavanju istoga trebamo ispisati, ne smijemo dopustiti da se više puta upisuje od ispisivanja i obrnuto.

8.6.1. Grafičko korisničko sučelje i kontrole

Grafičko korisničko sučelje ili skraćeno GUI (eng. *graphical user interface*) je jedan od bitnih elemenata igre kao i raznih softvera današnjice. Preko GUI-a korisnik komunicira s računalom točnije s programom u ovom slučaju igre. GUI je posrednik između igrača i samog „mozga igre“. Današnja GUI-a su vođena događajima (eng. *event driven*) pa možemo pretpostaviti kako programer treba dobro poznavati događaje i sve vezano uz njih. Iako kod GUI veću ulogu imaju dizajn i područje zvano HCI (eng. *human computer interaction*), u ovim vježbama ćemo se zadržati isključivo na programiranju.



Sl. 8.48 Izgled elemenata sučelja za interakciju (GUI)

Na slici (Sl. 8.48) vidi se već isprogramirani element sučelja u akciji. Kao što možemo zaključiti crvena sličica nam označava nepoželjno stanje dok zelena sličica predstavlja poželjno stanje. Za prebacivanje između dva stanja imamo traku koja se puni od 0 do 4. Tekst sa slike (Sl. 8.49) i brojevi sa slike (Sl. 8.48) su zapisani unutar jedne od kontrola (eng. *control*) zvane Label. Osim ove kontrole postoji niz drugih kontrola koje možemo a i ne moramo dodati na sučelje. Kontrole možemo na neki način podijeliti u dvije skupine,

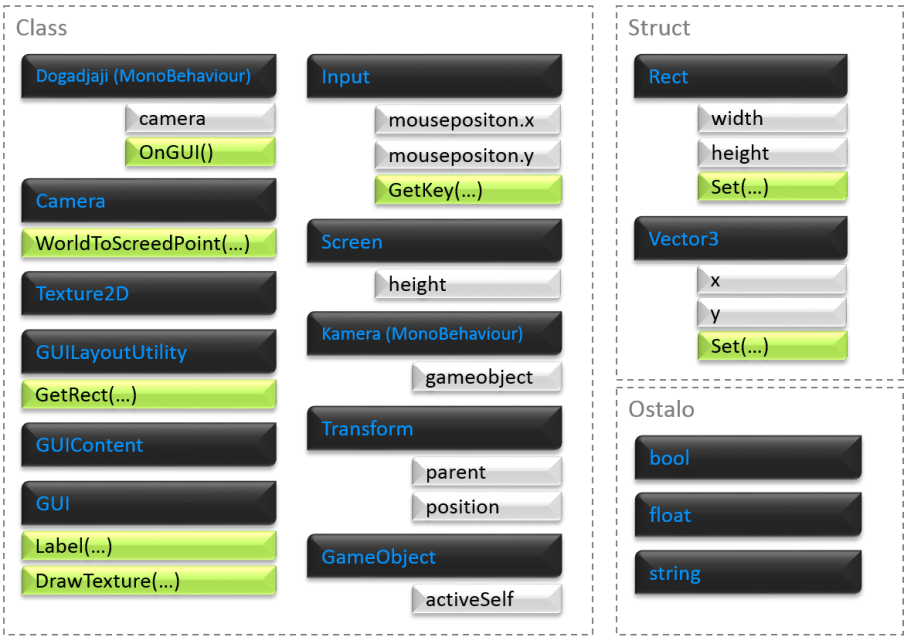
one koje nam služe za prikaz poruka korisniku (npr. okvir – eng. *Label*) i one koje nam služe za izravnu interakciju preko miša ili tipkovnice (npr. dugme – eng. *button*).

Sl. 8.49 Prikaz opisa na pokazivaču miša

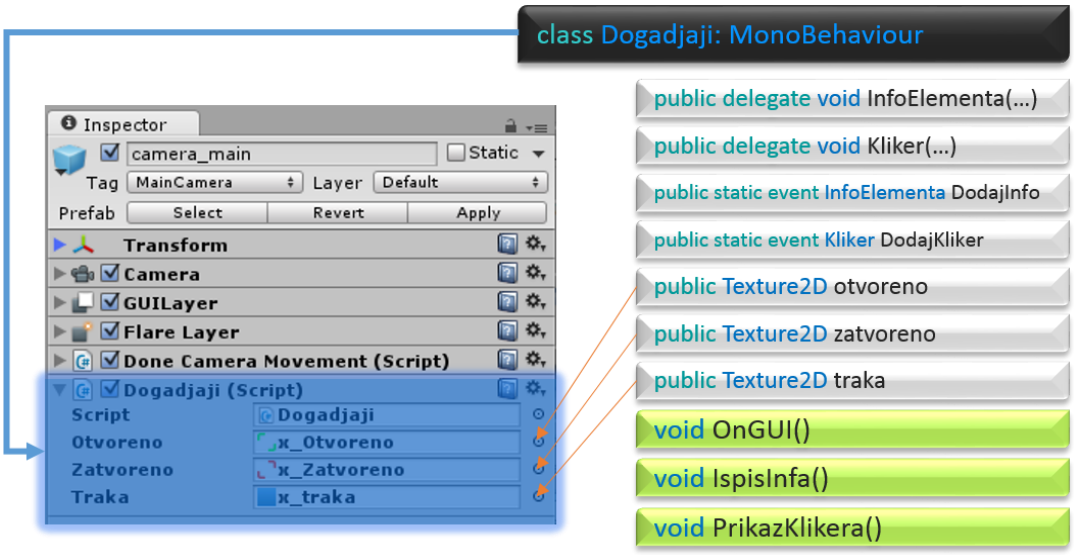
Slike koje nazivamo teksture (eng. *texture*) su također sastavni dio GUI-a. Uz pomoć njih izrađujemo vlastiti dizajn kontrola dok pozadina (programski kod) ostaje nepromijenjen. Teksture se također koriste i za oslikavanje 3D objekata. Sve to je vidljivo na slikama (Sl. 8.48 i Sl. 8.49).

8.6.2. Vježba br_6

Opet je zadan ciljani izgled klase (Sl. 8.51) te minimalan broj elemenata za uspješno obavljanje zadatka (Sl. 8.50). Događaje ćemo koristiti na dva načina, prvi je aktiviranje događaja kada se neka klasa zapiše, a drugi način je kada igrač pritisne tipku E.



Sl. 8.50 Podatci potrebni za izvršavanje zadatka



Sl. 8.51 Ciljani izgled klase Dogadjaji

Korak 1: Napraviti novu skriptu naziva Dogadjaji te ju spremiti u direktorij Vjezba6

Korak 2: Vezati klasu na glavnu kameru u igri (kamera koja prati igrača – camera_main)

- Događaje je najbolje vezati za glavnu kameru igre zbog pristupa samom objektu kamere

Korak 3: Implementirati sve elemente sa slike (Sl. 8.51)

- delegate elementi neka budu tipa void, prvi delegate neka ima jedan izlazni parametar tipa string naziva poruka, drugi delegate neka ima pet parametara (prvi neka bude

naziva kamera klase Camera, drugi neka bude naziva pozicija tipa Vector3, treći neka bude naziva ukljuceno tipa bool i zadnja dva neka budu tipa float s nazivima _od i _do). U drugom delegate neka svi parametri, osim kamere, budu izlazni parametri

  - Camera je klasa koja opisuje kamere u igri kao što je glavna kamera koja prati igrača
  - Izlazni parametri su parametri koji moraju poprimiti neku vrijednost unutar metode koju pozivamo, pri pozivanju metode kao parametar šaljemo neku varijablu ili objekt u kombinaciji s ključnom riječi out. Za deklariranje metode također trebamo koristiti ključnu riječ out. Npr. deklaracija void Metoda(out int broj){…} – pozivanje Metoda(mojBroj)

- Događaji (event) neka budu izvedeni od delegate tipova kao na slici

Korak 4: Teksture pronaći u Unity-u pod direktorijem Textures te ih vezati na skriptu koju smo postavili na glavnu kameru

- Tri teksture trebaju biti javne kako bi preko Inspector-a mogli vezati slike iz direktorija projekta
- Teksture trebaju biti klasa Texture i Texture2D. Klasa Texture2D je izvedena klasa klase Texture

Korak 5: Metoda OnGUI() neka pokreće metode IspisInfa() i PrikazKlikera()

- Prethodno smo spomenuli na koji način želimo obavještavati klase o nekoj zanimljivoj promjenu u programu. U ovom slučaju ćemo implementirati događaje u metode IspisInfa() i PrikazKlikera(). Pošto naši događaji preko izlaznih parametara sakupljaju podatke iz drugih klasa kako bi te iste podatke ispisali na ekran, trebamo se koristiti metodom OnGUI() koja je kao i Update() metoda pokrenuta nekim drugim događajima.
- Ovaj zadatak se mogao riješiti i bez događaja tako da svaka od klasa sadrži metodu OnGUI() koja će posebno crtati elemente sučelja. Razlog korištenja događaja je efikasnost i u svrhu demonstracije. U igri imamo više elemenata koji zahtijevaju crtanje istih kontrola, a znamo da korištenjem sučelja svejedno moramo raditi implementacije iste metode više puta. Zaključujemo da je najefikasnije koristiti jednu odvojenu klasu s metodom za crtanje tih elemenata (crtanje elemenata sučelja može biti dugotrajan proces). Unutar svih klasa koje zahtijevaju te elemente za crtanje implementiramo samo metode koje nam daju informacije za ispis.

Korak 6: Metoda IspisInfa() neka pokreće događaj DodajInfo pod uvjetom da nije prazan.

- Prisjetimo se: događaje pokrećemo kao najobičnije metode
- Napomena: pokrenuti događaj, ukoliko ima parametre, mora primati parametre stoga je potrebno deklarirati varijable i objekte unutar metode IspisInfa()

Korak 7: Unutar klase Kamera implementirati metodu koja odgovara delegate InfoElementa te upisati i ispisati tu metodu s događaja DodajInfo uz pomoć metoda OnMouseEnter() i OnMouseExit().

- Metoda treba biti istoga tipa te primati isti broj parametra istoga tipa i redoslijeda. Naziv metode i parametara nije bitan premda se preporuča staviti sličan ako ne i isti naziv zbog lakše orijentacije
- Unutar metode dovoljno je postaviti vrijednost parametra (vrijednost parametra neka bude tekstualna poruka "Za gašenje kamere držati tipku E").
- Metode, unutar klase Kamera, upisujemo koristeći operator +=, a ispisujemo koristeći operator -=

Korak 8: Metode iz drugih klasa će slati tekstualnu poruku preko out parametra, cilj je da nakon pokretanja događaja unutar klase Dogadjaji napravimo okvir (Label) u koji ćemo ispisati poruku.

- Napomena: out parametri nisu isto što i return.
- Okvire (label) radimo preko statičke metode GUI.Label(…). Možemo koristiti jednu od metoda s tim imenom npr. onu koja prima parametre: strukturu tipa Rect, poruku tipa string, stil tipa string ili stil klase GUIStyle)
  - Rect trebamo napraviti, poruku smo pročitali uz pomoć događaja, a za stil je dovoljno upisati "box" (vlastiti stilovi se mogu izrađivati uz pomoć klase GUIStyle)
- Rect je struktura koja predstavlja okvir u obliku pravokutnika s dimenzijama i pozicijom na zaslonu ekrana
  - možemo napraviti pravokutnik tipa Rect uz pomoć konstruktora i vlastitih dimenzija ako su sljedeće dvije toče nerazumljive (dimenzije i poziciju postavljati konstruktorom)
  - Preko metode GUILayoutUtility.GetRect(…) možemo napraviti novi objekt tipa Rect čije dimenzije će se računati u odnosu na parametare (u ovom slučaju duljinu teksta). Za parametre šaljemo poruku ali kao GUIContent i naziv tipa kontrole ("label"). Pošto poruku moramo poslati kao GUIContent, moramo napraviti novi objekt klase GUIContent uz pomoć konstruktora koji prima poruku tipa string (new GUIContent("poruka")). Ako smo na ovaj način napravili pravokutnik onda trebamo dimenzije i poziciju mijenjati metodom Set(…) koju posjeduje svaka varijabla tipa Rect.

  - Pozicija pravokutnika: Input.mousePosition.x + 20 za x-os i Screen.height - Input.mousePosition.y za y-os
  - Za x-os radimo pomak za 20 piksela
  - Za y-os potrebno je u odnosu na visinu ekrana okrenuti poziciju jer pretvaranjem koordinate y iz svijeta na ekran dobijemo vrijednost reflektiranu u odnosu na sredinu ekrana
  - Visinu i širinu ekrana iščitavamo preko statičkih atributa width i height klase Screen
- Ako koristimo metodu Set(…) onda moramo poslati parametre za dimenzije, dimenzije možemo ostaviti iste (pravokutnik.width i pravokutnik.height ako je pravokutnik tipa Rect)
- Napomena: za svaku metodu prvo provjeriti kojeg tipa su parametri prelaskom pokazivača miša preko metode ili dok zapisujemo njen naziv u Monodevelop ili Visual Studio

Korak 9: Metoda PrikazKlikera() neka pokreće događaj DodajKliker pod uvjetom da nije prazan i da je zadržana tipka E.

- Prisjetimo se: događaje pokrećemo kao najobičnije metode
- Napomena: pokrenuti događaj, ukoliko ima parametre, mora primati parametre stoga je potrebno deklarirati varijable i objekte unutar metode PrikazKlikera()

Korak 10: Unutar klase Kamera implementirati metodu koja odgovara delegate Kliker te upisati i ispisati tu metodu s događaja DodajKliker uz pomoć metoda OnTriggerEnter(…) i OnTriggerExit(…)

- Na isti način kao i u koraku 7 trebamo upisati i ispisati metodu iz događaja, ali ovaj put uz pomoć metoda OnTriggerEnter(…) i OnTriggerExit(…). U prethodnom slučaju se info metode izvrše prelaskom miša preko 3D objekta bez obzira posebne uvjete, ali u ovom slučaju, iako upisane u događaje, metode se izvršavaju samo onda kada igrač zadrži tipku E
- Varijabla „pozicija“ neka sadrži vrijednost pozicije 3D objekta, ali na ekranu ne u svijetu
  - Poziciju objekta u svijetu dohvaćamo preko objekta transform (transform.position)
  - Ako na ovaj način pristupimo poziciji nećemo dobiti poziciju 3D sigurnosne kamere jer je na taj 3D model vezan još jedan objekt igre koji simulira crvene svjetlosne zrake. U stablastoj hijerarhiji scene sigurnosna kamera je roditelj ovog objekta stoga joj pristupamo kao transform.parent, a pozicija roditelja bi onda bila transform.parent.transform.position

  - Pozicija koju tražimo je tipa Vector3. Da bi pretvorili koordinate svijeta u koordinate sučelja trebamo se koristiti glavom kamerom igre i metodom WorldToScreenPoint(…). Ova metoda kao parametar prima poziciju u svijetu koja je također tipa Vector3.
- Varijabla „ukljuceno“ u slučaju sigurnosne kamere neka odgovara atributu objekta gameObject koji nam govori je li objekt aktivan (Prisjetimo se: gameObject.activeSelf)
- Varijable „_od“ i „_do“ u slučaju sigurnosne kamere neka odgovaraju atributima klase Kamera (Brojac i HakirajNa)

Korak 11: Metode drugih klasa će slati određene informacije preko out parametara, koristeći te parametre treba nacrtati 3 Label kontrole redom za sliku uključeno/isključeno, sliku trake i tekst trake.

- Kada smo uspješno dohvatili parametre koji su nam bili potrebni možemo krenuti na crtanje pravokutnika i okvira sučelja (label)
- Pravokutnik za zelenu i crvenu sliku:
  - Pozicija-x je pozicija koju smo pročitali uz pomoć izlaznog parametra „pozicija“ – 25
  - Pozicija-y je visina ekrana – vrijednost y-osi izlaznog parametra „pozicija“ – 25
  - Širina neka je 50
  - Visina neka je 50
  - Preko GUI.Label(…) nacrtati sliku – parametri metode su pravokutnik i tekstura (teksturu birati u odnosu na parametar „ukljuceno“)
- Pravokutnik za sliku trake:
  - Pozicija-x je pozicija koju smo pročitali uz pomoć izlaznog parametra „pozicija“ + 25
  - Pozicija-y je visina ekrana – vrijednost y-osi izlaznog parametra „pozicija“ – 25
  - Širina neka se računa u postotcima u odnosu na parametre _od i _do (max 100)
  - Visina neka je 25
  - Preko GUI.DrawTexture(…) nacrtati sliku bez okvira – parametri metode su pravokutnik i tektura trake
- Pravokutnik za ispis trenutnog stanja hakiranja (može u formatu % ili od/do:
  - Pozicija i dimenzije moraju biti iste kao i kod slike trake s tim da je širina jednaka maximalnoj širini trake (100)
  - Preko GUI.Label(…) ispisati stanje hakiranja – parametri metodu su pravokutnik, stanje i stil (“box“)

Korak 12: Ukoliko je sve pravilno implementirano u igri se elementi sučelja ponašaju na sljedeći način:

- Prelaskom pokazivača miša preko 3D objekta u igri se prikazuje okvir s porukom kako hakirati tu napravu (Slika 6.2)
- Približavanjem 3D objektu se ne događa ništa, ali zadržavanjem tipke E imamo rezultat kao sa slike 6.1. Nakon što se traka napuni prema varijablama _od i _do mijenja se tekstura zeleno - crveno
- Funkcionalnost svih ostalih klasa nije ugrožena te se trebaju svi ostali elementi ponašati kao i prije ove vježbe

Korak 13: Kao dodatnu vježbu potrebno je implementirati metode po uzoru na klasu Kamera i to u klase Konzola i Vrata. Klasa Vrata će neka ima samo metodu za info elementa s tim da se taj info prikazuje samo kada prelazimo pokazivačem miša preko zaključanih vrata

```
using UnityEngine;
using System.Collections;
public class Dogadjaji : MonoBehaviour{

    public delegate void InfoElementa(out string poruka);
    public delegate void Kliker(Camera glavna_kamera,
                                out Vector3 pozicija,
                                out bool ukljuceno,
                                out float _od,
                                out float _do);
    public static event InfoElementa DodajInfo;
    public static event Kliker DodajKliker;

    public Texture2D otvoreno;
    public Texture2D zatvoreno;
    public Texture2D traka;

    void OnGUI(){
        IspisInfa();
        PrikazKlikera();
    }
    void IspisInfa(){
        string poruka;
        if (DodajInfo != null)
        {
            DodajInfo(out poruka);
```

```
            Rect labelRect = GUILayoutUtility.GetRect(
                  new GUIContent(poruka),
                  "label");
            labelRect.Set(
                  Input.mousePosition.x + 20,
                  Screen.height - Input.mousePosition.y,
                  labelRect.width + 10,
                  labelRect.height + 5);
            GUI.Label(labelRect, poruka, "box");
        }
    }
    void PrikazKlikera(){
        Vector3 pozicija;
        bool ukljuceno;
        float _od, _do;
        if (DodajKliker != null && Input.GetKey(KeyCode.E)){
            DodajKliker(
                  camera,
                  out pozicija,
                  out ukljuceno,
                  out _od,
                  out _do);
            Rect slikaRect = new Rect(
                  pozicija.x - 25,
                  Screen.height - pozicija.y - 25,
                  50,
                  50);
            GUI.Label(
                  slikaRect,
                  ukljuceno ? zatvoreno : otvoreno);
            Rect trakaRect = new Rect(
                  pozicija.x + 25,
                  Screen.height - pozicija.y - 25,
                  100 * (_od / _do),
                  25);
            GUI.DrawTexture(trakaRect, traka);
                  Rect porukaRect = new Rect(
                  pozicija.x + 25,
                  Screen.height - pozicija.y - 25,
```

```
                    100,
                    25);
                GUI.Label(
                    porukaRect,
                    _od.ToString("f1") + "/" +
                    _do.ToString("f1"),
                    "box");
            }
        }
    }
```

Kod 8.15 Konačan izgled klase Dogadjaji

Na slici (Sl. 8.52) prikazan je kod koji se izravno veže na kod (Kod 8.15), bojama (plava, zelena) označene su metode koje se dodaju u događaje koji „čekaju“ trenutak kada će izvršiti te iste metode.

```
void OnTriggerEnter(Collider other)
{
    if (other.gameObject == igrac_gameobject && gameObject.activeSelf)
    {
        Dogadjaji.DodajKliker += Kliker_Kamera;
        detektiran = true;
    }
}
void OnTriggerExit(Collider other)
{
    if (other.gameObject == igrac_gameobject && gameObject.activeSelf)
    {
        Dogadjaji.DodajKliker -= Kliker_Kamera;
        brojac = 0;
    }
}
void OnMouseEnter()
{
    Dogadjaji.DodajInfo += Info_Kamera;
}
void OnMouseExit()
{
    Dogadjaji.DodajInfo -= Info_Kamera;
}

void Info_Kamera(out string poruka)
{
    poruka = "Za gašenje kamere kliknuti E";
}
void Kliker_Kamera(Camera glavna_kamera,
                   out Vector3 pozicija,
                   out bool ukljuceno,
                   out float _od,
                   out float _do)
{
    pozicija = glavna_kamera.WorldToScreenPoint(transform.parent.transform.position);
    ukljuceno = gameObject.activeSelf;
    _od = Brojac; _do = hakirajNa;
}
```

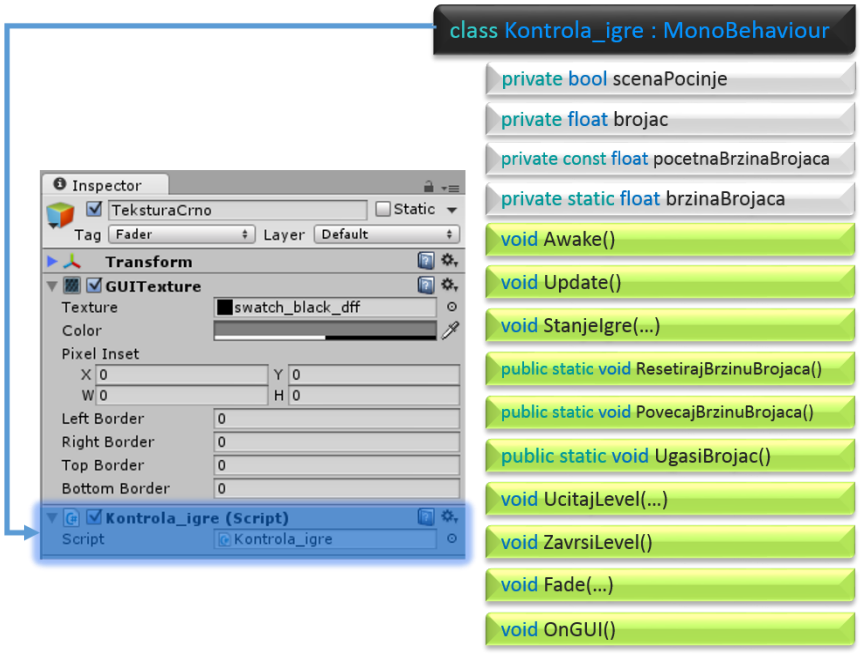
Sl. 8.52 Primjer korištenja događaja unutar klase Kamera

Događaji su sami po sebi komplicirani za shvaćanje posebno kod programera početnika. U kombinaciji s drugim konceptima OOP-a postaju još kompleksniji i apstraktniji što otežava njihovo shvaćanje ali i korištenje.

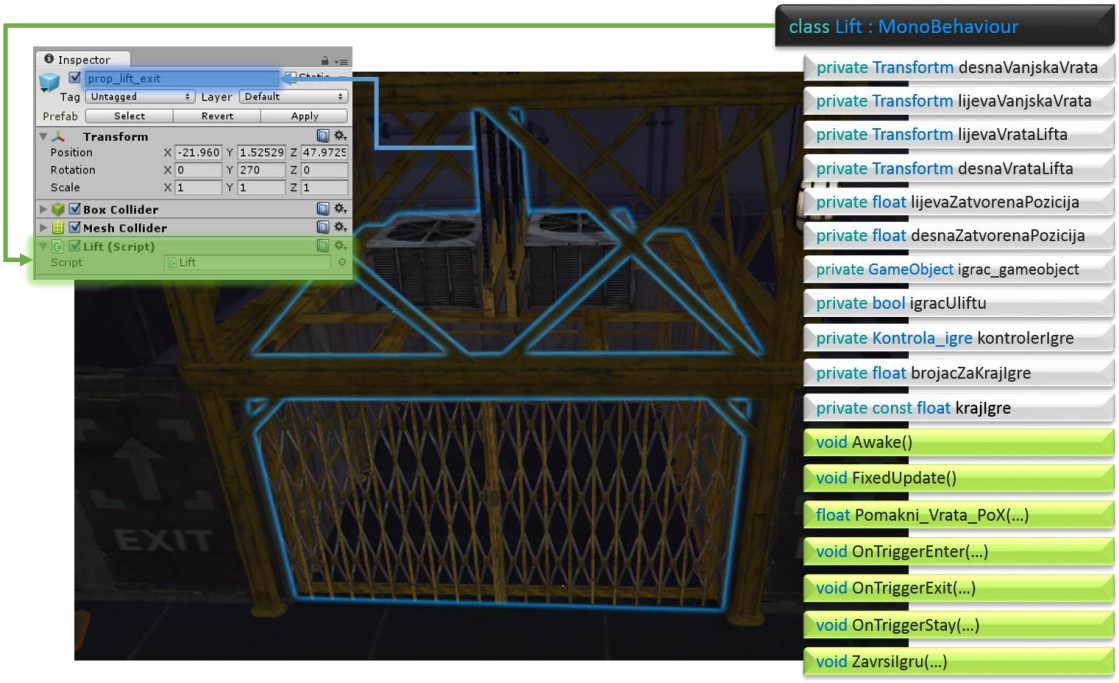
Kontrole su uvijek vezane za događaje, ali način izrade kontrola nije jednak kod različitih razvojnih okruženja. Npr. za Unity kontrole postoje statičke metode koje crtaju te iste kontrole prema strukturama kao što je pravokutnik Rect i klasama kao što je GUIStyle. Drugi primjer bi bile kontrole za izradu Windows aplikacija uz pomoć formi (WindowsFormApplication). U tom slučaju su sve kontrole i forme predstavljene kao klase, a ne kao metode koje samo crtaju te iste kontrole.

8.7. Dodatni dio programskog koda za vježbe

U nastavku prezentirani su dijelovi programskog koda koji odgovaraju slikama (Sl. 8.53 i Sl. 8.54).



Sl. 8.53 Ciljani izgled klase Kontrola_Igre



Sl. 8.54 Ciljani izgled klase Lift

```
using UnityEngine;
using System.Collections;

public class Kontrola_igre : MonoBehaviour {
    private bool scenaPocinje = true;
    private float brojac = 100f;
    private const float pocetnaBrzinaBrojaca = 1f;
    private static float brzinaBrojaca;

    void Awake(){
            brzinaBrojaca = pocetnaBrzinaBrojaca;
            guiTexture.pixelInset = new Rect(
                  0f,
                  0f,
                  Screen.width,
                  Screen.height);
    }
    void Update(){
        if (scenaPocinje) UcitajLevel();
        StanjeIgre();
    }
    void StanjeIgre(){
        brojac -= Time.deltaTime * brzinaBrojaca;
        //Debug.Log((int)brojac + "sec");
        if (brojac <= 0) ZavrsiLevel();
    }
    public static void ResetirajBrzinuBrojaca(){
            brzinaBrojaca = pocetnaBrzinaBrojaca;
    }
    public static void PovecajBrzinuBrojaca(){
        brzinaBrojaca++;
    }
    public static void UgasiBrojac(){
        brzinaBrojaca = 0;
    }
    void UcitajLevel(){
        Fade(true);
        if (guiTexture.color.a <= 0.05f){
            guiTexture.enabled = false;
            scenaPocinje = false;
```

```
        }
    }
    public void ZavrsiLevel(){
        guiTexture.enabled = true;
        Fade(false);
        if (guiTexture.color.a >= 0.95f)
            Application.LoadLevel(0);
    }
    void Fade(bool crno_u_cisto){
        if (crno_u_cisto)
            guiTexture.color = Color.Lerp(
                  guiTexture.color,
                  Color.clear,
                  1.5f * Time.deltaTime);
        else guiTexture.color = Color.Lerp(
                  guiTexture.color,
                  Color.black,
                  1.5f * Time.deltaTime);
    }
      void OnGUI(){
            GUI.Label(
                  new Rect(1,1,200,30),
                  brojac.ToString("f1") + "s do kraja.",
                  "box");
      }
}
```

Kod 8.16 Konačan izgled klase Kontrola_Igre

```
public void AktivirajAlarm(){
        if (!aktivan){
            aktivan = true;
            Kontrola_igre.PovecajBrzinuBrojaca();
        }
    }
    public void UgasiAlarm(){
        if (aktivan){
            aktivan = false;
            Kontrola_igre.ResetirajBrzinuBrojaca();
        }
```

```
    }
```

Kod 8.17 Pristup kontroli igre iz klase Alarm

```
using UnityEngine;
using System.Collections;
using namOznake;

class Lift : MonoBehaviour
{
    private Transform lijevaVanjskaVrata;
    private Transform desnaVanjskaVrata;
    private Transform lijevaVrataLifta;
    private Transform desnaVrataLifta;
    private float lijevaZatvorenaPozicija;
    private float desnaZatvorenaPozicija;

    private GameObject igrac_gameobject;
    private bool igracUliftu;

    private Kontrola_igre kontrolerIgre;
    private float brojacZaKrajIgre;
    private const float krajIgre = 2f;

    void Awake()
    {
        lijevaVanjskaVrata =
GameObject.Find("door_exit_outer_left_001").transform;
        desnaVanjskaVrata =
GameObject.Find("door_exit_outer_right_001").transform;
        lijevaVrataLifta =
GameObject.Find("door_exit_inner_left_001").transform;
        desnaVrataLifta =
GameObject.Find("door_exit_inner_right_001").transform;
        lijevaZatvorenaPozicija =
lijevaVrataLifta.position.x;
        desnaZatvorenaPozicija = desnaVrataLifta.position.x;
        igrac_gameobject =
GameObject.FindGameObjectWithTag(Tagovi.player);
```

```
        kontrolerIgre =
GameObject.FindGameObjectWithTag(Tagovi.gameController).GetCo
mponent<Kontrola_igre>();
    }
    void Update()
    {
        if (!igracUliftu)
        {
            lijevaVrataLifta.position = new Vector3(
                  Pomakni_Vrata_PoX(
                        lijevaVrataLifta.position.x,
                        lijevaVanjskaVrata.position.x),
                  lijevaVrataLifta.position.y,
                  lijevaVrataLifta.position.z);
            desnaVrataLifta.position = new Vector3(
                  Pomakni_Vrata_PoX(
                        desnaVrataLifta.position.x,
                        desnaVanjskaVrata.position.x),
                  desnaVrataLifta.position.y,
                  desnaVrataLifta.position.z);
        }
        else if (brojacZaKrajIgre >= krajIgre)
        {
            lijevaVrataLifta.position = new Vector3(
                  Pomakni_Vrata_PoX(
                        lijevaVrataLifta.position.x,
                        lijevaZatvorenaPozicija),
                  lijevaVrataLifta.position.y,
                  lijevaVrataLifta.position.z);
            desnaVrataLifta.position = new Vector3(
                  Pomakni_Vrata_PoX(
                        desnaVrataLifta.position.x,
                        desnaZatvorenaPozicija),
                  desnaVrataLifta.position.y,
                  desnaVrataLifta.position.z);
            ZavrsiIgru();
        }
    }
    float Pomakni_Vrata_PoX(float odPozicijeX, float
doPozicijeX){
```

```
            float novaPozicija_X = Mathf.Lerp(
                                        odPozicijeX,
                                        doPozicijeX,
                                        7f*Time.deltaTime);
            return novaPozicija_X;
        }
        void OnTriggerEnter(Collider other){
            if (other.gameObject == igrac_gameobject)
                igracUliftu = true;
        }
        void OnTriggerExit(Collider other){
            if (other.gameObject == igrac_gameobject){
                igracUliftu = false;
                brojacZaKrajIgre = 0f;
            }
        }
        void OnTriggerStay(Collider other){
            if (other.gameObject == igrac_gameobject)
                brojacZaKrajIgre += Time.deltaTime;
        }
        void ZavrsiIgru(){
            igrac_gameobject.transform.parent = transform;
            transform.Translate(
                Vector3.up * 3f * Time.deltaTime);
            Kontrola_igre.UgasiBrojac();
            if(brojacZaKrajIgre >= krajIgre*2f)
                kontrolerIgre.ZavrsiLevel();
        }
    }
```

Kod 8.18 Konačan izgled klase Lift