Osijek, 2023.

ELPROS
REGIONALNI CENTAR KOMPETENTNOSTI
ELEKTROTEHNIČKA I PROMETNA ŠKOLA OSIJEK

Andrija Zorić i Jan Juračić

RAZVOJ ALTERNATIVNIH OKRUŽENJA

Priručnik za polaznike

ESF UČINKOVITI LJUDSKI POTENCIJALI
EUROPSKI STRUKTURNI I INVESTICIJSKI FONDOVI

RAZVOJ ALTERNATIVNIH OKRUŽENJA

Priručnik za polaznike

Algebra d.o.o.

Osijek, 2023.

Projekt je sufinancirala Europska unija iz Europskog socijalnog fonda.

IMPRESUM

Autori: **Andrija Zorić i Jan Juračić**

Urednica i stručnjakinja za metodologiju izrade nastavnih materijala: **Martina Podobnik**

Recenzent: **Aleksander Radovan**

Grafičko oblikovanje: **BARREK d.o.o.**

Lektorica: **Ivana Previšić**

Naslov: **Razvoj alternativnih okruženja** (priručnik za polaznike u programu obrazovanja za stjecanje djelomične kvalifikacije „Specijalist razvoja alternativnog okruženja“ – obrazovanje odraslih, razina 5 HKO)

Nakladnik i nositelj prava: **Elektrotehnička i prometna škola Osijek**

Odgovorna osoba: **ravnatelj Antun Kovačić, dipl. ing. el.**

Mjesto i godina izdanja: **Osijek, 2023.**

Za više informacija:
Elektrotehnička i prometna škola Osijek
Istarska 3
31000 Osijek
E-mail: ured@ss-elektrotehnicka-prometna-os.skole.hr

Regionalni centar kompetentnosti ELPROS

Osijek, 2023.

Sadržaj:

1. TEMELJI PROGRAMSKOG JEZKA C#, RAZVOJ U UNITYJU I XR SVIJET

Ovaj priručnik uvodi u osnovne i napredne koncepte vezane uz programiranje, razvoj igara i proširenu stvarnost. Omogućuje stvaranje čvrstih temelja potrebnih za postizanjem stručnosti u tim područjima. Bez obzira na stupanj predznanja, priručnik pomaže nadograditi i proširiti znanja i razviti vještine.

1.1. Uvod u C#

Na početku putovanja u svijet programiranja, ključnu važnost ima izbor programskoga jezika kojim će se koristiti. ***C#*** je postao jednim od najistaknutijih izbora mnogih razvojnih programera, i to s dobrim razlogom. Taj jezik, koji je razvio *Microsoft* kao dio .NET platforme, kombinira snagu, fleksibilnost i čitljivost, što ga čini idealnim za početnike i iskusne programere.

Sadržaj priručnika započinje osnovama programiranja koristeći se jezikom *C#*. Iako je C# svestran i moćan, njegova stvarna ljepota leži u intuitivnosti i strukturiranosti. Te ga karakteristike čine odličnim izborom za one koji tek ulaze u svijet kodiranja. U poglavlju „Uvod u *C#*“ proučavat će se **varijable**, koje su osnovni spremnici informacija u programiranju, i različiti načini njihova korištenja. Istražit će se i **petlje**, koje omogućavaju ponavljanje određenih dijelova koda da bi se automatizirali zadaci. Naposlijetku, **kontrolne strukture**, poput uvjetnih iskaza, pružaju mogućnost usmjeravanja toka programa na temelju određenih uvjeta.

1.2. Objektno orijentirano programiranje (OOP)

Objektno orijentirano programiranje (OOP) predstavlja jednu od najrevolucionarnijih promjena u načinu pristupanja programiranju, uvodeći koncepte koji se usredotočuju na organizaciju i strukturiranje koda oko objekata – entiteta, koji sadrže podatke i funkcionalnost. Ta paradigmatska promjena ne omogućava samo pogled na kod iz sasvim nove perspektive, već oprema alatima za pisanje modularnijega, održivijeg i fleksibilnijeg koda.

Razmatrat će se ključni aspekti objektno orijentiranog programiranja, poput klasa, nasljeđivanja i enkapsulacije. Također će biti istraženi različiti oblikovni obrasci koji mogu pomoći u rješavanju uobičajenih programerskih problema.

1.3. Razvoj okruženja u Unityju

Unity je s vremenom postao sinonim za moderni razvoj igara i interaktivnih aplikacija. Svojom neusporedivom sposobnošću da podrži niz platformi, od stolnih računala do mobilnih uređaja i VR/AR/MR naglavnih setova (eng. *Headsetova*), *Unity* je postao omiljen izbor mnogih razvojnih programera, od entuzijasta do profesionalaca.

S ciljem omogućavanja dubljeg razumijevanja toga moćnog alata, ovo poglavlje priručnika nudi detaljan pregled osnovnih komponenata koje čine *Unity* okruženje. Počet će se s temeljima – kako kreirati i manipulirati objektima unutar 3D scene, postavljati kamere, koristiti se osvjetljenjem da bi se postigao željeni estetski izgled te kako kontrolirati interakcije unutar scene.

U konačnici, cilj je ovog poglavlja priručnika omogućiti stjecanje znanja potrebnih za samouvjereno pristupanje svakom projektu *Unity* platforme. Osim čvrstih temelja u tehničkom smislu, obradit će se i načini kako najbolje iskoristiti alate i tehnike koje *Unity* nudi u svrhu realizacije vizija i ideja s povjerenjem i sigurnošću.

1.4. Razvoj AR, VR i MR aplikacija

Proširena (AR), virtualna (VR) i mješovita stvarnost (MR) nisu samo etikete tehnologije, već revolucionarne platforme koje mijenjaju način percepcije i interakcije s digitalnim svijetom. Od igara i zabave, preko medicinske i obrazovne primjene, do industrijskih aplikacija – ti oblici digitalne stvarnosti proširuju granice mogućega i definiraju budućnost interaktivnih iskustava.

U ovom segmentu priručnika biti će obrađene teorijske osnove AR-a, VR-a i MR-a, te će se usredotočiti na njihovu praktičnu primjenu posredstvom *Unity* razvojnoga okruženja. Razmotrit će se različiti SDK-ovi, alati i tehnike, koje se uobičajeno koriste za kreiranje uživljujućih aplikacija. Uz praktične primjere i vježbe, ovo poglavlje pomoći će u razumijevanju osnovnih i naprednih koncepata svake od navedenih tehnologija, omogućujući kreiranje tehnološki naprednih i korisnički privlačnih aplikacija.

Naposlijetku, ovaj priručnik je više od skupa uputa. On nudi priliku za stjecanje znanja koja će oblikovati karijere, otvoriti vrata novim mogućnostima i pomoći pojedincima na putu prema ostvarenju snova u svijetu digitalnog razvoja.

2. UVOD U PROGRAMIRANJE U PROGRAMSKOM JEZIKU C#

U OVOM POGLAVLJU NAUČIT ĆETE:

- razlikovati programski jezik visokoga i niskog nivoa te osnovne karakteristike jezika C#
- koristiti se radnim površinama, alatnim trakama i drugim ključnim komponentama Visual Studio IDE-a
- koristiti se sintaktičkim elementima jezika C#
- razlikovati pseudokod od stvarne implementacije
- razlikovati tipove podataka u C#-u
- osmisliti uvjetnu logiku te se koristiti bool izrazima i uvjetnim izjavama
- konstruirati petlje
- razlikovati vrste kolekcija u C#-u, kao što su nizovi, liste i rječnici
- konstruirati vlastite funkcije u svojim programima.

2.1. Osnovne pretpostavke programiranja

Programiranje je proces stvaranja računalnih programa radi postizanja određenoga cilja. Programi omogućavaju računalima izvršavanje niza uputa ili instrukcija za obavljanje određenoga zadatka ili pružanje određenih funkcionalnosti. U osnovi, programiranje se temelji na različitim pretpostavkama i konceptima koji pomažu programerima razviti efikasne i funkcionalne programe.
U nastavku će biti opisane **osnovne pretpostavke programiranja**.

1. **Precizne instrukcije:** programiranje se temelji na pretpostavci da će računalo izvršiti niz instrukcija kako bi postiglo određeni cilj. Računala nemaju vlastitu inteligenciju, stoga programer mora jasno definirati svaki korak i zadatke koje računalo treba obaviti.

2. **Redoslijed izvršenja:** programiranje pretpostavlja da će se instrukcije izvršavati u određenom redoslijedu. To znači da redoslijed u kojem su instrukcije napisane ima velik utjecaj na ponašanje programa.
3. **Skladištenje podataka:** podatci se često koriste u programima, a programiranje pretpostavlja da je moguće pohraniti podatke na određenim lokacijama i manipulirati njima.
4. **Upravljanje resursima računala:** programiranje pretpostavlja ograničenja resursa, kao što su memorija i procesorska snaga. Programeri moraju pažljivo upravljati navedenim resursima da bi osigurali učinkovit rad njihovih programa.
5. **Kontrola tijeka programa:** programiranje se temelji na pretpostavci da programer ima kontrolu nad tijekom programa radi donošenja odluka i ponavljanja određenih dijelova koda.
6. **Testiranje i provjera:** programiranje podrazumijeva testiranje programa da bi se osigurao njihov ispravan rad. To uključuje provjeru ispravnosti koda, ispravljanje grešaka i zadovoljavanje specifikacija programa.
7. **Apstrakcija:** programiranje uključuje apstrakciju, što znači da se programeri koriste apstraktnim konceptima i modelima kako bi pojednostavili složene probleme. To olakšava razmišljanje o programu na višem nivou i razdvaja tehničke detalje od glavne funkcionalnosti.

2.2. Programski jezici visokoga i niskog nivoa

Programski se **jezici** mogu podijeliti na dvije relativne kategorije: na **jezike visokog nivoa** (eng. *high-level languages*) i **jezike niskog nivoa** (eng. *low-level languages*), pri čemu svaka kategorija nudi različite razine apstrakcije i kontrole nad operacijom računala. Ta podjela opisuje koliko je funkcionalnost pojedinog jezika odvojena od specifičnih operacija računalnog hardvera, kao što su upravljanje memorijom i procesorom. **Jezici niskog nivoa** „bliže“ su hardveru računala, što omogućuje preciznije upravljanje njime. S druge strane, **jezici visokog nivoa** „dalje“ su od hardvera računala, čime su svojom sintaksom bliže ljudskim jezicima. Podjela na jezike visokoga i niskog nivoa nije vrijednosna, odnosno jedna kategorija nije univerzalno bolja od druge, nego imaju različite snage i slabosti.

Slijedi kratko pojašnjenje razlike između ovih dviju vrsta programskih jezika.

Jezici visokog nivoa:

- dizajnirani su da bi bili jednostavni za programere i lakši za razumijevanje
- programer se manje brine o hardverskim detaljima računala, poput upravljanja memorijom
- koriste se apstraktnim naredbama i konceptima bližim ljudskom jeziku
- omogućavaju brži razvoj aplikacija jer pružaju više gotovih komponenti i resursa za rješavanje uobičajenih problema
- **primjeri jezika** visokog nivoa uključuju ***C#*, *JavaScript* i *Python***.

Jezici niskog nivoa:

- pružaju veću kontrolu nad hardverom računala
- programer je odgovoran za upravljanje memorijom
- bliži su strojnom jeziku računala i omogućuju direktno programiranje hardvera
- često se koriste u situacijama gdje je potrebna maksimalna učinkovitost i brzina računala, kao što je razvoj operativnih sustava ili sustava za ugrađene uređaje
- **primjeri jezika** niskog nivoa uključuju ***C*** i ***C++***.

Ukratko, razlika između jezika niskoga i visokog nivoa leži u razini apstrakcije i kontrole nad računalom. Jezici visokog nivoa olakšavaju razvoj aplikacija, dok jezici niskog nivoa omogućavaju dublju kontrolu i optimizaciju, ali su često zahtjevniji za programiranje.

2.3. Osnovne karakteristike programskog jezika C#

C# je jezik visokog nivoa koji je razvio *Microsoft*, a koristi se u razvojnom alatu *Unity*. ***Unity*** je aplikacija za razvoj videoigara i aplikacija alternativne stvarnosti, koja se obrađuje u kasnijim poglavljima ovog priručnika.

Osnovne su karakteristike programskog jezika *C#*:

- **jednostavnost korištenja:** *C#* je dizajniran kako bi bio jednostavan za učenje i korištenje. Sintaksa je relativno čitljiva i slična prirodnom jeziku, čime se olakšava pisanje koda
- **objektno orijentirano programiranje:** *C#* podržava koncepte objektno orijentiranog programiranja (OOP), omogućavajući programerima svrstavanje koda u objekte i klase radi bolje organizacije i ponovne upotrebe

- **automatsko prikupljanje smeća:** *C#* ima ugrađeno automatsko prikupljanje smeća (eng. *garbage collection*) koje olakšava upravljanje memorijom. Programerima nije potrebno ručno oslobađati memoriju
- **integracija s razvojnim okruženjem *Visual Studio*:** programeri često koriste ***Microsoft Visual Studio***, moćno razvojno okruženje za pisanje i testiranje *C#* aplikacija.

C# omogućava programerima usmjeravanje na razvoj aplikacija umjesto na suočavanje s detaljima niskog nivoa i upravljanje memorijom. To čini *C#* atraktivnim izborom za razvoj aplikacija, posebno u slučajevima bavljenja aplikacijama za *Windows* i mrežu.

2.3.1. Upravljanje memorijom i automatsko prikupljanje smeća

Upravljanje memorijom u programiranju proces je nadziranja i raspodjele radne memorije računala tijekom izvođenja programa. To je ključno jer računala imaju ograničenu količinu radne memorije, a nepravilno upravljanje može dovesti do ozbiljnih problema, uključujući **prekoračenje memorije** (eng. *memory overflow)* i **curenje memorije** (eng. *memory leak*).

U jezicima niskog nivoa potrebno je eksplicitno upravljati radnom memorijom računala. To znači da se za svaki podatak koji program pamti tijekom svoje provedbe, mora dodijeliti dio računalne memorije, i naknadno ga osloboditi nakon što taj podatak više nije potreban, kako bi se stvorilo prostora u memoriji za druge podatke. Taj se **proces** naziva **prikupljanje smeća** (eng. *garbage collection*).

C# je jezik s automatskim prikupljanjem smeća, što znači da se prilikom programiranja u njemu, u većini slučajeva, programer ne mora brinuti o oslobađanju memorije, te će *C#* sam osloboditi memoriju kada neki podatak više nije potreban za operaciju programa.

2.3.2. Kompajliranje

Računala ne mogu direktno razumjeti i izvršavati kod napisan u programskom jeziku. Prije nego što se kod napisan u programskog jeziku može provesti, najprije ga je potrebno prevesti u strojni jezik koji računalo razumije.

Kompajliranje (eng. *compiling*) jest proces prevođenja koda napisanog u programskom jeziku na strojni jezik koji računalo može razumjeti i izvršiti. Program koji

provodi kompajliranje zovemo **kompajlerom** (eng. *compiler)*. Moderna razvojna sučelja (poput *Microsoftova Visual Studia*) imaju ugrađeni kompajler, a što programerima omogućuje pisanje koda u istom programu u kojem ga kompajlira. Ukratko, kompajler uzima **izvorni kod**, odnosno tekst napisan na određenom programskom jeziku i prevodi ga u **izvršni kod,** odnosno niz naredbi koje računalo može izvršiti kako bi postiglo određeni cilj. Prilikom kompajliranja, kompajler provjerava sintaksu i semantiku izvornog koda kako bi utvrdio nepostojanje grešaka. To uključuje greške poput tipografskih, sintaksnih pogreški i mnogih drugih.

Izvorni kod koji ima greške, kompajler ne može prevesti u izvršni kod.

Proces pronalaženja i ispravljanja grešaka u izvornom kodu naziva se **debugiranje** (eng. *debugging*).

2.4. Integrirana razvojna okruženja

IDE (*Integrated Development Environment*) ili **integrirano razvojno okruženje** softverski je alat koji programerima pomaže u razvoju aplikacija.

Slijedi kratki popis funkcija koje integrirano razvojno sučelje (IDE) pruža.

- **Pisanje koda:** IDE programerima omogućuje okruženje za jednostavno i organizirano pisanje te uređivanje izvornog koda. Sintaksa programskog jezika podržanog u IDE-u obično se ispisuje u specifičnim bojama za lakše čitanje i pisanje koda.
- **Debagiranje (eng. *debugging*):** IDE nudi alate za debagiranje, a što znači pronalaženje i ispravljanje grešaka u kodu.
- **Kompajliranje:** IDE-ovi imaju integrirane kompajlere, što olakšava proces izgradnje (eng. *build*) i kompajliranja programa.
- **Integracija s alatima:** IDE-ovi su integrirani s raznim softverskim alatima, a što programerima omogućava pristup dodacima, sustavima za verzioniranje koda, upraviteljima paketa te drugim resursima direktno iz razvojnog okruženja.
- **Pomoć i savjetovanje:** IDE-ovi često nude automatsko popunjavanje koda, prijedloge i pomoć u dokumentaciji programskog jezika kako bi programerima bilo omogućeno brže i lakše pisanje ispravanog koda.

- **Upravljanje projektima:** IDE-ovi pomažu programerima u organizaciji projekata i datoteka. Omogućavaju stvaranje, uređivanje i upravljanje projektima na jednom mjestu.

IDE je neizostavan alat programerima jer olakšava cijeli proces razvoja softverskih aplikacija, od pisanja izvornog koda do testiranja i isporuke gotovog proizvoda. Osim toga, IDE pruža alate za učinkovito rješavanje problema.

Visual Studio IDE

Visual Studio Community besplatna je verzija integriranog razvojnog sučelja *Visual Studija* koja podržava razvoj aplikacija u *C#*-u i integraciju s *Unity* platformom za razvoj igara.

PREUZIMANJE *VISUAL STUDIO COMMUNITYJA*

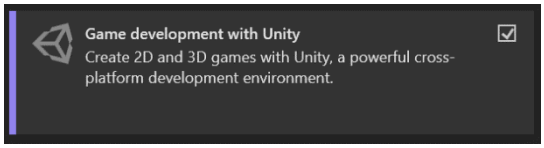
Prije instalacije *Visual Studija*, potrebno ga je preuzeti slijedeći u nastavku opisane korake:

1. korak: otvoriti mrežni preglednik i posjetiti službenu stranicu *Visual Studija*: https://visualstudio.microsoft.com/downloads/
2. korak: kliknuti na *Free download* ispod oznake *Visual Studio Community*
3. korak: preuzimanje će automatski započeti.

Instalacija *Visual Studio Communityja*

Nakon preuzimanja instalacijskog paketa za instalaciju *Visual Studio Communityja* potrebno je slijediti naredne korake:

1. korak: kliknuti dva puta na preuzeti instalacijski paket
2. korak: nakon pojave prozora za instalaciju, kliknuti na *Continue*
3. korak: slijediti upute na ekranu. Tijekom instalacije zatražit će se odabir komponenti koje se želi instalirati. **Pažnja:** za razvoj aplikacija u *C#*-u i *Unityju*, potrebno je osigurati da su odabrane sljedeće komponente:



Slika 1. Komponenta 1 (slika ekrana, Visual Studio)

4.

- *Universal Windows Platform development*
- *Game development with Unity*

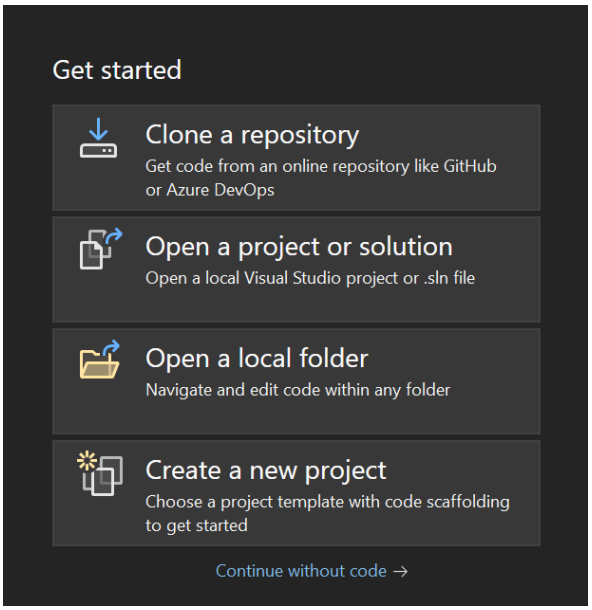
Slika 2. Komponenta 2 (slika ekrana, Visual Studio)

5. korak: nakon odabira željenih komponenti, kliknuti na gumb *Install* i pričekati dok se instalacija ne dovrši.

Korištenje *Visual Studio IDE-ja*

Nakon dovršetka instalacije, moguće je pokrenuti *Visual Studio* i početi s razvojem aplikacija.

Koraci za pokretanje *Visual Studija*



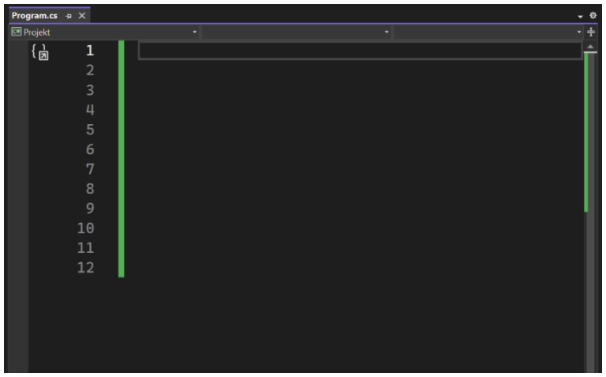
Slika 3. Početne opcije (slika ekrana, Visual Studio)

1. korak: dvaput kliknuti ikonu *Visual Studio Community* u radnom području ili u izborniku *Start*.
2. korak: prilikom prvog pokretanja bit će zatražena prijava s *Microsoftovim* računom. U slučaju nepostojanja računa, moguće je besplatno ga kreirati.
3. korak: nakon prijave, moguće je stvoriti novi projekt odabirom *Create a new project*.
4. korak: u popisu dostupnih predložaka, potrebno je odabrati *C# Console App* za jednostavne *C#* projekte.
5. korak: potrebno je slijediti upute na ekranu za konfiguriranje i stvaranje vlastitoga projekta.

> ⚠ **Pažnja**
>
> Prilikom konfiguracije, opcija *Do not use top-level statements* mora biti isključena.

Sučelje *Visual Studija* sastoji se od brojnih komponenata i alata, a u nastavku će biti nabrojene najbitnije.

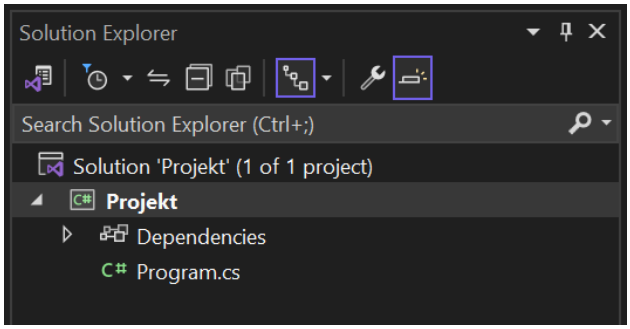
Radno područje (eng. *Workspace*)



Radno područje glavni je prostor u kojem se kod piše i uređuje. Bojanje sintakse podržano je u tekstualnom uređivaču, što znači da će različiti dijelovi koda biti prikazani različitim bojama, olakšavajući čitanje i razumijevanje.

Slika 4. Radno područje (slika ekrana, Visual Studio)

Rješenje (eng. *Solution*) i preglednik projekta (eng. *Solution Explorer*)

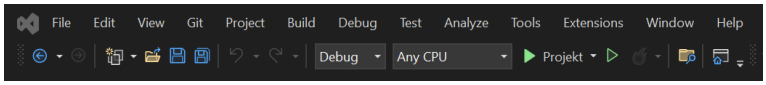


S desne strane sučelja nalazi se *Solution Explorer* u kojem je moguće vidjeti strukturu projekta, odnosno datoteke koje čine projekt.

Slika 5. Preglednik projekta (slika ekrana, Visual Studio)

Alatna traka (eng. *Toolbar*)

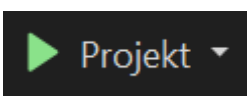
Na vrhu sučelja nalazi se alatna traka koja sadrži razne gumbe i opcije za brz pristup ključnim funkcijama *Visual Studija*.



Slika 6. Alatna traka (slika ekrana, Visual Studio)

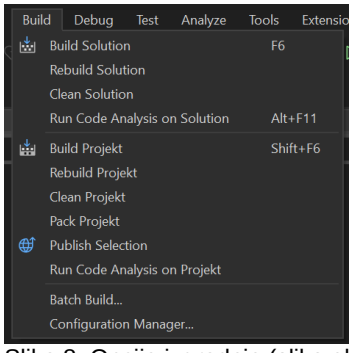
Gumb za reprodukciju (eng. *Play button*)

Označen zelenom strelicom, taj gumb omogućava pokretanje aplikacije. Kada se na nj klikne, kod se kompajlira i pokreće.

Tipka prečaca: **F5** 

Slika 7. Gumb za reprodukciju (slika ekrana, Visual Studio)

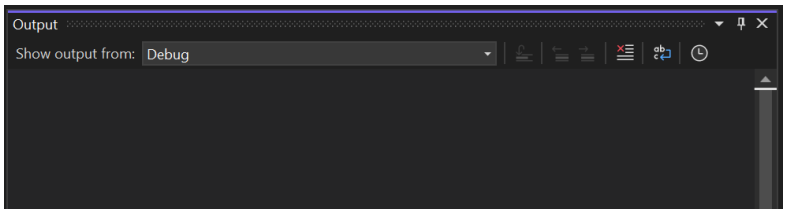
Izgradnja (eng. *Build*)



Opcija *Build* (hrv. „izgradi“) omogućuje kompajliranje koda bez pokretanja. Pogreške u kodu mogu se time provjeriti bez pokretanja same aplikacije.

Slika 8. Opcije izgradnje (slika ekrana, Visual Studio)

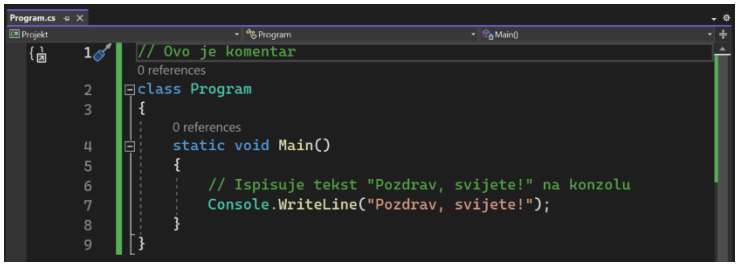
Izlazni prozor (eng. *Output Window*)



Slika 9. Izlazni prozor (slika ekrana, Visual Studio)

Na dnu sučelja može se pronaći izlazni prozor, a u kojem se prikazuju poruke tijekom kompajliranja, upozorenja, pogreške i druge informacije o projektu.

Osnovne sintakse programskog jezika *C#*



Slika 10. Primjer programa (slika ekrana, Visual Studio)

U nastavku su objašnjeni osnovni koncepti sintakse jezika *C#* pomoću jednostavnoga primjera koda napisanog u radnoj površini *Visual Studija*.

2.4.1. Instrukcije

Instrukcija je osnovna jedinica rada koju program obavlja. U gornjem primjeru ***Console.WriteLine("Pozdrav, svijete!");*** instrukcija je koja daje naredbu programu da ispiše određeni tekst na konzolu.

Svaka instrukcija u jeziku *C#* završava točkom sa zarezom (;). Kako bi kod bio čitak, programerska je konvencija da se svaka instrukcija piše u jednom redu (u jednoj liniji koda) te da svaki red koda može sadržavati samo jednu instrukciju. U većini programskih jezika, uključujući *C#*, **instrukcije se izvršavaju po redu,** jedna za drugom, s vrha prema dolje. To znači da instrukcije koje se nalaze više u kodu (tj. bliže početku) imaju prioritet i izvršavaju se prije instrukcija koje slijede ispod njih.

2.4.2. Komentari

U jeziku *C#* sve što slijedi nakon dvostrukih kosih crta **(*//*)**, tretira se kao komentar i ignorira se tijekom izvođenja koda.

2.4.3. Zagrade

U jeziku *C#* koriste se različite vrste zagrada, svaka s vlastitom funkcijom, koje su podrobnije opisane u ostatku priručnika. Slijedi kratki pregled tipova zagrada koje se koriste u C#-u.

- **Vitičaste zagrade { }**: koriste se za definiranje bloka koda.
    - Blok koda grupa je instrukcija koje predstavljaju neku odvojenu funkcionalnu cjelinu. U gornjem primjeru, sve unutar vitičastih zagrada nakon ***class Program*** definira sadržaj klase (poglavlje 2, *Objektno orijentirano programiranje*).
- **Okrugle zagrade ( )**: koriste se za grupiranje i definiranje parametara u funkcijama, kao u primjeru ***Console.WriteLine("Pozdrav, svijete!")***.
- **Uglate zagrade [ ]**: koriste se primarno pri stvaranju kolekcija i tijekom pristupanja elementima kolekcije.

Važno je napomenuti da svaka otvorena zagrada mora biti zatvorena da bi kod bio ispravno napisan te se izbjegle greške u izvođenju.

2.5. Implementacija i pseudokod

U svijetu programiranja često se susrećemo s pojmovima kao što su **implementacija** i **pseudokod**. Iako su oba termina povezana s procesom stvaranja softvera, oni predstavljaju različite aspekte razvoja. U nastavku je objašnjena osnovna razlika između ovih dvaju pojmova.

2.5.1. Pseudokod

Pseudokod je apstraktni opis procesa koji se koristi za planiranje ili objašnjavanje logike programa. Takav opis zovemo i **algoritmom**. Napisan je na način da ga ljudi lako razumiju, a ne računala. Pseudokod ne slijedi striktna sintaktička pravila kao stvarni programski jezici. Karakteriziraju ga:

- **apstraktnost:** pseudokod je generalan i usredotočen je na logiku rješenja, a ne na specifične detalje implementacije
- **jezik:** nema stroga pravila pisanja. Sličan je prirodnom jeziku s elementima programskog jezika
- **izvršivost:** pseudokod sam po sebi nije izvršiv. Služi kao vodič za stvarnu implementaciju koda.

2.5.2. Implementacija

Implementacija se odnosi na proces pretvaranja dizajna ili specifikacije programa u stvarni, funkcionalni kod koji može biti izvršen na računalu. Implementacija je konkretna i koristi se određenim programskim jezikom, poput *C#*, *Java* ili *Python*. Karakteriziraju je:

- **konkretnost:** implementacija je specifična i detaljna. Opiše točno kako će program raditi
- **jezik:** koristi se stvarni programski jezik koji strojevi mogu interpretirati i izvršavati
- **izvršivost:** nakon implementacije kod može biti kompajliran i izvršen da bi se postigla željena funkcionalnost.

2.5.3. Primjer pseudokoda i njegove implementacije u programskom jeziku C#

Pseudokod:

```
1   Započni program
2       Unesi prvi broj i spremi ga kao BrojA
3       Unesi drugi broj i spremi ga kao BrojB
4       Zbroji BrojA i BrojB i spremi rezultat kao Zbroj
5       Prikaži Zbroj
6   Završi program
```

Implementacija pseudokoda u programskom jeziku *C#*:

```
int BrojA = 2;
int BrojB = 4;
int Zbroj = BrojA + BrojB;
Console.WriteLine(Zbroj);
```

Zaključno, dok implementacija predstavlja konkretne korake napisane u programskom jeziku koji računalo može izvršavati, pseudokod je metoda koja programerima pomaže razraditi i komunicirati logiku programa prije nego što se započne sa stvarnom implementacijom. Pseudokod je most između konceptualne ideje i njezine konkretne realizacije u implementaciji.

2.6. Projekti ConsoleApp u Visual Studiju

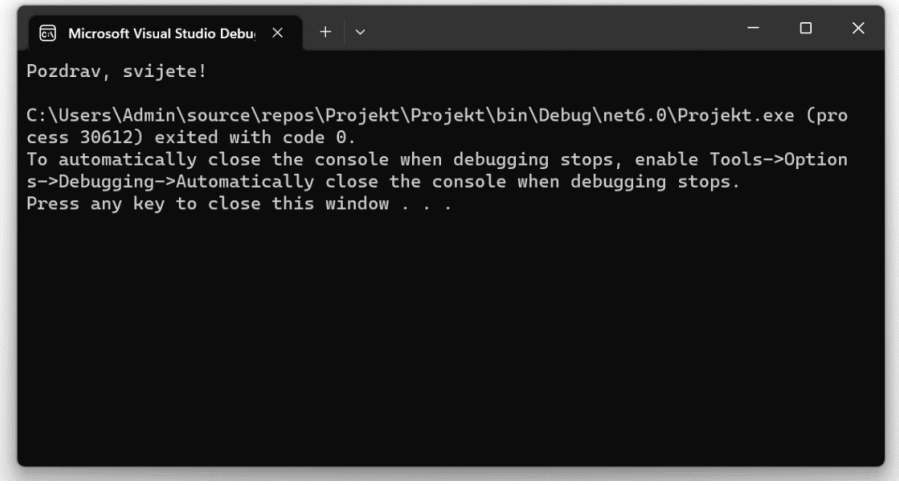
U okruženju *Visual Studija* ***ConsoleApp*** je tip projekta namijenjen izradi konzolnih aplikacija. **Konzolne aplikacije** su osnovni programi koji komuniciraju s korisnikom posredstvom tekstualnog sučelja, tj. konzole. Ne koriste **grafičko korisničko sučelje** (**GUI**), što ih čini idealnim za učenje osnovnih programerskih koncepta i za razne skripte.

Ovo poglavlje se isključivo bavi izradom konzolnih aplikacija.

2.6.1. Konzola

Konzola je tekstualno sučelje kojim korisnik unosi podatke i prima povratne informacije od programa. U kontekstu konzolnih aplikacija u *C#-u*, kada se program pokrene,

otvara se prozor konzole gdje se izvršava kod, a korisnik može vidjeti rezultate ili unositi podatke.



Slika 11. Ispis konzole (slika ekrana, Visual Studio)

Kada se pokrene *ConsoleApp* projekt u *Visual Studiju*, automatski se otvara konzola.

***Console.WriteLine()* funkcija**

Console.WriteLine() je funkcija ugrađena u jezik *C#*, koja programerima omogućuje ispisivanje teksta ili informacija u konzoli. Kada se navedena funkcija pozove, tekst ili podatci upisani u zagrade bit će ispisani u konzoli.

Primjer korištenja *Console.WriteLine()* funkcije:

```
Console.WriteLine("Pozdrav, svijete!");
```

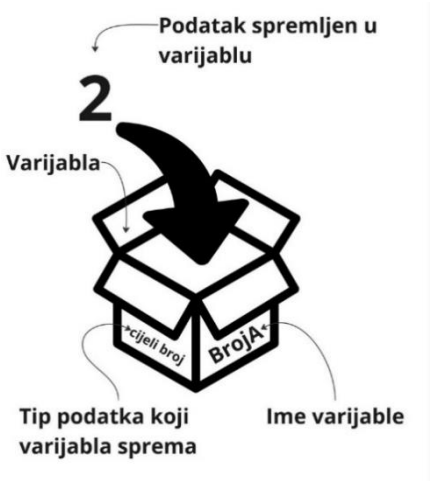
Kada se taj kod izvrši, tekst *"Pozdrav, svijete!"* bit će prikazan u konzoli, kao u prethodnoj slici konzole.

Ukratko, ***ConsoleApp* projekti** u *Visual Studiju* osnovni su projekti koji omogućuju interakciju s korisnikom pomoću konzole. A *Console.WriteLine()*jednostavan je alat koji programerima omogućuje ispisivanje informacija unutar te konzole.

2.7. Varijable

Varijable su spremnici za podatke s kojima programer želi manipulirati, odnosno upravljati. Konceptualno, varijable su spremnici s imenom, koji mogu spremiti određeni tip podataka. Specifična vrijednost spremljena u varijablu može se mijenjati tijekom izvođenja programa, tj. ona može varirati, od čega proizlazi termin „varijabla“.

U *C#*-u varijable moraju imati **ime** koje im programer određuje, i moraju imati **određen tip podataka koji mogu pohraniti**.



Slika 12. Dijagram pohrane podataka (Juračić, autorov rad)

2.7.1. Osnovna implementacija varijabli u programskom jeziku C#

U *C#*-u varijable se implementiraju na sljedeći način:

```
int a = 2;
```

U gornjem primjeru „a“ je **ime varijable**, dok je „2“ **vrijednost spremljena u varijablu**. Prva riječ *int* je **deklaracija tipa varijable**, koja definira koju vrstu vrijednosti varijabla „a“ može spremiti. U ovom slučaju varijabla „a“ može spremiti samo cijele brojeve, na što ukazuje skraćenica ***int***, od eng. *integer*, što znači cijeli broj.

Ime varijable može biti proizvoljno, međutim dvije varijable unutar istog konteksta (istoga bloka) ne mogu imati isto ime.

Postoje dva procesa u kreaciji varijable, koje je potrebno razlikovati:

- **deklaracija**
- **inicijalizacija**.

Deklaracija označava proces stvaranja varijable, odnosno spremnika za neki tip vrijednosti, dok je **inicijalizacija** proces u kojem se varijabli prida njezina početna vrijednost. Dakle, tijekom procesa deklaracije spremnik je imenovan, ali još prazan. Tijekom inicijalizacije on se prvi put puni.

U gornjem primjeru (int a = 2;) deklaracija i inicijalizacija istodobno su obavljene, što je tipična praksa, ali se isto moglo postići dvjema linijama koda:

```
int a;          //Deklaracija.
a = 0;          //Inicijalizacija.
```

Nakon što je varijabla inicijalizirana, moguće se koristiti varijablom u drugim dijelovima programa, tako da joj se upiše ime na potrebno mjesto. Prilikom korištenja varijablama, **program se koristi s vrijednosti spremljenom u varijablu**.

Sljedeći primjer pokazuje kako se moguće koristiti varijablom:

```
int a = 2;
Console.WriteLine(a);
```

Konzola neće ispisati ime varijable, nego njezinu vrijednost.

Varijable dobivaju svoju početnu vrijednost tijekom inicijalizacije, ali se njihova vrijednost može promijeniti tijekom provedbe programa. U sljedećem primjeru mijenja se vrijednost varijable nakon njezine inicijalizacije:

```
int a = 2;
a = 4;
Console.WriteLine(a);
```

Bitno je podsjetiti da će iz razloga što se sve instrukcije u programu provode sekvencijalno, odnosno jedna za drugom, konzola ispisati vrijednost 4.

Redoslijed je operacija sljedeći:

1. varijabla „a“ se deklarira i inicijalizira vrijednošću 2
2. vrijednost varijable „a“ promijeni se u 4
3. funkcija *Console.WriteLine()* ispisuje trenutnu vrijednost varijable „a“ u konzolu.

2.7.2. Načini pohrane vrijednosti u varijable: literali i izrazi

U gornjim primjerima unesena je vrijednosti u *int* varijable tako da je napisan konkretan cijeli broj koji se želi pohraniti. Vrijednost pohranjena na takav način naziva se **literalom** (eng. *literal*), što znači „doslovno“.

Literali su konstante koje direktno predstavljaju određene vrijednosti, kao što su brojevi, znakovi ili nizovi znakova, ovisno o tipu podatka koji se pohranjuje u varijablu. U programskom jeziku *C#* literali se koriste za inicijalizaciju varijabli.

Primjer inicijalizacije literalom:

```
int a = 2;      //Inicijalizacija literalom.
```

Osim što je unos vrijednosti u varijablu moguće napraviti s pomoću literala, može se obaviti i s pomoću izraza. **Izraz** u programiranju označava termin koji je usko srodan matematičkim izrazima (poput 2 + 2 ili $^{9}/_{3}$) te predstavlja kombinaciju literala, varijabli i operatora, koja se izračunava da bi se dobio rezultat.

Primjer korištenja izraza:

```
int a = 2;
int b = 3;
int c = a + b;//Inicijalizacija izrazom.
```

Varijable „a“ i „b“ su inicijalizirane literalima, dok je vrijednost „c“ inicijalizirana izrazom. Vrijednost „c“ će u trenutku njezine inicijalizacije biti 5, odnosno zbroj trenutnih vrijednosti varijabli „a“ i „b“.

Bitno je razumjeti da se **izrazi evaluiraju samo u instrukciji u kojoj su implementirani**. Drugim riječima, izrazi ne opisuju neki konstantni odnos između dviju varijabli. Na primjer:

```
int a = 2;
int b = 3;
int c = a + b;
a = 6;
b = 10;
Console.WriteLine(c);
```

Ako se pokrene program, bit će moguće primijetiti da je vrijednost „c“ ostala 5, iako su u međuvremenu vrijednosti „a“ i „b“ promijenjene. Navedeno nastaje iz razloga što je izraz koji određuje njezinu vrijednost (c = a + b) evaluiran prije nego što su varijable „a“ i „b“ promijenjene.

2.8. Tipovi podataka u programskom jeziku C#

Varijable mogu pohraniti razne vrste podataka te je ranije u poglavlju prikazano nekoliko primjera koristeći se cijelim brojevima u varijablama s tipom *int*. U ovom poglavlju bit će objašnjeno što „tip“ u *C#* programskom kontekstu znači i kako se koristiti najučestalijim tipovima podataka.

Tip (eng. *type*) pojam je koji predstavlja vrstu podatkovne vrijednosti koja se može pohraniti u varijablu. **Nativni** (ili ugrađeni) **tipovi podataka** jesu tipovi koji su predefinirani unutar programskog jezika *C#* i predstavljaju osnovne građevne blokove za pohranu i manipulaciju podatcima.

U nastavku je prikazan pregled glavnine nativnih tipova podataka u *C#*-u.

1. **Cijeli brojevi (eng. *Integral Types*):**
    - **int:** koristi se za pohranu 32-bitnih cjelobrojnih vrijednosti
    - **long:** koristi se za pohranu 64-bitnih cjelobrojnih vrijednosti
    - **short:** koristi se za pohranu 16-bitnih cjelobrojnih vrijednosti
    - **byte:** koristi se za pohranu 8-bitnih cjelobrojnih vrijednosti.
2. **Tipovi podataka s pomičnim zarezom (eng. *Floating-Point Types*):**
    - **float:** koristi se za pohranu decimalnih brojeva s jednostrukom preciznošću
    - **double:** koristi se za pohranu decimalnih brojeva s dvostrukom preciznošću (većom od *float*)
    - **decimal:** koristi se za pohranu decimalnih brojeva s visokom preciznošću, često se koristi u financijskim aplikacijama.
3. **Znakovni tipovi (eng. *Character Types*):**
    - **char:** koristi se za pohranu pojedinačnih znakova *Unicode* znakovnog skupa.
4. **Logički tip (eng. *Boolean Type*):**
    - **bool:** koristi se za pohranu istinitih vrijednosti *true* ili *false*.
5. **Tipovi podataka za nizove znakova (eng. *String Types*):**
    - **string:** Koristi se za pohranu niza znakova.
6. **Tip podataka za datum i vrijeme (eng. *DateTime Type*):**
    - **DateTime:** koristi se za pohranu datumskih i vremenskih informacija.
7. **Tipovi podataka za enumeracije (eng. *Enum Types*):**

- **enum:** koristi se za definiranje skupa naziva vrijednosti kojima je pridružena numerička vrijednost.

Osim navedenih osnovnih nativnih tipova podataka, programski jezik *C#* također omogućuje stvaranje vlastitih tipova podataka pomoću **struktura** (eng. *struct*) i **klasa** (eng. *class*). Formiranje je vlastitih tipova pobliže objašnjeno u poglavlju *2. Objektno orijentirano programiranje*.
Od ranije spomenutih, pobliže će biti razložene funkcionalnosti tipova *int*, *float*, *string* i *bool*, čije je razumijevanja ključno za izradu većine kompleksnijih aplikacija.

2.8.1. Int i float, numerički tipovi

Int je numerički tip, koji je korišten u prijašnjim primjerima unutar ovog poglavlja, a koji služi za pohranu cijelih brojeva, odnosno brojeva bez decimalnih mjesta. *Int* je skraćenica od eng. *integer*, što znači cijeli broj. ***Float*** je numerički tip koji se koristi za pohranu brojeva s decimalnim mjestima, odnosno cijelih i decimalnih brojeva. *Float* je skraćeno od eng. *floating point number* – broj s plutajućom točkom. *Float* se koristi za reprezentiranje brojeva koji mogu imati decimalnu vrijednost.

Float varijable se deklariraju i inicijaliziraju na sljedeći način:

```
float f1 = 0.3f;
float f2 = 3.5f;
float f3 = 5f;
```

Kao što je vidljivo, literali *float* vrijednosti završavaju slovom „f“ kako bi se jasno razlikovali od istih vrijednosti tipa *int*. Ta distinkcija je veoma bitna zato što **logika matematičkih izraza u programskom jeziku *C#* ovisi o tipu podatka koji se koristi u pojedinom izrazu**.

Na primjer:

```
float rezultat = 3f / 4f;
Console.WriteLine(rezultat);
```

Izraz koja inicijalizira „rezultat“ dijeljenja je *float* literala 3 s 4, što se prepoznaje po sufiksu „f“ pored znamenki. Vrijednost varijable „rezultat“ ako se pokrene program, iznosi 0.75, što se i očekuje.

Promotrimo što se dogodi ako se program promijeni na sljedeći način:

```
float rezultat = 3 / 4;
Console.WriteLine (rezultat);
```

U posljednjem slučaju vidljivo je da je vrijednost varijable „rezultat“ 0. Zašto? Varijabla „rezultat“ tipa je *float,* i time sposobna pohraniti decimalnu vrijednost, ali izraz u njezinoj inicijalizaciji koristi se brojevima tipa *int* (nemaju sufiks „f“ koji označava *float* literale).

Iz razloga što je operator dijeljenja između dvaju *int* brojeva, on provodi operaciju dijeljenja namijenjenu *int* brojevima. U operaciji dijeljenja dvaju *int* brojeva rezultat je *int* broj, odnosno cijeli broj. **Svaki operator u izrazima u programskom jeziku *C#* provodi kontekstualno uvjetovanu operaciju, koja ovisi o tipovima elemenata operacije.** Dakle, u tom primjeru operator dijeljenja (/) provjerava tip vrijednosti sa svoje lijeve i desne strane i ovisno o tome producira rezultat određenog tipa. Primjerice, *int* / *int* daje *int* rezultat, *int* / *float* ili *float* / *int* daju *float* rezultat.

Primjetno je kako je u gornjem primjeru varijabla „rezultat“ za svoju vrijednost dobila *int* 0. Dakle, varijabla je jednog tipa pohranila vrijednost drugog tipa. Dogodio se primjer **implicitnog prebacivanja** (eng. *implicit cast*). **Prebacivanje je proces u kojem se vrijednost jednog tipa pretvara u vrijednost drugog tipa kako bi se neka operacija mogla provesti.** U tom implicitnom prebacivanju program je tijekom kompajliranja prebacio vrijednost *int* u ekvivalentnu vrijednost *float* da bi je pohranio u varijablu „rezultat“.

Prebacivanje je implicitno kada god programeri ne zatraže eksplicitno da program pretvori jedan tip vrijednosti u drugi. Implicitna prebacivanja se događaju u ograničenom broju slučajeva kada je operacija nedvojbena i samorazumljiva. Primjerice, *int* brojevi se uvijek mogu implicitno prebaciti u *float* brojeve, ali ne i

suprotno. Postoje brojne situacije u kojima programeri žele ili trebaju **eksplicitno prebaciti** jedan tip vrijednosti u drugi.

Primjerice, neka postoje dvije *int* varijable te je potrebno naći rezultat njihova dijeljenja u *float* formatu:

```
int brojŽenskihGostiju = 34;
int ukupniBrojGostiju = 42;
float postotakŽenskihGostiju = brojŽenskihGostiju / ukupniBrojGostiju;
Console.WriteLine(postotakŽenskihGostiju);
```

Navedeni se primjer koristi konceptima iz stvarnosti, ali je ekvivalentan gornjem primjeru, te ako se pokrene program, moguće je vidjeti da je varijabla „postotakŽenskihGostiju“ pohranila vrijednost 0. Problem je u tome što su varijable „brojŽenskihGostiju“ i „ukupniBrojGostiju“ *int* tipa (što ima smisla), međutim želja je da ih operator dijeljenja (/) tretira kao *float* tipove. Stoga, ih treba eksplicitno prebaciti.

Da bi se to napravilo, potrebno je treću liniju modificirati na sljedeći način:

```
float postotakŽenskihGostiju = (float)brojŽenskihGostiju / (float)ukupniBrojGostiju;
```

Sada su eksplicitno prebačene *int* vrijednosti u *float* vrijednosti, a što se u slijedu operacija događa prije nego što se evaluira izraz. Stoga će se operator dijeljenja ponašati na željeni način. Općenito, sintaksa za eksplicitno prebacivanje ista je kao u gornjem primjeru. Ispred varijable u zagradi potrebno je napisati tip u koji se želi prebaciti vrijednost varijable:

```
(noviTip)imeVarijable
```

Nisu sva eksplicitna prebacivanja moguća te je potrebno toga biti svjestan prilikom planiranja programa.

U zaključku, *int* i *float* su numerički tipovi u programskom jeziku *C#* prilagođeni različitim potrebama. *Int* se koristi za pohranu cijelih brojeva bez decimalnih mjesta, dok se *float* koristi za pohranu decimalnih brojeva s pomičnim zarezom i omogućuje decimalnu aritmetiku. Odabir između tih dvaju tipova ovisi o konkretnim potrebama programa.

2.8.2. String

String je tip podataka koji se koristi za pohranu tekstualnih podataka. To je niz znakova koji se može sastojati od slova, tipografskih znakova, brojeva, simbola i razmaka. Ranije je u poglavlju već korišten ***string* literal** u prvom primjeru korištenja funkcije ***Console.WriteLine()***:

```
Console.WriteLine("Zdravo, svijete!");
```

Riječi u zagradi primjer su *string* literala. Kao i sve druge tipove, moguće je *string* vrijednosti pohraniti u varijablu:

```
string punoIme = "Petar Perić";
```

Kako bi niz riječi programski jezik prepoznao kao *string* literal, potrebno ih je smjestiti između dvostrukih (“) ili jednostrukih navodnika (‘).

Kao što numeričke varijable mogu pohraniti rezultat izraza, tako i *string* varijable mogu pohraniti rezultat izraza:

```
string ime = "Petar";
string prezime = "Perić";
string punoIme = ime + " " + prezime;
```

U primjeru vrijednost varijable „punoIme“ jest „Petar Perić“, ali dvije stvari važno je primijetiti:

1. kao što je u ranijem primjeru s *int* i *float* tipovima funkcija operatora dijeljenja (/) ovisila o numeričkom tipu broja, ovdje se funkcija operatora zbrajanja prilagodila tipu *string*. Umjesto da operacija pokuša zbrojiti vrijednosti (što nema smisla kada govorimo o nizovima znakova), ona je spojila odvojene *stringove* u jedan zajednički *string*. Takva operacija se tipično zove **konkatenacija**. Operator zbrajanja (+) obavlja zbrajanje u kontekstu brojeva i konkatenaciju u kontekstu *stringova*

2. izraz se sastoji od dviju *string* varijabli („ime“ i „prezime“) i jednog *string* literala “ “, koji se sastoji samo od jednog razmaka.

Kao što je vidljivo iz primjera s numeričkim tipovima, **moguće je napisati izraz koji se koristi s više od jednog tipa** te se time izlaže riziku da, u slučaju smanjene pažnje, operatori unutar izraza kontekstualno izaberu neku neželjenu operaciju. U korištenju *stringova* postoji jedna tipična greška, a vidljiva je iz sljedećeg primjera:

```
int a = 2;
int b = 2;
Console.WriteLine("a + b je: " + a + b);
```

Ispisani rezultat toga programa bit će „a + b je: 22“, zato što se operatori provode sekvencijalno slijeva nadesno. Prvi će „+“ probati provesti operaciju na tipovima *string* („a + b je: “) i *int* (a) te zaključiti da je potrebna operacija konkatenacije i dostaviti novi *string* „a + b je: 2“. Zatim će drugi „+“ uzeti taj *string* i konkatenirati ga sa *int*-om „b“.

Takve se greške mogu izbjeći pažljivim programiranjem, primjerice instanciranjem rezultata u odvojenu varijablu i zatim prenošenjem te varijable u funkciju.

Interpolirani *stringovi* (eng. *interpolated strings*) način su da se unutar *string* literala unose izrazi koje će se evaluirati prve:

```
Console.WriteLine($"Rezultat a + b je: {2 + 2}");
```

Interpolirani se *stringovi* inicijaliziraju dodavanjem prefiksa „$“ ispred navodnika *string* literala. Unutar teksta interpoliranog *stringa* moguće je onda dodati par vitičastih zagrada unutar kojih se mogu napisati izraze koje se žele interpolirati prije nego što se uklope u *string*. Ako se taj gornji program pokrene, dobit će se očekivan rezultat u prvom primjeru.

2.9. Uvjetna logika i logički (bool) tipovi

U dosadašnjim primjerima raspisani su programi imali jednostavnu i jednosmjernu strukturu. Programi su uzimali varijable i obavljali operacije na njima, ali su sve naredbe provedene neovisno o rezultatima. Kompleksniji programi zahtijevaju da se određene operacije obavljaju samo u određenim slučajevima. Kako bi se to postiglo, potrebno se upoznati s uvjetnom logikom i tipom podataka zvanim ***bool***.

U programiranju, ***bool*** ili ***boolean*** tip je podataka koji predstavljaju dvije logičke vrijednosti: istina (eng. *true*) ili laž (eng. *false*). **Logički (*bool*) tipovi podataka** imaju

ključnu ulogu u kontroliranju tijeka izvršavanja programa i u upravljanju logičkim operacijama.

Primjer je dviju inicijalizacija logčke (*bool*) varijable sljedeći:

```
bool b1 = true;
bool b2 = false;
```

Vrijednost „istina“ drži „b1“, a „b2“ drži vrijednost „laž“.

Obje su inicijalizirane logičkim literalima, od kojih postoje samo dva: true i false. Kao i s drugim tipovima podataka i logičke tipove možemo dobiti kao produkt izraza, ali operatori za logičke izraze imaju svojstvenu sintaksu koju je bitno pobliže razumjeti.

2.9.1. Bool izrazi i logički operatori

Logički izrazi sastoje se od posebnih (tzv. logičkih) operatora, koji uspoređuju dvije vrijednosti i daju nazad logičku vrijednost, ili istina – *true*, ili neistina – *false*. Pojedini logički operatori mogu uspoređivati samo određene tipove podataka, a u ovoj će cjelini biti obrađeni najosnovniji logički operatori te primjeri njihova korištenja.

Operatori == i !=

Operatori „==“ i „!=“ koriste se za usporedbu jednakosti između dviju vrijednosti ili objekata (objekti će biti detaljnije objašnjeni u poglavlju *2. Objektno orijentirano programiranje*). Oni se mogu koristiti s bilo kojim tipom podataka.

Operator „==“ daje vrijednost istina ako su vrijednosti jednake:

```
bool b1 = 2 == 2;    //true
bool b2 = 2 == 3;    //false
```

Operator „!=“ daje vrijednost istina ako su vrijednosti različite:

```
bool b1 = 2 != 2;    //false
bool b2 = 2 != 3;    //true
```

Operatori < i >

Operatori se „<“ i „>“ koriste za usporedbu veličine brojeva s obje svoje strane. Oni se koriste isključivo s numeričkim tipovima. Operator „<“ daje vrijednost istina ako je lijevi broj manji od desnog, a operator „>“ suprotno, kao u sljedećim primjerima izraza:

```
bool b1 = 2 < 3;      //true
bool b2 = 2 > 3;      //false
bool b3 = 2 < 2;      //false, zato što lijeva vrijednost nije manja nego jednaka
bool b4 = 2 > 2;      //false,  zato što lijeva vrijednost nije veća nego jednaka
```

Operatori „<“ i „>“ mogu uspoređivati različite numeričke tipove u jednom izrazu, primjerice *int* i *float*:

```
bool b1 = 2 < 3f;      //true
```

Operatori && i ||

Operatori „&&“ i „||“ evaluiraju kombinacije dviju logičkih vrijednosti.

Operatori „&&“ (logičko „i“) i „||“ (logičko „ili“) koriste se za izvršavanje logičkih operacija nad logičkim vrijednostima. Ti operatori evaluiraju parove logičkih vrijednosti i daju nam nazad novu logičku vrijednost. Operatori „&&“ (logičko „i“) daje vrijednost istina samo ako su obje logičke vrijednosti istina:

```
bool b1 = true && true;    //true
bool b2 = true && false;   //false
bool b3 = false && true;   //false
bool b4 = false && false; //false
```

Operator „||“ (logičko „ili“) daje vrijednost istina ako je ijedna od vrijednosti istina, i ako ako su obje vrijednosti istina:

```
bool b1 = true || true;      //true
```

```
bool b2 = true || false;     //true
bool b3 = false || true;     //true
bool b4 = false || false;    //false
```

2.9.2. Uvjetne izjave: if, else, if, else

Na koji način logički tip omogućava korištenje uvjetnom logikom u programima?

Odgovor na to pitanje biti će objašnjen na primjeru jednostavnog programa koji će reći je li određeni broj „a“ veći ili manji od broja „b“. Počinje se s **inicijalizacijama varijabli**:

```
int a = 3;
int b = 2;
bool aJeVeci = a > b;
```

U tom je primjeru vrijednost logičke varijable „aJeVeci“ istina. Zatim, primjerice program treba ispisati frazu „A je veći od B“ ako je to istina. Da bi to bilo postignuto, potrebna je *if* izjava. *If* je engleska riječ „ako“ te indicira da će se određeni blok koda provesti samo ako je određeni uvjet ispunjen.

U svrhu većeg razumijevanja, sljedeće linije koda bit će dodane gornjem primjeru:

```
if (aJeVeci)
{
    Console.WriteLine("A je veći od B.");
}
```

Navedeno je primjer ***if*** izjave. Ukoliko je logička vrijednost u zagradi iza riječi *if* istinita, utoliko će se linije unutar vitičastih zagrada provesti. Linije koda unutar vitičastih zagrada nazivaju se **blokom koda** ili **programskim blokom**, a što je izraz kojim se opisuju sintaktički izolirane sekcije koda, odnosno dijelovi koda odvojeni od okolnog konteksta.

Ako se sada pokrene program, konzola će ispisati „A je veći od B“.

ISPROBAJTE SAMI:

Potrebno je promijeniti vrijednosti „a“ i „b“ tako da je „b“ veći od „a“ te zatim ponovno pokrenuti program. Moguće je primjetiti da se ništa nije ispisalo na konzoli jer je program preskočio operacije unutar zagrada iz razloga što uvjet „aJeVeci“ nije ispunjen.

Što ako je potrebno ipak ispisati određenu poruku, poput „A nije veći od B.“?

Primjer:

```
if (aJeVeci)
{
   Console.WriteLine("A je veći od B.");
}
else
{
   Console.WriteLine("A nije veći od B.");
}
```

To je primjer korištenja ***else*** izjave. *Else*, eng. „onda“ izjava je koja se samo može koristiti iza bloka *if* izjave te će se njezin blok koda provesti samo ako nije ispunjen uvjet *if* izjave. *Else* izjava ne može stajati sama za sebe. Ako se sada pokrene program, kod u *if* bloku će se provesti ako je „aJeVeci“ istina, a kod u *else* bloku provest će se ako nije.

U trenutnom primjeru program točno javlja kada su varijable „a“ i „b“ različitih vrijednosti, međutim neće ukazati kada su „a“ i „b“ isti broj. U trenutnom programu ako su varijable „a“ i „b“ isti broj, konzola će ispisati „A nije veći od B.“ jer je uvjet u *if* zagradi neistinit. Postoje tri stanja usporedbe „a“ i „b“:

1. a je veći od b
2. b je veći od a
3. a i b su jednaki.

Trenutni program ne poznaje razliku između drugoga i trećeg stanja. Moguće je prilagoditi program tako da točno opiše sva tri stanja:

```
int a = 3;
int b = 2;
if (a > b)
{
   Console.WriteLine("A je veći od B.");
}
else if (a == b)
{
   Console.WriteLine("A i B su jednaki.");
}
```

```
else
{
    Console.WriteLine("B je veći od A.");
}
```

Najprije je moguće primijetiti da je u zagradi *if* izjave zamijenjena logička varijabla s logičkim izrazom. Na taj se način osigurava da postoji jedna varijabla manje za praćenje te je uvjet u *if* zagradi eksplicitno napisan, što smanjuje rizik greške. Drugo je moguće primijetiti da je između *if* bloka i *else* bloka dodan novi blok *else if*.

Else if izjava funkcionalno je ekvivalentna *if* izjavi, ali se uvjet *else if* izjave samo provjerava ako nije ispunjen uvjet bloka iznad nje. *Else if* blok se ne može koristiti sam, niti kao početak niza uvjetnih blokova, ali se može koristiti nakon *if* bloka ili nakon drugog *else if* bloka.

U ovom slučaju, program će najprije provjeriti je li uvjet *if* bloka istinit (a > b), a ako nije, provjerit će je li uvjet *else if* bloka istinit (a == b). Ukoliko ni on nije istinit, utoliko će program ući u *else* blok i provesti naredbe u njemu. Bitno je razumjeti da će se u ovakvom nizu blokova samo jedan od njih provesti.

Uvjetne se izjave mogu nalaziti i unutar uvjetnih blokova, primjerice:

```
bool uvjet1 = true;
bool uvjet2 = false;
if (uvjet1)
{
    if (uvjet2) //Provjera uvjet2 će se provesti samo ako je uvjet1 ispunjen.
    {
        // Kod će se u ovom bloku provesti ako su ispunjeni uvjet1 i uvjet2.
    }
    else
    {
        // Kod će se u ovom bloku provesti ako je ispunjen uvjet1, ali ne i uvjet2.
    }
}
```

Na taj se način uvjetni blokovi mogu ugrađivati jedni u druge i formirati kompleksniju uvjetnu logiku. Ugrađivanje uvjetnih blokova također čini kod manje čitkim te se stoga sugerira izbjegavanje zamršenijih oblika takvih struktura.

2.10. Petlje

Petlje su sintaktičke strukture u programiranju koje omogućavaju ponavljanje određenog skupa instrukcija više puta. Predstavljaju ključni element u većini programskih jezika, uključujući i *C#*. Petlje omogućuju automatizaciju ponavljajućih zadataka, što doprinosi pisanju fleksibilnijega i preglednijeg koda. Za demonstraciju funkcionalnosti petlji, potrebno je zamisliti kako se žele ispisati svi produkti određenog cijelog broja i niza brojeva do nekog maksimuma, poput jednog retka u tablici množenja.

Primjerice, zadatak je ispisati produkte broja 3 i niza od 0 do 10 te da se program isprogramira korištenjem do sada obrađenih tehnika:

```
int a = 3;
Console.WriteLine(a * 0);
Console.WriteLine(a * 1);
Console.WriteLine(a * 2);
Console.WriteLine(a * 3);
Console.WriteLine(a * 4);
Console.WriteLine(a * 5);
Console.WriteLine(a * 6);
Console.WriteLine(a * 7);
Console.WriteLine(a * 8);
Console.WriteLine(a * 9);
Console.WriteLine(a * 10);
```

Problemi s takvim rješenjem višestruki su:

1. program je nečitak i time se izlaže riziku od potkradanja pogrešaka
2. ukoliko se pokuša povećati niz s kojim se varijabla „a“ množi, potrebno je dodati novu instrukciju za svaki broj, a što praktično onemogućuje proširivanje niza iznad određene vrijednosti, npr. želimo li ispisati sve produkte broja 3 od 1 do 1000
3. prvi faktor „a“ smješten je u varijablu i time ga je lako modificirati, međutim, drugi faktori su literali koje je potrebno pojedinačno mijenjati.

Da bi navedeni problemi bili riješeni elegantno, može se koristiti *for* petljom. Primjerice:

```
int a = 3;
```

```
for (int i = 0; i <= 10 ; i++)
{
   Console.WriteLine(a * i);
}
```

Taj program producira isti rezultat kao i prijašnji kod, koristeći se funkcionalnošću *for* petlje. Promotrimo sljedeći apstraktni primjer strukture *for* petlje:

```
for (inicijalizacija varijable; provjera uvjeta; modifikacija varijable)
{
   //Programski blok
}
```

Iza riječi *for* (hrv. „za“) u zagradi slijedi niz elementa koje zajedno određuju koliko puta će se programski blok petlje provesti. U nastavku će biti objašnjen svaki element redom:

1. **inicijalizacija varijable:** prva je izjava u nizu **inicijalizacija varijable** te je ta varijabla uvijek tipa *int* i tipično naziva „i“. Prva se izjava provodi jednom na početku petlje. U prvoj se izjavi definira varijabla o kojoj ovisi koliko se puta provodi programski blok petlje
2. **provjera uvjeta:** druga je izjava u nizu **logičkih izraza**. U gornjem primjeru ona je „i <= 10“. Drugim riječima, ona je stina ako je vrijednost varijable „i“ manja ili veća od 10. Taj je logički izraz u funkciji identična kao logički izraz u *if* izjavi. To jest, ako je rezultat logičkog izraza istina, onda će program provesti instrukcije u programskom bloku, a ako je rezultat neistina, onda će program preskočiti programski blok i nastaviti dalje. Druga izjava se obavlja svaki put prije ulaska u programski blok
3. **modifikacija varijable:** treći je element izjava koja modificira varijablu definiranu u prvoj izjavi. U gornjem primjeru varijabla „i“ se povećava za 1, tj. „i = i + 1“. Treća se izjava obavlja nakon svakoga završenog programskog bloka.

Dakle, promotrimo još jednom primjer programa:

```
int a = 3;
for (int i = 0; i <= 10; i++)
{
   Console.WriteLine(a * i);
}
```

Počevši od trenutka kada program stigne do petlje, redoslijed operacija u *for* petlji sljedeći je:

1. inicijalizira se nova varijabla „i“ i prida joj se vrijednost 0 (**int i = 0**). Navedeno se događa samo jednom na početku
2. provjerava se uvjet (**i <= 10;**). Ako je rezultat uvjeta istina, nastavlja se na korak 3, a ako je rezultat neistina, program izlazi iz petlje i nastavlja dalje
3. obavljaju se instrukcije u programskom bloku. Programski se blok može koristiti varijablom inicijaliziranom u koraku 1, te se u primjeru jednom provodi funkcija ***Console.WriteLine****(a * i)*
4. varijabla „i“ se modificira (**i++**). U tom se slučaju vrijednost „“ povećava za 1 i program se vraća na korak 2.

Dakle, programski blok *for* petlje će se, u ovom slučaju, ponoviti 11 puta, odnosno jednom za svaku vrijednost varijable „i“ od 0 do 10.

Petlje su izrazito moćan alat koji omogućava fleksibilno korištenje programom. U ovom primjeru moguće je vidjeti da se promjenom brojki u prvoj i drugoj izjavi u zagradi *for* petlje može modificirati početak i završetak niza:

```
for (int i = 5; i <= 400; i++)
{
   Console.WriteLine(a * i);
}
```

Ta petlja, primjerice, počinje niz brojem 5 (prvi će ispisani rezultat biti 3 * 5), a završava brojem 400. Kada se usporedi jednostavnost navedene prilagodbe, u odnosu na istovrijedan pokušaj prilagodbe prvog primjera, koji se nije koristio petljama, vrijednost petlji postaje očigledna.

Preinakom parametara *for* petlje moguće je niz obrnuti u redoslijedu, kao u sljedećem primjeru:

```
for (int i = 10; i >= 0; i--)
{
```

```
    Console.WriteLine(a * i);
}
```

Umjesto da se varijablu „i“ svakom iteracijom kroz petlju povećava do iznad maksimuma definiranog u drugom parametru, inicijalizira je se maksimalnim brojem te smanjuje za 1 dok ne postane manja od 0.

Prije nego što se nastavi s drugim oblicima petlji, potrebno je proučiti potencijalne rizike pri njihovoj implementaciji.

2.10.1. Rizik u implementaciji petlje – beskonačna petlja

Primarni rizik korištenja petljama proizlazi iz mogućnosti njihove implementacije tako da program nikada ne izađe iz petlje. Drugim riječima, programski blok petlje se ponavlja zauvijek. Takva se petlja naziva **beskonačnom petljom** (eng. *infinite loop*).

POKUŠAJTE SAMI

U donjem primjeru uočite problem:

```
for (int i = 0; i >= 0; i++)
{
    Console.WriteLine(i);
}
```

Taj primjer petlje ima uvjet koji će uvijek biti istinit. Varijabla „i“ počinje jednaka 0 i svakom iteracijom raste za 1, te stoga uvijek ispunjava uvjet **„i >= 0“.**

Kompajler razvojnog sučelja ne može prepoznati da je implementirana beskonačna petlja, stoga ne može upozoriti na nju. Beskonačne će petlje „zaglaviti“ provedbu programa u petlji i operacije nakon petlje nikada se neće provesti. U skladu s time, program se nikada neće dovršiti. Beskonačna petlja vrsta je greške u provedbi programa (eng. *runtime error*), ali ona ne stvara iznimke (eng. *exceptions*) koje bi sučelje naknadno prepoznalo. Stoga su beskonačne petlje oblik „tihih“ grešaka, a koje

je teško odrediti u kodu. Ukoliko program uđe u beskonačnu petlju, utoliko je potrebno pritisnuti tipku zaustavljanja u razvojnom sučelju za prekid procesa.

2.11. While petlja

While petlje sintaktički su jednostavnije od *for* petlji te jednako kao i *for* petlje omogućavaju izvršavanje određenog bloka koda dok je određeni uvjet ispunjen. Njihova je sintaksa sljedeća:

```
while (provjera uvjeta)
{
   //Programski blok
}
```

Kao što je vidljivo, izjava *while* petlje sastoji se od samo jednog elementa: **uvjeta**, to jest, **logičkih izraza**. Ako je uvjet ispunjen, program ulazi u petlju te nakon završavanja petlje provjerava je li uvjet još uvijek ispunjen i ponovno ulazi u nju. Petlja se ponavlja dokle god je uvjet ispunjen.

Primjer *while* petlje:

```
int i = 1;
while (i <= 5)
{
   Console.WriteLine("Broj: " + i);
   i++;
}
Console.WriteLine("Petlja je završila.");
```

U ovom primjeru postoji varijabla „i“ inicijalizirana vrijednošću 1. Dokle god je uvjet „i <= 5“ istinit, petlja će se izvršavati. Svaki put kada se petlja izvrši, ispisuje se vrijednost varijable „i“ te se ona povećava za 1. Petlja će se izvršavati sve dok „i“ ne postane veći od 5. Nakon toga petlja će se zaustaviti, i program će nastaviti s izvođenjem ostatka koda.

Kao što je vidljivo, *while* petlja omogućava ponavljanje određenog bloka koda dok je uvjet ispunjen. Navedeno je korisno u situacijama kada nije unaprijed poznato koliko puta će se petlja izvršiti, već to ovisi o dinamičkim uvjetima u programu.

While petlja često se koristi u situacijama kada se želi izaći iz petlje pod nekoliko različitih nepovezanih ili asimetričnih uvjeta. Izlazak iz bilo koje petlje moguć je direktno iz programskog bloka pomoću odskočnih naredbi, odnosno **odskočnih izjava.**

2.12. Odskočne izjave – continue i break

Odskočne naredbe (eng. *jump statements*) u programskom jeziku *C#* posebne su naredbe za kontrolu redoslijeda izvođenja programa, omogućujući preskakanje ili prekidanje izvršavanja određenih dijelova programskog bloka unutar petlji. Postoje dvije glavne naredbe za skakanje u *C#*.

Naredba *break* (prekid)

Naredba *break* koristi se za trenutno prekidanje izvođenja petlje. Kada se *break* naredba izvrši, program napušta trenutnu petlju i nastavlja izvođenje koda nakon petlje.

Primjer upotrebe naredbe *break* u *for* petlji:

```
for (int i = 1; i <= 10; i++)
{
   if (i == 5)
   {
      Console.WriteLine("Petlja će sada završiti.");

      break; // Prekida petlju ako je „i“ jednak 5.
   }
   Console.WriteLine("Broj: " + i);
}
Console.WriteLine("Petlja je završila. ");
```

Naredba *continue* (nastavi)

Naredba *continue* koristi se za preskakanje ostatka trenutne iteracije petlje i nastavak s idućom iteracijom. To znači da se blok koda ispod *continue* neće izvršiti za trenutnu iteraciju, već će program odmah nastaviti s idućom iteracijom.

Primjer upotrebe naredbe *continue* u petlji:

```
for (int i = 1; i <= 5; i++)
{
```

```
    if (i == 3)
    {
        continue;  // Preskače ostatak iteracije ako je „i“ jednak 5.
    }
    Console.WriteLine("Broj: " + i);
}
```

Prikazana će petlja preskočiti ispisivanje broja 3 iz razloga što će se, u trenutku kada je ispunjen uvjet ugrađenog uvjetnog *if* bloka (i == 3), pozvati *continue* naredba i preskočit će se ostatak iteracijskog bloka *for* petlje. Petlja će tada pokrenuti novu iteraciju, a nakon što provede svoju modifikacijsku izjavu (i++).

Odskočne izjave omogućavaju programerima veću kontrolu nad izvršavanjem programa i često se koriste za kompleksnije upravljanje petljama. Važno ih je koristiti pažljivo da bi se izbjegle beskonačne petlje ili neželjeni preskoci u kodu.

2.13. Kolekcije

Kolekcije su tipovi koji predstavljaju strukture podataka koje se koriste za pohranu i upravljanje više elemenata. Kolekcije, kao što im ime nalaže, tipovi su podataka koje su skupine drugih tipova podataka. Omogućavaju programerima efikasno manipuliranje podatcima, dodavanje, uklanjanje, pretraživanje i mijenjanje elemenata prema njihovim potrebama. U programskom jeziku *C#* postoje različite vrste kolekcija, a u ovoj cjelini bit će obrađene najčešće korištene:

- nizovi (eng. *arrays*)
- liste (eng. *lists*)
- rječnici (eng. *dictionaries*).

Kolekcije s redoslijedom

2.13.1. Nizovi

Nizovi su osnovne kolekcije u programskom jeziku *C#*. Oni predstavljaju fiksni broj elemenata istog tipa podataka (primjerice, niz cijelih brojeva). Deklariraju se s unaprijed definiranim brojem elemenata, i ne mogu dinamički mijenjati broj elemenata od kojih se sastoje. Vrijednost se elemenata u nizu može dinamički mijenjati, međutim broj se elemenata ne može mijenjati. Nizovi su oblik kolekcija s redoslijedom, a što

znači da je redoslijed elemenata u nizu stalan i moguće se koristiti pozicijom elementa u nizu za identifikaciju.

Nizovi se deklariraju na sljedeći način:

int[] brojevi;

U navedenom je primjeru deklariran niz *int* brojeva. Par pravokutnih zagrada nakon tipa *int* označava da je to niz *int* brojeva, na isti način je moguće deklarirati niz *stringova* (ili niz *boolova*).

Veličinu niza, to jest, broj elemenata koji se može spremiti u niz definira se tijekom inicijalizacije:

brojevi = new int[4];

Moguće je primijetiti sintaksu inicijalizacije kojom se do sada nije koristilo. Umjesto da je upisan literal ili izraz koji daje vrijednost varijable, upisana je izjava koja generira niz koji se pohranjuje u varijablu „brojevi“ (new int[4]). Značenje te izjave jest: „stvori novi niz s 4 *int* elementa“. Broj u pravokutnim zagradama definira količinu elemenata sadržanih u nizu. Tijekom inicijalizacije varijable moguće je i definirati koji se točno brojevi spremaju u niz, i to na sljedeći način:

brojevi = new int[4] {3, 45, 65, 4};

Brojevi u vitičastim zagradama pohranit će se u niz i njihov je redoslijed identičan redoslijedu u nizu. U tom primjeru niz „brojevi“ sastoji se redom od brojeva 3, 45, 65 i 4. Ukoliko se tijekom inicijalizacije ne definira vrijednost spremljenih elemenata, utoliko će vrijednosti elemenata biti pretpostavljene vrijednosti njihovog tipa (eng. *default values*).

Pretpostavljene su vrijednosti uobičajenih tipova:

- int: 0
- float: 0f
- bool: false

Nakon što je niz inicijaliziran kao i s drugim tipovima, moguće se njime koristiti. Najosnovnija funkcionalnost kolekcija dohvaćanje je elemenata iz kolekcije. Da bi element iz postojećeg niza bio dohvaćen, potrebno se koristiti sljedećom sintaksom:

ime_niza[broj elementa u nizu]

Broj u pravokutnim zagradama naziva se **indeksom niza**. Indeks niza je broj koji označava poziciju elementa u nizu. Važno je zapamtiti da **indeksi elemenata u nizu počinju brojem 0**. Dakle, prvo je mjesto u nizu indeks 0, drugo je mjesto u nizu indeks

1, i tako dalje. Drugim riječima, niz od 4 elementa počinje indeksom 0 i završava indeksom 3. Isto vrijedi za sve oblike kolekcija s redoslijedom.

Da bi se stiglo dohvatiti prvi element u nizu, u primjeru će biti korištena sljedeća sintaksa:

brojevi[0]

Tako dohvaćeni elementi mogu se koristiti na dva načina.

1. Prvo kao izraz, primjerice kad inicijaliziramo varijablu:

int broj3 = brojevi[3]; //Vrijednost „broj3“ postaje jednaka vrijednosti 3. elementa niza.

2. Drugo, kao samu varijablu, da bismo joj promijenili vrijednost:

brojevi[3] = 2; //3. element niza dobiva vrijednost 2.

Na taj način moguće je promijeniti sadržaj pojedinačnih elemenata u nizu. Osim što je moguće dohvatiti određeni element, moguće je također dohvatiti ukupni broj elemenata u nizu, a pomoću svojstva *Length* (duljina) niza:

brojevi.Length;

Svojstvo *Length* producira *int* vrijednost jednaku broju elemenata i može se koristiti kao izraz za vrijednost, međutim nije je moguće modificirati.

Nizovi su procesorski najefikasniji oblik kolekcija, i programeri im daju prednost u situacijama u kojima je optimizacija softvera prioritet.

2.13.2. Liste

Liste su kolekcije s redoslijedom čija se veličina može dinamički mijenjati. One su u svojoj funkcionalnosti veoma slične nizovima, ali za razliku od nizova njihov se broj elemenata može promijeniti nakon inicijalizacije. Kao i nizovi, liste mogu pohranjivati samo elemente jednog tipa podataka, a često su korištene kad broj elemenata nije poznat unaprijed.

Primjer deklaracije i inicijalizacije liste:

```
List<int> brojevi = new List<int><();
```

Navedeno je lista brojeva. Prilikom inicijalizacije broj elemenata u listi je 0, i za razliku od nizova, nije moguće listi odrediti veličinu unaprijed. Stoga ona ne pohranjuje automatski elemente s pretpostavljenom vrijednošću.

Tip pohranjenih podataka u listi definirana se deklaracijom tipa u uglatim zagradama nakon riječi *List*.

Kao i s nizovima, moguće je inicijalizirati novu listu s pohranjenim elementima:

```
List<int> brojevi = new List<int>() {22, 35, 4};
```

Na isti način na koji se dolazi do elemenata u nizu, dolazi se i do elementa u listi:

```
brojevi[0]
```

Kao i s nizovima, dohvaćeni se elementi mogu koristiti kao izrazi za pohranu u varijable ili kao varijable za promjenu vrijednosti pohranjene u elementu. Ukupni broj elemenata u listi moguće je dohvatiti pomoću svojstva *Count* (duljina) niza:

```
brojevi.Count
```

Potrebno je obratiti pažnju na to da je svojstvo za broj elementa niza *Length*, a broj elemenata liste *Count*.

Funkcije liste

Postoji niz funkcija u programskom jeziku *C#* koje se mogu koristiti s listama kako bi im se modificirao sadržaj. U nastavku će biti opisane najbitnije.

Dodavanje elemenata – *Add()*

Da bi se dodali novi elementi na kraj liste, koristi se funkcijom ***Add()***. Ona omogućuje da se na kraj liste doda novi element određene vrijednosti. *Add()* se koristi na sljedeći način:

```
List<int> brojevi = new List<int>() {22, 35, 4};
brojevi.Add(2);
```

Tom je funkcijom dodan element na kraj liste „brojevi“ s vrijednošću 2. Time je, također, povećan broj elemenata u listi.

Oduzimanje elemenata – *Remove()* i *RemoveAt()*

RemoveAt() omogućava uklanjanje elemenata na određenom indeksu liste. Iz razloga što se lista treba sastojati od neprekinutog niza indeksa, nakon što se ukloni element pomoću *RemoveAt()*, elementi se u kasnijim indeksima pomiču za jednu vrijednost unazad kako bi popunili izbrisani indeks.

Primjer korištenja *RemoveAt()*:

```
List<int> brojevi = new List<int>() {22, 35, 4};
brojevi.RemoveAt(1);
```

RemoveAt() će u tom slučaju izbrisati element pod indeksom 1, koji ima vrijednost 35 (podsjećamo da indeksi počinju brojem 0). Nakon toga će element 2 (vrijednost 4) promijeniti svoj indeks u indeks 1, kako bi „popunio rupu“.

Pokuša li se koristiti funkciju *RemoveAt()* na indeksu koji ne postoji u listi (primjerice, na indeksu koji je veći od najvećega postojećeg indeksa), program će javiti grešku i prekinuti proces. ***Remove()*** omogućuje uklanjanje prvog elementa određene vrijednosti ako element s tom vrijednosti postoji. Funkcija redom traži element s određenom vrijednošću te ukoliko nađe element, utoliko će ga ukloniti. Ostali elementi s većim indeksom smanjit će svoj indeks za 1, jednako kao i u funkciji *RemoveAt()*. Ako funkcija ne nađe element sa zadanom vrijednošću, ništa se neće dogoditi.

Primjer korištenja *Remove()*:

```
List<int> brojevi = new List<int>() {22, 35, 4, 35};
brojevi.Remove(35);
```

U primjeru funkcija *Remove()* traži element s vrijednošću 35, a lista „brojevi“, na kojoj funkcija traži element, ima takva dva elementa: prvi je na indeksu 1, a drugi na indeksu 3. *RemoveAt()* će samo maknuti prvi nađeni element, što u ovom slučaju znači element pod indeksom 1. Drugi element s istom vrijednošću ostat će nepromijenjen.

Uklanjanje svih elemenata – *Clear()*

Želi li se ukloniti sve elemente u listi, moguće je pozvati funkciju *Clear()*. Ona će ukloniti sve elemente iz liste i ostaviti praznu listu. Ta je lista još uvijek inicijalizirana, i moguće se njome koristiti kao i obično.

Primjer korištenja *Clear()*:

```
List<int> brojevi = new List<int>() {22, 35, 4, 35};
brojevi.Clear();
```

Clear() se koristi u slučaju kada postoji lista čija je zadaća privremeno držati neko kolekciju elemenata za obradu.

2.14. Petlje i kolekcije s redoslijedom, iteriranje kroz kolekcije

U programerskoj praksi petlje i kolekcije s redoslijedom (nizovi i liste) usko su povezane jer zajedno formiraju brojne fleksibilne i moćne predloške za rješavanje niza problema. Kroz nekoliko problemskih primjera bit će proučeni načini suradnje tih dviju struktura.

Zamislimo da imamo niz brojeva, i da ih je potrebno ispisati na konzoli jedan za drugim, istim redoslijedom kojim su oni pohranjeni u nizu. Najprije bi bilo potrebno inicijalizirati niz i pohraniti brojeve u elemente:

```
int[] brojevi = new int[5] {43, 5, 234, 8, 32};
```

Nakon što je niz kreiran, moguće se poslužiti *for* petljom za dohvaćanje redom svakoga elementa i ispisa na konzolu:

```
for (int i = 0; i < brojevi.Length; i++)
{
    Console.WriteLine(brojevi[i]);
}
```

Kao što se može vidjeti u primjeru, uvjet *for* petlje kao svoj maksimum uzima duljinu niza (*i* < *brojevi.Length*), a u funkciju *WriteLine()* se dostavlja element niza „brojevi“ na indeksu „i“. Drugim riječima, pri svakom prolasku kroz petlju, „i“ raste za 1, i koristi se vrijednost „i“ kako bi se dohvatio element iz niza „brojevi“. Petlja će proći svaki indeks niza „brojevi“ i omogućiti da se unutar programskog bloka obavljaju operacije s elementom pod tim indeksom. Takva operacija u kojoj se prolazi kroz sve elemente neke kolekcije s redoslijedom, zove se **iteriranje kroz kolekciju.**

Iteriranje kroz kolekcije jedna je od najčešćih operacija koje programeri implementiraju te je vrlo bitno razumjeti je konceptualno, i u praksi.

U programskom jeziku *C#* postoji oblik petlje specifično namijenjen iteriranju kroz kolekciju: *foreach* petlja.

2.14.1. Foreach petlja

Foreach petlja vrsta je petlje koja je specijalizirana za iteriranje kroz kolekcije. Ona je izuzetno korisna jer pojednostavljuje sintaksu prolaska kroz sve elemente u kolekciji. Naime, ne zahtijeva eksplicitno praćenje indeksa kao što je to potrebno s *for* petljom.

Sljedeći primjer koristi *foreach* petlju da postigne isti rezultat kao gornji primjer s *for* petljom.

```
int[] brojevi = new int[5] {43, 5, 234, 8, 32};
foreach (int broj in brojevi)
{
   Console.WriteLine(broj);
}
```

Foreach petlji je za operaciju potrebna kolekcija kroz koju iterira, a u zagradi te *foreach* petlje vidljive su sve sastavnice potrebne za implementaciju *foreach* petlje:

1. ***int*** broj je deklaracija varijable koja će pohranjivati svaki pojedinačni element koji se dohvaća iz kolekcije
2. ***in*** je sastavnica *foreach* sintakse koja odvaja deklaraciju elementa (stavka 1) od kolekcije u kojoj se element nalazi (stavka 3)
3. **brojevi** je kolekcija kroz koju se iterira.

Foreach petlja će proći kroz svoj programski blok jednom za svaki element kolekcije. Pri izlasku iz svakog bloka element u deklariranoj varijabli (u primjeru: *int* broj) promijenit će se u sljedeći element u kolekciji. Kada je petlja prošla kroz sve elemente u kolekciji, program izlazi iz petlje.

Glavna prednost *foreach* petlje njezina je čitkost. *For* petlje se mogu koristiti za razne operacije, stoga je potrebno pažljivo proučiti *for* petlju da bi joj se saznala funkcija, a s obzirom na to da se *foreach* petlje koriste isključivo za iteriranje kroz kolekcije, svrha je očita. U programskom bloku *foreach* petlje je, također, jasno kada se blok koristi s elementom iterirane kolekcije. Element u primjeru s *for* petljom dohvaća se pomoću „brojevi[i]“, dok se u primjeru s *foreach* petljom dohvaća s „broj“.

Glavni je **nedostatak *foreach* petlji** u usporedbi s *for* petljama da su one slabije optimizirane. Ako program jako često iterira kroz neku kolekciju, i prioritet je optimizacija softvera, tada je smisleno koristiti *for* petlju umjesto *foreach* petlje za iteriranje.

2.15. Rječnici – kolekcije bez redoslijeda

Rječnik (eng. *dictionary*) je kolekcija bez redoslijeda i bez predodređenog broja elemenata, a omogućava pohranjivanje elemenata u parovima: jedan je dio para **ključ**, a drugi je dio para **vrijednost**. Ključ je jedinstvena identifikacija podatka, dok je

vrijednost pohranjena informacija pod tim ključem. Svaki ključ mora biti jedinstven, dok vrijednosti ne moraju biti. Korištenje rječnikom korisno je kada se želi brzo pristupiti vrijednostima na temelju njihovih ključeva, a što omogućava učinkovito pretraživanje i manipulaciju podatcima.

Zamislimo da se želi stvoriti popis osoba i njihovih visina u centimetrima. Moguće bi bilo spremiti te podatke u rječnik u kojem je svaki ključ ime osobe, a vrijednost spremljena u ključ – njihova visina.

U nastavku je prikazan primjer inicijalizacije takvog rječnika:

```
Dictionary<string, int> izumiteljiVisina = new Dictionary<string, int>()
```

U uglatim zagradama iza riječi *Dictionary* upisuje se prvo tip ključa i, nakon toga, tip vrijednosti. Kao što se može vidjeti iz primjera, ključevi i vrijednosti mogu biti različitih tipova. U tom slučaju potrebno je napraviti da imena budu ključevi, što znači da će biti korišten tip *string*, a vrijednosti da su brojevi, stoga će biti korišten neki od numeričkih tipova, u ovom slučaju *int*.

Kao i u ostalim kolekcijama, moguće je pri inicijalizaciji odmah definirati elemente rječnika:

```
Dictionary<string, int> izumiteljiVisina = new Dictionary<string, int>()
{
   {"Nikola Tesla", 188},
   {"Thomas Edison", 178},
   {"Henry Ford", 178}
};
```

Kako bi se dohvatio spremljeni element u rječniku, potrebno je u rječnik unijeti jedan od pohranjenih ključeva:

```
izumiteljiVisina["Nikola Tesla"]
```

Kao što je vidljivo, prikazana operacija je analogna nalaženju elemenata u kolekcijama s redoslijedom (nizovi i liste), ali umjesto da se upisuje indeks u pravokutne zagrade, u njih se unosi ključ. Vrijednosti su promjenjive ali ključevi nisu, što znači da se dohvaćenom elementu rječnika može promijeniti vrijednost, ali ne i ključ:

```
izumiteljiVisina["Nikola Tesla"] = 170;
```

Funkcije rječnika

Postoji niz funkcija u programskom jeziku *C#* koje se mogu koristiti s rječnicima za modificiranje sadržaja. U nastavku će biti opisane samo najvažnije.

Dodavanje elemenata – *Add()*

Da bi se dodao novi element u rječnik, koristi se funkcijom *Add()*. Funkcija *Add()* za rječnike zahtijeva unos dvaju parametara. Prvi je ključ novog elementa, a drugi je vrijednost spremljena u ključu.

Na sljedeći način moguće je dodati novi element u rječnik iz gornjeg primjera:

```
izumiteljiVisina.Add("Steve Jobs",  189);
```

Rječnik mora sadržavati jedinstvene ključeve, te stoga nije moguće funkcijom *Add()* dodati element s ključem koji već postoji u rječniku.

Oduzimanje elemenata – *Remove()*

Element iz rječnika moguće je maknuti pomoću funkcije *Remove()*. Parametar je funkcije *Remove()* ključ. Ako ključ postoji u rječniku, element će se izbrisati iz rječnika, ako ključ ne postoji, ništa se neće dogoditi.

Primjer implementacije:

```
izumiteljiVisina.Remove("Steve Jobs");
```

Uklanjanje svih elemenata – *Clear()*

Želi li se ukloniti sve elemente u rječniku, moguće je pozvati funkciju *Clear()*. Ona će ukloniti sve elemente iz rječnika, ali rječnik ostaje inicijaliziran, te se njime može koristiti kao i obično.

Primjer implementacije:

```
izumiteljiVisina.Clear();
```

2.16. Funkcije

Tijekom razvoja aplikacija programeri se svakodnevno nalaze u situacijama potrebe za implementiranjem implementirati istog niza instrukcija na raznim mjestima u programu. Takvi identični nizovi mogu poremetiti čitkost koda i usporavati razvoj aplikacije. Nasreću, moderni programski jezici našli su rješenje za taj svakodnevni problem, a rješenje su **funkcije**.

Funkcije su osnovni građevni blokovi u programiranju, koji omogućavaju organizaciju i ponovnu upotrebu koda. U programskom jeziku *C#* funkcije predstavljaju skup instrukcija koje izvršavaju određeni zadatak kada su pozvane. Funkcije olakšavaju razdvajanje koda na manje, logički povezane dijelove, čime se poboljšava čitljivost, održavanje i skalabilnost aplikacija.

U ovom poglavlju već je bilo doticaja s funkcijama:

```
Console.WriteLine();          // Ispisuje tekst na konzolu.
lista.Add(2);                 // Dodaje element na kraj liste.
rječnik.Clear();              // Briše sve elemente.
```

Kada se program koristi funkcijama, kaže se da je program pozvao funkciju.

Korištene funkcije, integrirane su s programskim jezikom *C#*, te je njihova funkcionalnost unaprijed određena. Međutim, programeri mogu implementirati vlastite funkcije i koristiti se njima u svojem programu.

Kao i varijable, funkcije se moraju deklarirati prije nego što se koriste.

Sintaksa deklaracije funkcije izgleda ovako:

```
PovratniTip NazivFunkcije(Parametri)
{
   // Tijelo funkcije
}
```

Objašnjenja svakog dijela deklaracije funkcije:

1. **povratni tip (eng. *return type*)**: označava tip podataka koji će funkcija vratiti nakon izvršenja. Ako funkcija ne vraća nikakvu vrijednost, koristi se ključna riječ ***void***. Ukoliko funkcija vraća neku vrijednost, utoliko je potrebno koristiti ključnu riječ *return* i nakon nje podatak koji se vraća
2. **naziv funkcije (eng. *function name*)**: označava jedinstveni identifikator koji se koristi za pozivanje funkcije iz drugih dijelova programa. Naziv treba biti deskriptivan i jasno opisati svrhu funkcije
3. **parametri (eng. *parameters*)**: to su ulazne vrijednosti koje funkcija prima da bi izvršila svoj zadatak. Mogu se definirati nula ili više parametara, a svaki parametar sastoji se od tipa i naziva
4. **tijelo funkcije (eng. *function body*)**: sadrži skup instrukcija koje se izvršavaju kada se funkcija pozove. Tu se nalazi logika funkcije.

Primjer deklaracije funkcije koja zbraja dva broja i vraća njihov rezultat:

```
int Zbroj(int broj1, int broj2)
{
   int rezultat = broj1 + broj2;
   return rezultat;
}
```

Povratni je tip navedene funkcije *int*, a njen naziv je „Zbroj()“. Funkcija uzima dva parametra (broj1, broj2), od kojih su oba tipa *int*. Instrukcija ***return* rezultat;** na kraju funkcije vratit će vrijednost varijable „rezultat“.

Primjer korištenja navedene funkcije:

```
int broj = Zbroj(2, 4);
```

Zato što funkcija „Zbroj()“ vraća *int* vrijednost, moguće ju je pozvati da bi se pohranila ta vrijednost u varijablu. Kao što je moguće koristiti izraze u bilo kojem kontekstu u kojem bi mogli koristiti i literal nekog tipa podataka, na isti se način moguće koristiti i funkcijom u kontekstima gdje je njezina povratna vrijednost ispravan tip podataka. Primjerice, moguće je pozvati funkciju kao jedan od parametara poziva druge funkcije.

```
int broj = Zbroj(2, Zbroj(3, 6));
```

U navedenom primjeru najprije će se pozvati funkcija unutar parametara vanjske funkcije. Nakon što unutarnja funkcija vrati svoju vrijednost, vanjska će funkcija uzeti tu vraćenu vrijednost kao svoj drugi parametar. Taj primjer nema praktičnu primjenu, ali prikazuje općenit princip: gdje god je potreban neki tip podataka, funkcija koja ga vraća može se direktno koristiti u tom kontekstu.

2.16.1. Ključne riječi return i void

Ključna riječ *return* koristi se unutar funkcija da bi se vratio određeni rezultat ili vrijednost iz funkcije. **Kada se *return* naiđe unutar funkcije, izvršavanje te funkcije prestaje,** i vrijednost ili izraz koji slijedi ključnu riječ *return,* vraća se pozivatelju funkcije. Vrsta vrijednosti koja se vraća, mora biti kompatibilna s tipom povratne vrijednosti, koji je deklariran za funkciju.

Na primjer, ako je funkcija deklarirana da vraća *int*, tada *return* mora vratiti vrijednost tipa *int*:

```
int Zbroji(int a, int b)
{
   return a + b;
}
```

U ovom primjeru funkcija „Zbroji“ prima dva cijela broja kao argumente i vraća njihov zbroj koristeći ključnu riječ *return*. Kada funkcija u *C#*-u ima povratni tip *void*, to znači

da funkcija ne vraća nikakvu vrijednost. Međutim, ključna riječ *return* može se i dalje koristiti unutar takve funkcije, ali samo za prekid izvršavanja funkcije.

Na primjer:

```
void IspisiPorukuAkoJeBrojVeciOd10(int broj)
{
   if (broj < 10)
   {
      return; // Prekida izvršavanje funkcije
   }
   Console.WriteLine("Broj je parni.");
}
```

U tom primjeru ako naiđe *return*, izvršavanje će se funkcije prekinuti, i neće se nastaviti s daljnjim linijama koda unutar funkcije. Dakle, tu se koristi da se preskoči ispisivanje poruke ako broj nije veći od 10.

PITANJA ZA PONAVLJANJE:

Provjerite svoje znanje odgovorima na sljedeća pitanja:

1. Kako se programski jezici visokog nivoa razlikuju od jezika niskog nivoa?
2. Što je automatsko prikupljanje smeća i zašto je važno u kontekstu upravljanja memorijom u *C#*-u?
3. Koja je glavna svrha *Visual Studio IDE*-ja i koje su njegove ključne komponente?
4. Kako se koristi funkcija *Console.WriteLine()* u *C#*-u?
5. Objasnite razliku između pseudokoda i stvarne implementacije.
6. Iz kojih razloga bi programer koristio pseudokod?
7. Kako definirati varijablu u *C#*-u? Kako pohraniti vrijednosti unutar varijable?
8. Što su numerički tipovi podataka, poput *int* i *float*? Kako se koristi prebacivanje tipova?
9. Kako se logički izrazi koriste u uvjetnoj logici?
10. Navedite primjer jedne uvjetne izjave.

11. Objasnite razliku između *for* petlje i *foreach* petlje.
12. Što je rizik beskonačne petlje i kako ga izbjeći?
13. Koja je razlika između kolekcija s redoslijedom i kolekcija bez redoslijeda?
14. Kako se definira funkcija u *C#*-u?
15. Što znače ključne riječi *return* i *void*?

3. OBJEKTNO ORIJENTIRANO PROGRAMIRANJE

U OVOM POGLAVLJU NAUČIT ĆETE:

- primijeniti osnovne koncepte objektno orijentiranog programiranja (OOP)
- razlikovati osnovne komponente i gradivne elemente OOP-a. Koristiti se sintaktičkim elementima jezika C#
- razlikovati klase i objekte
- kombinirati upotrebu modifikatora pristupa poput public i private
- urediti pristup klasama koristeći se svojstvima
- napisati konstruktore za klase
- razlikovati referentne i vrijednosne tipove
- koristiti se nasljeđivanjem u strukturi vašeg programa
- koristiti se apstraktnim klasama i virtualnim metodama.

Objektno orijentirano programiranje (**OOP**) široko je korištena i popularna paradigma, odnosno metodologija programiranja koja se temelji na konceptu objekata. Ta paradigma nizom različitih funkcionalnosti omogućava programerima strukturiranje koda na način koji olakšava razumijevanje, održavanje i ponovnu uporabu koda. Trenutna praksa razvoja aplikacija alternativne stvarnosti bazira se na idejama objektno orijentiranog programiranja. Alati za razvijanje aplikacija alternativne stvarnosti (poput razvojnog alata *Unity*) zahtijevaju od svojih korisnika razumijevanje objektno orijentirane paradigme.

C# je objektno orijentiran programski jezik namijenjen korištenju u sklopu objektno orijentirane paradigme. *C#* pruža niz funkcionalnosti koje olakšavaju implementiranje objektno orijentirane strukture programa. Te funkcionalnosti uključuju nasljeđivanja, enkapsulaciju i polimorfizam, koje će biti podrobnije opisane u nastavku ovog poglavlja.

Prvo poglavlje udžbenika, ***Uvod u programiranje u C#-u***, nije se doticalo funkcionalnosti objektno orijentiranog programiranja, s ciljem pojednostavljenja konteksta u kojem se obrađuju osnovni principi programiranja. Međutim, za optimalno korištenje programskim jezikom *C#*, programeri moraju biti upoznati s principima objektno orijentiranog programiranja i njihovom primjenom u *C#*-u. Razumijevanje koncepata obrađenih u prvom poglavlju potrebno je radi razumijevanja primjene objektno orijentiranog programiranja u *C#*-u, te će određene teme iz prvog poglavlja (poput tipova i varijabli) biti detaljnije obrađene u ovom poglavlju.
Ovo poglavlje započinje gradivnim elementima objektno orijentiranog programiranja, a zatim se u drugom dijelu nastavlja s dizajnerskim obrascima i ciljevima objektno orijentiranog programiranja.

3.1. Gradivni elementi objektno orijentiranog programiranja

3.1.1. Klase i objekti

Osnovni su gradivni elementi objektno orijentiranog programiranja klase i objekti. **Klasa** je temeljni građevni blok objektno orijentiranog programiranja u *C#*-u. To je apstraktni predložak koji opisuje zajednička svojstva i ponašanja za određeni tip, koji se naziva objektom klase. Klasa definira varijable koje sadržavaju objekt i funkcije, a koje opisuju ponašanje objekata tog tipa. Odnosno, klase su predlošci/šablone za svoje objekte. **Objekt** je instanca klase, odnosno stvarna pojava određenog tipa. Za svaku klasu može postojati neograničen broj objekata.

Primjer deklaracije klase „**Student**“:

```
public class Student
{
    public string Ime;
    public string Prezime;
    public int Dob;

    public void PredstaviSe()
    {
        Console.WriteLine($"Zdravo, ja sam {Ime} {Prezime}! Imam {Dob} godina.");
```

```
    }
}
```

Svaka implementirana klasa definira novi tip podatka kojim se program može služiti. Tom je klasom definiran tip podatka zvan „Student“, te će se u nastavku tim primjerom koristiti da bi se razložili osnovni elementi deklaracije klase.

3.1.2. Elementi deklaracije klase

Deklaracija klase počinje **potpisom klase**, a on uključuje ime klase i određene ključne riječi koje definiraju njezine osobine (primjerice, **modifikatori pristupa**). Nakon potpisa klase slijedi blok koda ograđen vitičastim zagradama u kojem se nalazi sadržaj klase. Taj se blok naziva tijelom klase ili njezinim opsegom.

```
//Potpis klase.
public class Student
{
//Tijelo klase.
}
```

Potpis klase

Public je modifikator pristupa klase. **Modifikator pristupa** dio je deklaracija varijabli, funkcija i klasa u objektno orijentiranom programiranju, koje kontroliraju razinu vidljivosti i dostupnosti izvan njihova deklariranog opsega. Modifikatori pristupa biti će preciznije obrađeni u dijelu ***3.2. Modifikatori pristupa*****.** Međutim, bitno je razumjeti da je klasa s modifikatorom *public* (hrv. „javno“) potpuno vidljiva izvan svojeg opsega deklaracije, odnosno da se klasa može koristiti iz bilo kojeg dijela programa, bez obzira gdje je deklarirana. Primjerice, iako je klasa deklarirana u XY, moći će biti korištena iz XY. Za klasu koja ima modifikator pristupa *public,* kažemo da je javna.

class je ključna riječ koja upućuje da se radi o deklaraciji klase, za razliku od deklaracija varijabli ili funkcija.

„**Student**“ je ime klase.

Tijelo klase

Vitičaste zagrade **{ }** ograđuju prostor tijela klase. Unutar vitičastih zagrada nalaze se varijable i funkcije koje su dio neke specifične klase i njezinih instanci. Elemente unutar tijela klase zovemo **članovima klase**. Za članove pojedine klase kažemo da su u opsegu (eng. *scope*) te klase, a sve što nije dio tijela specifične klase, smatramo da se nalaze van opsega klase. Navedeno će biti opisano u primjeru koji slijedi.

Tijelo klase „Student“:

```
{
    public string Ime;
    public string Prezime;
    public int Dob;
    public void PredstaviSe()
    {
        Console.WriteLine($"Zdravo, ja sam {Ime} {Prezime}! Imam {Dob} godina.");
    }
}
```

3.1.3. Instanciranje objekata

Klase služe kao predlošci za instanciranje objekata, odnosno stvaranja objekata, a za bolje razumijevanje toga, bit će detaljnije proučen primjer instanciranja ranije korištene klase „Student“:

Nakon implementiranja klase „Student“, moguće je deklarirati varijablu u programu tipa „Student“:

```
Student student1;
```

Navedenu varijablu moguće je inicijalizirati tako da se kao njezina vrijednost spremi instanca klase „Student“, odnosno objekt tipa „Student“:

```
student1 = new Student();
```

Može se primijetiti da je korištena ključna riječ *new*, jednako kao i u primjerima inicijalizacije listi i rječnika obrađenih u prethodnom poglavlju priručnika. Ranije prikazanim postupkom u varijablu „student1“ pohranjen je novi objekt tipa „Student“. Također, nakon toga može biti instancirana i druga varijabla, primjerice, „student2“, i u nju spremljen odvojen objekt „Student“. Navedeno jevidljivo u sljedećem primjeru:

```
Student student2 = new Student();
```

Sada postoje dva različita objekta iste klase „Student“ (jedan je pohranjen u „student1“, a drugi u „student2“). Oba objekta mogu sadržavati različite vrijednosti. Ako se promotri ranije izrađena deklaracija klase, može se primijetiti da u njezinu tijelu postoji niz varijabli:

```
public class Student
{
    public string Ime;
    public string Prezime;
    public int Dob;
    public void PredstaviSe()
    {
        Console.WriteLine($"Zdravo, ja sam {Ime} {Prezime}! Imam {Dob} godina.");
    }
}
```

Navedene varijable deklarirane su u tijelu klase, ali nisu instancirane. Varijable unutar klasa nazivaju se i **poljima klase**. One imaju modifikatore pristupa *public*, a što znači da se njima može pristupiti i izvan te deklaracije. Takve javne varijable mogu se inicijalizirati naknadno, odnosno nakon instanciranja objekta.

U sljedećem primjeru bit će instancirana polja novih varijabli „student1“ i „student2“:

```
student1.Ime = "Pero";
student1.Prezime = "Perić";
student1.Dob = 25;

student2.Ime = "Ivan";
student2.Prezime = "Horvat";
student2.Dob = 34;
```

Kao što je prikazano u primjeru, sintaksa za pristupanje članovima objekta jest:

imeObjekta.imeČlana

Rezultat je te operacije da objekti klase „Student“ sadrže različite podatke. Drugim riječima, svaki objekt „Student“ pohranjuje specifično ime, prezime i dob.

Osim što je moguće definirati varijable unutar objekata, moguće je i definirati funkcije unutar klase:

```
public class Student
{
    public string Ime;
    public string Prezime;
    public int Dob;

    public void PredstaviSe()
    {
        Console.WriteLine($"Zdravo, ja sam {Ime} {Prezime}! Imam {Dob} godina.");
    }
}
```

Funkcije unutar klasa zovu se i **metodama**. Da bi metoda bila pozvana u nekom objektu, potrebno se koristiti sljedećom sintaksom:

imeObjekta.imeMetode()

Primjer pozivanja metode „PredstaviSe()“ objekata „student1“ i „student2“:

```
student1.PredstaviSe();
student2.PredstaviSe();
```

Stoga što su polja promijenjena u „student1“ i „student2“, pozivanje iste funkcije na oba objekta producirat će dva različita rezultata:

Zdravo, ja sam Pero Perić! Imam 25 godina.

Zdravo, ja sam Ivan Horvat! Imam 34 godina.

Bitno je zaključiti da se svaki poziv funkcije „PredstaviSe()“ u ranijem primjeru koristio specifičnim vrijednostima pohranjenim u varijablama pripadajućeg objekta. Iz primjera se zaključuje i kako funkcije na različitim objektima iste klase mogu dati različite rezultate ovisno o pohranjenim vrijednostima njihovih varijabli (Pero Perić, Ivan Horvat). Ukupnost specifičnih vrijednosti polja nekog objekta zovemo **stanjem objekta**.

Zaključno o klasama i objektima

- Klasa je predložak za novi tip podatka, svaku instancu tog podatka zovemo objektom.
- Varijable i funkcije unutar tijela klase zovemo članovima klase.
- Varijable svakog pojedinog objekta imaju vlastite vrijednosti, nevezane s vrijednostima drugih objekata iste klase. Ukupnost svih vrijednosti naziva se stanjem objekta.
- Funkcije unutar klase zovemo metodama, a metode se koriste varijablama svog specifičnog objekta.

Nadalje, objekti omogućuju da se u jednoj programskoj strukturi objedini i stanje (varijable) i ponašanje (funkcije), koji su konceptualno srodni ili se odnose na istu stvar. U primjeru iz ovoga poglavlja postoji struktura koja pohranjuje nekoliko različitih podataka, a koje opisuju istog studenta: ime, prezime, dob. Također, moguće je i proširiti klasu „Student“ na način da njezini objekti sadržavaju i druge članove koji bi mogli biti relevantni za funkcionalnost programa koji se razvija, poput popisa ocjena, upisanih predmeta, metode za izračun prosjeka i slično. Programi u objektno orijentiranoj paradigmi sastoje se od brojnih klasa i objekata strukturiranih u odnose koji daju željeni rezultat.

3.2. Modifikatori pristupa

Modifikatori pristupa ključne su riječi u programskom jeziku *C#*, koje kontroliraju razinu vidljivosti i dostupnosti klasa i članova klasa izvan njihova deklariranog opsega. Svaki član klase može biti označen određenim modifikatorom pristupa, koji određuje tko može pristupiti određenim članovima i na koji način. U nastavku će biti detaljno razmotreni najčešći modifikatori pristupa za članove klasa u *C#*-u te neke od programerskih konvencija za njihovo korištenje.

Osnovni modifikatori pristupa su:

- *private*
- *public*
- *protected (obrađen u poglavlju 3.6.2.).*

3.2.1. Modifikator pristupa – public

Modifikator *public* čini član klase potpuno vidljivim izvan klase, a istovremeno omogućavajući pristup istom članu iz bilo kojeg dijela programa. To je ujedno i najširа razina vidljivosti te se često koristi za članove koji trebaju biti dostupni drugim dijelovima programa.

U nastavku će biti prikazan primjer opsega klase s javnom varijablom:

```
class Student
{
    public string Ime;
}
```

Pristupanje prethodno navedenoj javnoj varijabli izvan opsega klase implementira se na sljedeći način:

```
Student noviStudent = new Student();

noviStudent.Ime = "Pero Perić";
string imeStudenta = noviStudent.Ime;
```

Iz primjera je vidljivo da se moguće slobodno koristiti javnim poljem klase izvan njezina opsega. Na isti način moguće je pozivati i javne metode klase.

Nadalje, deklaracija javne metode u klasi može se implementirati na sljedeći način:

```
class Student
{
    public string Ime;

    public void PredstaviSe()
    {
        Console.WriteLine($"Ja sam {Ime}!");
```

```
    }
}
```

Pozivanje te javne metode izvan klase implementira se na sljedeći način:

```
Student noviStudent = new Student();

noviStudent.Ime = "Pero Perić";
noviStudent.PredstaviSe();
```

Konvencija za imenovanje – javna polja

Pojam konvencije za imenovanje javnih polja podrazumijeva da imena javnih polja počinju velikim slovom te da svaka riječ u imenu također počinje velikim slovom. Ta se vrsta notacije tipično naziva ***PascalCase***. U nastavku su istaknuti primjeri konvencija za imenovanje javnih polja:

```
public string ImeStudenta;
public string DobStudentaUGodinama;
```

3.2.2. Modifikator pristupa – private

Modifikator *private* čini član klase vidljivim samo unutar specifične klase u kojoj se nalazi. To znači da se član može koristiti samo unutar iste klase i nije dostupan izvan njezina opsega. Navedeno je korisno za skrivanje detalja implementacije.

Opseg klase s privatnim članom može se implementirati na sljedeći način:

```
class Student
{
    private string ime;
}
```

Važno je razumjeti da se privatnim varijablama ne može pristupiti izvan opsega klase. Privatne su varijable nevidljive razvojnom sučelju, a ako se pokuša pristupiti privatnoj varijabli izvan njezina opsega, to će proizvesti grešku u kompajliranju programa:

```
Student noviStudent = new Student();        // Instanciranje novog objekta.

noviStudent.ime = "Pero Perić";             // To su ilegalne instrukcije.
string imeStudenta = noviStudent.ime;       // To su ilegalne instrukcije.
```

U skladu s time, unutar opsega klase može se koristiti privatnim članovima.

```
class Student
{
   private string ime;

   public void PromijeniImeStudenta(string novoIme)
   {
      ime = novoIme;
   }
}
```

Metoda „PromijeniImeStudenta()“ javna je te je se može pozvati izvan opsega klase, a s obzirom na to da se navedena metoda nalazi u istoj klasi kao i privatna varijabla „ime“, ona ima pristup toj varijabli.

Privatne metode klase mogu se pozivati samo unutar opsega njihove klase, a što u praksi znači da se privatne metode mogu samo pozivati iz drugih metoda u istoj klasi. Privatne metode služe primarno za organiziranje koda klase. Kad god se niz identičnih instrukcija ponavlja u javnim metodama neke klase, potrebno je razmisliti bi li bilo preglednije da se niz instrukcija pretvori u privatnu metodu.

Konvencija za imenovanje – privatna polja

Pojam konvencije za imenovanje privatnih polja podrazumijeva da njihova imena počinju malim slovom, ali da svaka sljedeća riječ u imenu počinje velikim slovom. Ta vrsta notacije tipično se naziva ***camelCase***. Na primjer:

```
private string imeStudenta;
private string dobStudentaUGodinama;
```

Konvencija za imenovanje – metode

Metode klasa koriste se konvencijom *PascalCase* neovisno o tome jesu li privatne ili javne. Nazivi su metoda tipično rečenice u imperativu. U nastavku će biti prikazan primjer imenovanja metoda:

```
public void PromijeniImeStudenta(string novoIme);
public void IspisiStudenta();
private float VratiProsjekOcjena();
```

3.3. Svojstva i pristupnici

Svojstva su važan koncept u programskom jeziku *C#*, a koji omogućava kontrolirano pristupanje i upravljanje poljima klase. Pristupnici su posebne metode vezane za funkcionalnost svojstava.

Koji problem svojstva mogu riješiti?

Na samom početku potrebno se zapitati: kako bi se omogućilo dohvaćanje nekog polja klase izvan njezina opsega bez istodobnoga omogućavanja promjene njezine vrijednosti izvan njezina opsega?

Naime, svaka varijabla omogućuje dvije različite operacije. Jedna operacija omogućuje dohvaćanje vrijednosti varijable, što se naziva **dobivanje varijable**, a druga operacija omogućuje promjenu vrijednosti varijable, što se naziva **postavljanje varijable**. U slučaju određivanja razine dostupnosti varijable pomoću modifikatora pristupa,

određuju se opsezi iz kojih je moguće obavljati obje operacije: dobivanje i postavljanje. Kada je namjera privatno postavljati varijablu u klasi, uz želju da je se javno dobije, moguće je implementirati javnu metodu koja dobavlja privatnu varijablu te privatnu metodu koja postavlja tu istu varijablu:

```
class Student
{
   private string ime;
      public string DobiImeStudenta(string novoIme)
   {
      return ime;
   }
      private void PostaviImeStudenta(string novoIme)
   {
      ime = novoIme;
   }
 }
```

U tom primjeru moguće je dobiti varijablu „ime“ iz vanjskog opsega, ali ju je moguće postaviti samo iz unutarnjeg opsega. Potreba za takvom kontrolom pristupanja polju klase vrlo je česta u razvoju softvera s obzirom na funkcionalnost programskoga jezika *C#,* koja pojednostavljuje sintaksu za implementiranje takve strukture. **Svojstva** su posebni članovi klasa koji omogućavaju programeru kontrolirati dobivanje i postavljanje određenog polja unutar klase. Svojstva se definiraju pomoću ***get*** **i** ***set*** **blokova**. ***Get*** **blok** definira kako se vrijednost polja dobiva, dok ***set*** **blok** definira kako se postavlja nova vrijednost polja. *Get* i *set* blokove zovemo pristupnicima (eng. *accessors).* Drugim riječima, oni su posebne metode unutar svojstva koje definiraju na koji se način dobiva određena varijabla (eng. *get*) i kako se postavlja (eng. *set*).

U nastavku je prikazan primjer klase sa svojstvom:

```
public class Osoba
{
   private string ime; // Privatno polje.
   // Svojstvo za pristup imenu.
   public string Ime
   {
      get
```

```
        {
            return ime;
        }
        set
        {
            ime = value;
        }
    }
}
```

U tom primjeru svojstvo „Ime“ kontrolira pristup privatnom polju „ime“. Važno je naglasiti da se pristupnik *get* uvijek mora koristiti ključnom riječi *return* te mora vraćati isti tip vrijednosti deklarirane u potpisu svojega svojstva. S druge strane, pristupnik *set* koristi se ključnom riječi *value*, koja referira na vrijednost koju želimo postaviti u polje. Primjer korištenja svojstva izvan opsega klase sljedeći je:

```
Osoba osoba = new Osoba();
osoba.Ime = "Ana";                    // Postavljanje vrijednosti svojstva.
string imeOsobe = osoba.Ime;  // Dohvaćanje vrijednosti svojstva.
```

U gornjem primjeru svojstvo je javno te se, stoga, može pristupiti svojstvu „Ime“ izvan opsega klase. S druge strane, moguće je i *get* i *set* blokovima definirati zasebne modifikatore pristupa, dokle god su oni ograničeniji od modifikatora pristupa svojstva. Navedeno je vidljivo iz sljedećeg primjera klase s privatnim *set* pristupnikom:

```
public class Osoba
{
    private string ime; // Privatno polje.
    // Svojstvo za pristup imenu.
    public string Ime
    {
        get
        { return ime; }
        private set
        { ime = value; }
    }
}
```

U toj je verziji klase *get* pristupnik javan jer je svojstvo javno, ali je *set* pristupnik privatan. To znači da se vrijednost svojstva „Ime“ može dobiti izvan opsega klase, međutim, može biti postavljen samo unutar opsega klase.

3.4. Konstruktori

Konstruktori su posebne metode u programskom jeziku *C#*, kojima se koristi za inicijalizaciju objekata prilikom njihova stvaranja. Konstruktori omogućuju postavljanje početnih vrijednosti članova objekta i izvođenje drugih inicijalizacijskih radnji. U nastavku ovog poglavlja detaljno će biti razmotren koncept konstruktora, te kako ih definirati i njima se koristiti.

Definicija konstruktora

- Konstruktori se definiraju unutar klase i imaju isto ime kao i klasa. Konstruktori nemaju povratnu vrijednost, čak ni tip *void*, u potpisu svoje metode, ali pozivanje konstruktora vraća novu instancu klase. U osnovi, konstruktor je specijalna metoda koja se automatski poziva kada se stvori nova instanca klase.

U nastavku je prikazan primjer klase s jednostavnim konstruktorom:

```
class Student
{
   private string ime;
   private string prezime;
   private int dob;

   public Student(string ime, string prezime, int dob)
   {
      this.ime = ime;
      this.prezime = prezime;
      this.dob = dob;
   }
}
```

U tom primjeru definiran je konstruktor „Student()“ koji prima dva argumenta, „ime“, „prezime“ i „dob“, te postavlja članove klase „ime“, „prezime“ i „dob“ na vrijednosti tih argumenata. Stoga što su imena parametara konstruktora jednaka imenima članova klase, korištena je ključna riječi *this* kako bi se inicijalizirala polja klase. Ključna riječ

this precizira da se radi o članovima klase i koristi se u nekoliko različitih situacija. Primjerice, može se koristiti kako bi se eksplicitno naglasilo da je neki element član klase, i da se kod učini čitkijim, ili kada je potrebno s ključnom riječi *this* naznačiti na koju se varijablu referira, kao što je bio slučaj u ranije prikazanom primjeru.

Korištenje konstruktora

Konstruktori se koriste prilikom stvaranja objekata klase. Kada se objekt stvara pomoću operatora *new*, odgovarajući se konstruktor automatski poziva i izvršava. Na primjer:

```
Student noviStudent = new Student("Pero", "Perić", 25);
```

U tom se slučaju konstruktor „Student“ poziva parametrima „Pero“, „Perić“ i „25“, što rezultira stvaranjem objekta „noviStudent“ s postavljenim članovima „ime“ na „Pero“, „prezime“ na „Perić“ i „dob“ na „25“.

Jedna od tipičnih svrha konstruktora postavljanje je početne vrijednosti polja objekta, osobito kada se ne želi da polja objekata budu javna. Iz sljedećeg je primjera moguće primijetiti da su polja privatna:

```
class Student
{
   private string ime;
   private string prezime;
   private int dob;

   public Student(string ime, string prezime, int dob)
   {
      this.ime = ime;
      this.prezime = prezime;
      this.dob = dob;
   }
}
```

Drugim riječima, ime, prezime i dob studenta u toj se klasi mogu postaviti samo pomoću konstruktora.

Ako programer ne definira nijedan konstruktor za klasu, *C#* će automatski stvoriti tzv. **zadani konstruktor** (eng. *default constructor*), koji nema parametre i ne izvodi nikakve

posebne radnje. Takav konstruktor korišten je u prvom primjeru klase „Student“, kada nije eksplicitno definiran.

> 🕮 **Zadani konstruktor**
> Ako programer ne definira nijedan konstruktor za klasu, *C#* će automatski stvoriti tzv. **zadani konstruktor** (eng. *default constructor*), koji nema parametre i ne izvodi nikakve posebne radnje. Takav konstruktor korišten je u prvom primjeru klase „Student“, kada nije eksplicitno definiran.

3.4.1. Overloading konstruktora

Programski jezik *C#* omogućava *overloading* konstruktora (eng. *overloading* znači preopterećivanje), a što znači da je moguće definirati više različitih konstruktora u istoj klasi s različitim parametrima. Programeru navedeno omogućuje više opcija inicijalizacije objekta.

Preopterećivanje konstruktora opisano je sljedećim primjerom:

```
public class Pravokutnik
{
   private float duljina;
   private float sirina;

   // Konstruktor s dvama parametrima.
   public Pravokutnik(float duljina, float sirina)
   {
      this.duljina = duljina;
      this.sirina = sirina;
   }

   // Konstruktor s jednim parametrom.
   public Pravokutnik(float duljinaISirina)
   {
      duljina = duljinaISirina;
```

```
        sirina = duljinaISirina;
    }
}
```

Iz svega ranije navedenog može se zaključiti da su konstruktori važan dio objektno orijentiranog programiranja u *C#*-u. Oni omogućavaju inicijalizaciju objekata i postavljanje početnih vrijednosti za članove klase. Korištenje konstruktorima pomaže u osiguravanju ispravnoga i dosljednog stanja objekata prilikom njihova stvaranja.

3.5. Tipovi – referentni i vrijednosni

Tipovi podataka podijeljeni su u dvije osnovne kategorije:

1. vrijednosni tipovi
2. referentni tipovi.

Vrijednosni tipovi predstavljaju podatke koji se pohranjuju direktno u memoriju. To znači da varijable vrijednosnih tipova sadrže stvarnu vrijednost podatka, a ne referencu na neku lokaciju u memoriji. Primjeri vrijednosnih tipova u *C#*-u uključuju:

- ***int***
- ***float***
- ***char***
- ***bool***
- ***struct***.

Referentni tipovi predstavljaju podatke koji se pohranjuju kao reference na objekte u memoriji. Varijabla referentnog tipa ne sadrži stvarnu vrijednost podatka, već referencu (adresu) na objekt u memoriji. Primjeri referentnih tipova u *C#*-u uključuju:

- **objekte**
- ***string***, odnosno tip referentnog podatka koji se pohranjuje u memoriju pomoću reference, ali koji se ponaša sintaktički kao vrijednosni tip.

3.5.1. Korištenje vrijednosnih tipova

Činjenica da se referentni i vrijednosni tipovi drugačije pohranjuju u memoriju, utječe na pisanje koda. Sljedeći se primjer koristi vrijednosnim tipovima:

```
int broj1 = 2;
int broj2 = broj1;

broj1 = 4;
Console.WriteLine($"broj1: {broj1}, broj2: {broj2}");
```

U tom primjeru, kada se inicijalizira vrijednost varijable „broj2“, postavlja joj vrijednost varijable „broj1“. Nakon navedenog postavljanja, varijabla „broj2“ više nema referencu na vrijednost varijable „broj1“. Drugim riječima, to znači da navedene dvije varijable („broj1“ i „broj2“) pohranjuju odvojene vrijednosti te naknadno postavljanje nove vrijednost varijabli „broj1“ neće utjecati na promjenu vrijednosti varijable „broj2“.

> ⚠ **Uspoređivanje vrijednosnih tipova**
>
> Prilikom usporedbe varijabli s ugrađenim vrijednosnim tipovima, uspoređuju se vrijednosti pohranjene u varijabli, a što je vidljivo iz sljedećeg primjera:
>
> ```
> int broj1 = 2;
> int broj2 = 2;
> bool b1 = broj1 == broj2; //Istina.
> ```
>
> U prikazanom je primjeru usporedba jednakosti istinita.

3.5.2. Korištenje referentnih tipova

Gornji primjer može se usporediti sa sljedećim primjerom u kojem se koristi referentni tip:

```
int[] brojevi1 = new int[] {2, 4, 7};
int[] brojevi2 = brojevi1;

brojevi1[0] = 100;
Console.WriteLine($"brojevi1[0]: {brojevi1[0]}, brojevi2[0]: {brojevi2[0]}");
```

U navedenom primjeru inicijalizacija niza „brojevi2“ pohranjuje referencu na niz pohranjen u „brojevi1“, a što znači da se obje varijable referiraju na isti niz. Bilo koja promjena pohranjenog niza, odrazit će se na niz pohranjen u obje varijable.

⚠ **Uspoređivanje referentnih tipova**

Prilikom usporedbe varijabli s referentnim tipovima, logički operatori provjeravaju referiraju li varijable na isti objekt. Sljedeći primjer prikazuje usporedbu varijabli s referentnim tipovima:

```
int[] brojevi1 = new int[] { 2, 4, 7 };
int[] brojevi2 = new int[] { 2, 4, 7 };
bool b1 = brojevi1 == brojevi2;  //Neistina
```

U navedenom primjeru nizovi u “brojevi1” i “brojevi2” inicijalizirani su s istim elementima, ali su nizovi dvaju odvojenih objekata. Stoga ih operator „==“ne smatra identičnima. Ako se postavi vrijednost jedne varijable s drugom varijablom, onda će nam logički operator vratiti vrijednost istina:

```
int[] brojevi1 = new int[] { 2, 4, 7 };
int[] brojevi2 = brojevi1;
bool b1 = brojevi1 == brojevi2;  //Istina.
```

3.6. Nasljeđivanje

Nasljeđivanje je ključni koncept u objektno orijentiranom programiranju (**OOP**) koji omogućava stvaranje novih klasa na temelju postojećih. U programskom jeziku *C#* nasljeđivanje omogućava jednoj klasi (nazvanoj podklasa ili derivirana klasa) preuzimanje svojstva i metode druge klase (nazvane nadklasa ili bazna klasa). Taj koncept omogućava ponovnu uporabu koda i organizaciju hijerarhije klasa. Za definiranje nasljeđivanja u programskom jeziku *C#* koristi se znak **":"** nakon imena klase, a potom slijedi ime klase koja se nasljeđuje. Na primjer:

```
public class Vozilo
{
    public string Marka;
    public string Model;

    public void Pokreni()
    {
        Console.WriteLine("Vozilo je pokrenuto.");
```

```
    }
}

public class Automobil : Vozilo
{
    public int BrojVrata;
}
```

U tom primjeru klasa „Automobil“ nasljeđuje klasu „Vozilo“. To znači da će „Automobil“ automatski dobiti sve članove klase „Vozilo“, uključujući i članove podataka („Marka“ i „Model“) te metode („Pokreni()“).

> **VAŽNO JE ZNATI**
>
> Klase mogu imati više deriviranih klasa, koje mogu imati i svoje podklase, međutim, svaka klasa može imati samo jednu baznu klasu.

Sljedeća važna tema u kontekstu nasljeđivanja jest korištenje naslijeđenim članovima. Naime, nakon nasljeđivanja, članovi nadklase mogu biti dostupni u podklasi. Na primjer:

```
Automobil auto = new Automobil();
auto.Marka = "Toyota";
auto.Model = "Camry";
auto.BrojVrata = 4;
auto.Pokreni(); //Poziva metodu iz nadklase „Vozilo“.
```

U tom slučaju objekt „auto“ tipa „Automobil“ nasljeđuje članove klase „Vozilo“ i dodaje vlastiti član „BrojVrata“. Također, objekt „auto“ može pozivati metodu „Pokreni()“ iz svoje bazne klase. Osim što se klase mogu postaviti u varijable svog tipa, mogu se postaviti i u varijable svog baznog tipa. U tom ih slučaju pomoću varijable tretiramo kao bazni tip, te nemamo pristup članovima koji su deklarirani u deriviranoj klasi. Na primjer:

```
Vozilo vozilo = new Automobil();
vozilo.Marka = "Toyota";
vozilo.Model = "Camry";
vozilo.BrojVrata = 4; //To je nedopuštena operacija.
```

U tom primjeru neovisno o tome što je u varijablu „vozilo“ spremljen objekt „Automobil“, nije moguće pristupiti njezinu članu „BrojVrata“ iz razloga što varijabla tipa „Vozilo“ nema u sebi član „BrojVrata“.

> **VAŽNO JE ZNATI**
> Uvijek se može postaviti objekt podklase u varijablu nadklase, ali ne i suprotno.

3.6.1. Prednosti nasljeđivanja

Nekoliko je različitih prednosti nasljeđivanja, od kojih su najvažnije sljedeće:

- **ponovna uporaba koda:** nasljeđivanje omogućava ponovnu uporabu postojećih klasa i njihovih funkcionalnosti
- **organizacija koda:** hijerarhija klasa olakšava organizaciju i strukturu programa
- **proširivost:** nova funkcionalnost može se dodati podklasama bez mijenjanja postojećih klasa
- **razumljivost:** nasljeđivanje olakšava razumijevanje odnosa između različitih entiteta u programu.

Nasljeđivanje je moćan koncept u programskom jeziku *C#* koji omogućava stvaranje hijerarhije klasa, ponovnu uporabu koda i organizaciju programa. To je ključni element objektno orijentiranog programiranja, koji pomaže u razvoju fleksibilnih i skalabilnih aplikacija.

3.6.2. Modifikator pristupa – protected

Protected je modifikator pristupa u programskom jeziku *C#* koji se koristi za definiranje **članova klase** (polja ili metoda) koji su dostupni samo unutar te klase i svih njezinih **podklasa** (deriviranih klasa). To znači da se članovi označeni s ***protected*** mogu koristiti izravno u klasi, kao i u bilo kojoj klasi koja je nasljeđuje.

Na sljedećem primjeru bit će opisan odnos nadklase „Vozilo“ s *protected* članom „brojVrata“:

```
public class Vozilo
{
```

```
    protected int brojVrata;
}
```

Nakon toga, ako se stvori podklasa „Automobil“, koja nasljeđuje „Vozilo“, moguće je pristupiti *protected* članu „brojVrata“ unutar podklase. Navedeno je prikazano u sljedećem primjeru:

```
public class Automobil : Vozilo
{
    public void PostaviBrojVrataAutomobila(int broj)
    {
        brojVrata = broj; // Moguće je pristupiti članu „brojVrata“ unutar podklase.
    }
}
```

Članovima označenim s *protected* ne može se direktno pristupati izvan **hijerarhije nasljeđivanja** (izvan podklasa). Modifikator pristupa ***protected*** restriktivniji je od modifikatora ***public***, ali manje restriktivan od modifikatora ***private***, koji ograničava pristup članu klase samo u klasi u kojoj je deklariran. To ograničenje osigurava odgovarajuću kontrolu pristupa podatcima i metodama unutar hijerarhije klasa.

3.6.3. Konstruktori u naslijeđenim klasama

U slučaju da postoje hijerarhije naslijeđenih klasa, važno je razumjeti kako konstruktori rade u tim klasama i kako se ponašaju kada se objekt derivirane klase kreira. Uzevši u obzir da derivirane klase uključuju polja vlastitih baznih klasa, konstruktori deriviranih klasa omogućuju pozivanje konstruktora bazne klase i postavljanje vlastitih parametara.

Ponašanje konstruktora u naslijeđenim klasama

- Kada se stvara objekt **derivirane klase** (podklase), najprije se poziva konstruktor **bazne klase** (nadklase), a zatim se izvršava inicijalizacija objekta derivirane klase.
- Konstruktor se bazne klase poziva pomoću ključne riječi ***base*** u konstruktoru izvedene klase.

Sintaksa je za deklaraciju konstruktora derivirane klase sljedeća:

public ImeDeriviraneKlase(parametri derivirane klase) : base(parametri bazne klase)

```
{
        //Tijelo konstruktora.
}
```

U navedenom primjeru postoji bazna klasa „Osoba“ s konstruktorom i izvedena klasa „Student“, koja nasljeđuje klasu „Osoba“ te posjeduje vlastiti konstruktor:

```
public class Osoba
{
   public string Ime;
   public int Dob;

   public Osoba(string ime, int dob)
   {
      Ime = ime;
      Dob = dob;
   }
}

public class Student : Osoba
{
   public string Sveuciliste;

   public Student(string ime, int dob, string sveuciliste) : base(ime, dob)
   {
      Sveuciliste = sveuciliste;
   }
}
```

U navedenom primjeru konstruktor klase „Student“ poziva konstruktor klase „Osoba“ pomoću ključne riječi ***base*** da bi inicijalizirao svoje naslijeđene članove („Ime“ i „Dob“), a zatim postavlja vlastiti član „Sveuciliste“. Može se uočiti kako su u parametrima iza ključne riječi *base* unesene varijable koje su deklarirane u parametrima derivirane klase: „ime“ i „dob“. Kao što je ranije istaknuto, navedenim će se varijablama koristiti

za pozivanje konstruktora bazne klase „Osoba“ prije nego što se provedu instrukcije u tijelu konstruktora derivirane klase „Student“.

Primjer pozivanja konstruktora

Sljedeći primjer prikazuje način na koji je moguće stvoriti objekt izvedene klase „Student“:

```
Student student = new Student("Ana", 20, "Sveučilište Zagreb");
```

U navedenom primjeru konstruktor bazne klase „Osoba“ automatski postavlja ime i starost, dok konstruktor izvedene klase „Student“ postavlja sveučilište.

3.7. Apstraktne klase i apstraktne metode

Apstraktne klase u programskom jeziku *C#* koriste se za definiranje zajedničkog sučelja i strukture za više konkretnih (izvedenih) klasa. Apstraktne klase same po sebi ne mogu biti instancirane. Umjesto toga, koriste se kao predložak za izradu konkretnih klasa koje će naslijediti njihova svojstva i metode. Apstraktne se klase definiraju pomoću ključne riječi *abstract*. Mogu sadržavati apstraktne metode, koje nemaju konkretnu implementaciju, već samo definiraju metode za koje derivirane klase moraju stvoriti implementaciju. Samo apstraktne klase mogu deklarirati apstraktnu metodu.

Na primjer:

```
public abstract class GeometrijskiOblik
{
    public abstract float DobaviPovrsinu();
}
```

U navedenom primjeru apstraktna klasa deklarira apstraktnu metodu „DobaviPovrsinu()“. Međutim, takva apstraktna metoda ne daje konkretnu implementaciju, već svaka klasa koja nasljeđuje od „GeometrijskiOblik“ mora implementirati vlastitu verziju metode „DobaviPovrsinu()“.

Na primjer:

```
public class Kvadrat : GeometrijskiOblik
{
    public float Stranica;
```

```
    public override float DobaviPovrsinu()
    {
        return Stranica * Stranica;
    }
}

public class Krug : GeometrijskiOblik
{
    public float Radius;

    public override float DobaviPovrsinu()
    {
        return Radius * Radius * 3.14f;
    }
}
```

U ranijem primjeru derivirane klase „Kvadrat“ i „Krug“ implementiraju vlastite verzije apstraktne metode „DobaviPovrsinu()“, koristeći se ključnom riječi ***override***. Klase derivirane od apstraktne klase s apstraktnim metodama trebaju imati konkretne implementacije svake od apstraktnih metoda. Izračun površine za svako geometrijsko tijelo drugačiji je, i zahtijeva vlastitu implementaciju, stoga je smisleno koristiti se apstraktnom metodom za uspostavu zajedničkoga sučelja bez zajedničke implementacije.

Ključne prednosti apstraktnih klasa

- Apstraktne klase omogućavaju definiranje zajedničkog sučelja i ponašanja za grupu sličnih klasa.
- Omogućavaju organizaciju i strukturiranje koda hijerarhijom nasljeđivanja, bez stvaranja nepotrebnih konkretnih tipova.
- Pružaju apstraktnu strukturu za buduće konkretne implementacije.

3.8. Virtualne metode

Virtualne metode funkcionalno su slične apstraktnim metodama, međutim, za razliku od njih, virtualne omogućuju osnovnu implementaciju metoda unutar bazne klase koju onda derivirane klase mogu promijeniti po potrebi. Za razliku od apstraktnih metoda,

virtualne metode mogu biti članovi običnih i apstraktnih klasa. To znači da se ponašanje metode može, ali ne mora, prilagoditi za svaku konkretnu klasu koja je naslijedila tu metodu.

Kako definirati virtualne metode?

Da bi metoda bila označena kao virtualna, koristi se ključna riječ *virtual* prije deklaracije metode u baznoj klasi. Na primjer:

```
public class Osoba
{
    public virtual void Pozdrav()
    {
        Console.WriteLine("Pozdrav, ja sam obična osoba.");
    }
}
```

U navedenom primjeru, metoda „Pozdrav()“ u klasi „Osoba“ označena je kao virtualna. Klasa koja nasljeđuje od klase „Osoba“ može, po potrebi, promijeniti implementaciju metode „Pozdrav()“, koristeći se ključnom riječi *override*. Na primjer:

```
public class Student : Osoba
{
    public override void Pozdrav()
    {
        Console.WriteLine("Pozdrav, ja sam student.");
    }
}
```

U navedenom primjeru, klasa „Student“ prekriva virtualnu metodu „Pozdrav()“, definiranu u klasi „Osoba“. Kada se poziva metoda na objektu derivirane klase, pozvat će se prekrivena (eng. *override*) verzija metode te klase. Međutim, ako se ista metoda poziva na objektu nadklase, koji je stvoren kao instanca izvedene klase, pozvat će se prekrivena verzija izvedene klase. Primjerice:

```
Osoba osoba1 = new Osoba();
Osoba osoba2 = new Student();

osoba1.Pozdrav(); // Ispisuje „Pozdrav, ja sam obična osoba.“
osoba2.Pozdrav(); // Ispisuje „Pozdrav, ja sam student.“
```

VAŽNO JE ZNATI

Virtualne metode omogućavaju fleksibilnije korištenje metodama baznih klasa.

3.9. Dizajnerski ciljevi objektno orijentiranog programiranja

Dizajnerski su ciljevi OOP-a, kao što je spomenuto u uvodnom dijelu ovog poglavlja, razumijevanje, održavanje i ponovna uporaba koda. Dva su osnovna koncepta koji podržavaju navedene ciljeve: **polimorfizam** i **enkapsulacija**.

3.9.1. Polimorfizam

Polimorfizam predstavlja proces uspostavljanja zajedničkog sučelja za objekte različitih tipova. Polimorfizam je u praksi organizacija koda koja omogućuje pozivanje iste javne funkcije u različitim tipovima objekata te će oni imati sebi svojstvene implementacije iste funkcije. Pozivatelj funkcije ne treba znati ništa o implementaciji te funkcije, pa čak ni o specifičnom tipu objekta čiju funkciju poziva.

U prvom dijelu poglavlja: *Gradivni elementi objektno orijentiranog programiranja* implementirano je nekoliko primjera polimorfizma, a specifično u primjerima apstraktnih i virtualnih metoda.

Primjer apstraktnih metoda za postizanje polimorfizma:

```
public abstract class GeometrijskiOblik
{
   public abstract float DobaviPovrsinu();
}

public class Kvadrat : GeometrijskiOblik
{
   public float Stranica;
```

```
    public override float DobaviPovrsinu()
    {
        return Stranica * Stranica;
    }
}

public class Krug : GeometrijskiOblik
{
    public float Radius;

    public override float DobaviPovrsinu()
    {
        return Radius * Radius * 3.14f;
    }
}
```

U prikazanom primjeru metoda „DobaviPovrsinu()“ predstavlja manifestaciju polimorfizma. Ona je sučelje zajedničko nekoliko različitih tipova. „Krug“ i „Kvadrat“ koriste se istom metodom „DobaviPovrsinu()“ s različitim implementacijama. To znači da je moguće istom instrukcijom pozivati obje metode i time strukturirati program koji ne mora znati poziva li metodu iz klase „Krug“ ili „Kvadrat“.

Na primjer:

```
List<GeometrijskiOblik> oblici = new List<GeometrijskiOblik>()
{
    new Kvadrat(2f),
    new Krug(4f),
    new Kvadrat(5f)
};

foreach (GeometrijskiOblik oblik in oblici)
{
    float povrsinaOblika = oblik.DobaviPovrsinu();
```

```
    Console.WriteLine($"Ovaj oblik ima povrsinu: {povrsinaOblika}");
}
```

Inicijalizacija varijable „**povrsinaOblika**“ postavlja se pomoću vraćene vrijednosti metode „DobaviPovrsinu()“ pozvane na iteriranom elementu. Nigdje se u *foreach* petlji ne definira točan tip elementa, nego je samo poznata njegova bazna klasa „GeometrijskiOblik“. To je moć polimorfizma jer omogućava da se raznim tipovima koristimo u istu svrhu ili u istom kontekstu.

3.9.2. Enkapsulacija

Enkapsulacija je proces grupiranja podataka i mehanizama koji operiraju na istim podatcima te u isto vrijeme označava i strateško ograničavanje pristupa tim podatcima i mehanizmima. U praksi su podatci i mehanizmi jednaki članovima klasa ili *structova*, odnosno metodama i varijablama unutar pojedine klase. Pristup tim članovima klasa i *structova* ograničava se koristeći se modifikatorima pristupa te promišljenim strukturiranjem klase.

Jednostavno rečeno, objektno orijentirano programiranje disciplina je razdvajanja programskog koda na odvojene strukture, a enkapsulacija je rezultat tog procesa.

Enkapsulirana klasa ona je koja skriva detalje vlastite implementacije od korisnika, te koja izlaže minimalni broj konzistentnih sučelja, najčešće javnih metoda, a koje omogućuju da se njome koristimo. Smisao je enkapsulacije određene klase da njezini korisnici, odnosno druge klase koje se njome koriste, imaju minimalni kontakt s njom radi lakšega održavanja koda. Konačni je cilj da unatoč potrebi za promjenom implementacije određene klase, nije potrebno ništa mijenjati u klasama koje se njome koriste. Odnosno, da unatoč tome što je implementacija mijenjana, sučelje klase ostaje nepromijenjeno.

Enkapsulacija je, dakle, ambicija prema kojoj se stremi radi lakšega održavanja koda. Dobro enkapsulirane klase u programu smanjuju broj čimbenika o kojima programer mora razmišljati kada radi promjene u kodu.

3.10. Programski obrasci objektno orijentiranog programiranja

U praksi OOP-a postoji niz obrazaca kojima se rješavaju učestali problemi u organizaciji koda. **Programski** se **obrasci** mogu smatrati vrstama tipskih rješenja u OOP-u. U ovom dijelu bit će obrađena tri korisna obrasca u razvoju videoigara i razvoju okruženja alternativne stvarnosti. Bit će opisani problemi koje rješavaju i kako ih jednostavno implementirati.

Obrasci:

- **obrazac petlje igre** (eng. *game loop*)
- **promatrački obrazac** (eng. *observer pattern*)
- **obrazac stanja** (eng. *state pattern*).

Kao i svako rješenje u programiranju, specifični pristup i implementacija obrazaca može varirati ovisno o potrebi programera te specifičnostima problemskog zadatka, stoga je bitno zapamtiti da su prikazana rješenja u ovom poglavlju samo jedan od brojnih načina implementacije programskih obrazaca objektno orijentiranog programiranja.

S ciljem oslikavanja konteksta u kojima se implementiraju ovi obrasci, bit će korišteni primjeri iz područja razvoja videoigara i okruženja alternativne stvarnosti.

3.11. Obrazac petlje igre

Petlja igre (eng. *game loop*) programski je obrazac koji formira osnovnu logiku svih videoigara i okruženja alternativne stvarnosti. Gotovo sve videoigre simuliraju protok vremena neovisno o naredbama igrača: likovi se kreću virtualnim prostorom, animacije se reproduciraju, elementi se korisničkog sučelja ažuriraju i slično. Takva simulacija protoka vremena postiže se metodom koja je usporediva s reprodukcijom filmova i videa. Filmska se traka sastoji od niza statičnih slika, čija brza izmjena stvara dojam pokreta na filmu.

Ako se dovoljno slika izmijeni u jednoj sekundi videa, odnosno ako je dovoljno visoka frekvencija izmjene slika, tada ljudi doživljaju video kao pokretnu sliku umjesto niza statičnih slika. Takve statične slike u videoprodukciji zovemo statičnim kadrovima (eng. *still frame*). Kadar je, dakle, osnovna gradivna jedinica videa. Na isti se način programiraju igre tako da stvaraju niz statičnih kadrova koji se u izmjenjuju u visokoj frekvenciji za iluziju pomične slike. Takav obrazac zovemo petljom igre.

Petlja igre je beskonačna petlja koja se ponavlja određen broj puta u sekundi, a u svakoj iteraciji petlje najprije se ažurira stanje svih objekata u igri, a onda se njihova grafička reprezentacija iscrta na ekran. Kao što svaku sliku u videu zovemo kadrom, tako i svaku iteraciju u petlji igre zovemo kadrom (eng. *frame*). Na kraju svake iteracije u petlji igre poziva se funkcija koja pauzira provedbu programa na određeno vrijeme da ne bi prolazak kroz petlju bio prebrz, odnosno da se generiranje i iscrtavanje novih kadrova događa u željenoj brzini. Tipičan je ciljani broj kadrova 60 po sekundi.

Slijedi jednostavan primjer implementacije petlje igre:

```
public class Igra
 {
    private bool igraPokrenuta =  false;

    private List<Entitet> entiteti;

    public void PokreniIgru()
    {
       igraPokrenuta = true;

       while (igraPokrenuta)
       {

          foreach (Entitet entitet in entiteti)
          {
             entitet.Azuriraj();
          }

          PauzirajProvedbuPetlje();
       }
    }
}
```

U tom primjeru srž obrasca petlje igre nalazi se u *while* petlji unutar funkcije „PokreniIgru()“. Svako ponavljanje kroz *while* petlju u funkciji „PokreniIgru()“ predstavlja jedan kadar igre. U svakom se kadru ponavlja kroz listu objekata „entiteti“, te se svaki element ažurira pomoću polimorfične funkcije „Azuriraj()“.

U obrascu petlje igre entitet je bazna klasa od koje nasljeđuju svi objekti koji mogu postojati u igri u više od jedne iteracije petlje igre, odnosno u više od jednog kadra. Ako se uzme kao primjer konkretna igra, poput FIFA-e – simulacije nogometne utakmice, onda bi, između ostalih, entiteti u igri bili: igrači na terenu, lopta, publika, korisničko sučelje, itd. Neki entiteti imaju grafičku reprezentaciju u igri (igrači, lopta, publika), a neki entiteti nemaju (sustav umjetne inteligencije, audiosustav). Veže ih činjenica da oni moraju moći postojati u više od jednog kadra igre da bi igra ispravno funkcionirala. Svi entiteti u igri moraju biti pohranjeni u istu listu po kojoj se unutar petlje igre iterira. U tipičnoj implementaciji entiteta programer stvara klasu „Entitet“, koja definira polimorfičnu funkciju (virtualnu ili apstraktnu) „Ažuriraj()“, kojoj derivirane klase onda definiraju vlastitu implementaciju. U sljedećem je primjeru predstavljena jednostavna implementacija klase „Entitet“:

```
public class Entitet
{
    public virtual void Azuriraj()
    {

    }
}
```

Svaka će derivirana klasa „Entitet“ definirati svoje specifično ponašanje, tako da implementira vlastitu verziju funkcije „Azuriraj()“. U primjeru nogometne videoigre lopta bi definirala svoju implementaciju funkcije „Azuriraj()“, a igrač na terenu svoju.

Pojednostavljen primjer klasa deriviranih od „Entiteta“:

```
public class Nogometas : Entitet
{
```

```
    public override void Azuriraj()
    {
        //Logika koja opisuje ponašanje nogometaša u svakom kadru.
        //Procesiranje umjetne inteligencije, odabir animacija, smjera kretanja, itd.
    }
}

public class Lopta : Entitet
{
    public override void Azuriraj()
    {
        //Logika koja opisuje ponašanje lopte u svakom kadru.
        //Proračun njezine trenutne brzine, smjera, pozicije, sila, itd.
    }
}
```

Ovisno o potrebama specifične igre, bazna klasa „Entitet“ može definirati i druga zajednička polja i funkcionalnosti za sve derivirane entitete, poput pozicije u prostoru ili reference na audiosustav.

Primjena petlje igre u razvojnom alatu *Unity*

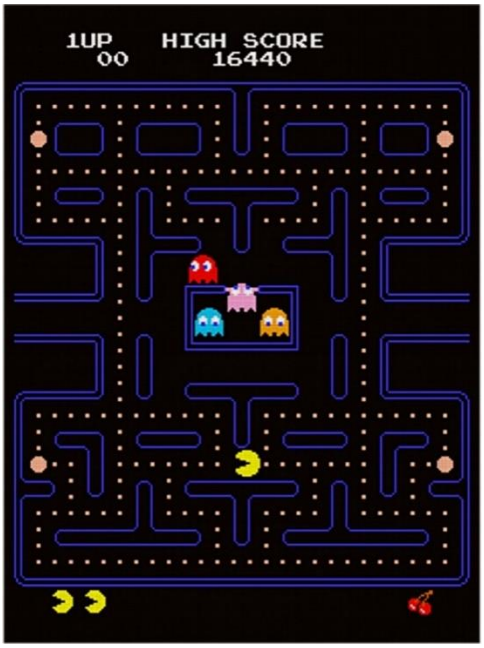
Moderni razvojni alati videoigara, odnosno „motori“ igara (eng. *game engine*) tipično imaju ugrađeni obrazac petlje igre u svojem programskom sučelju, što omogućuje programerima razvoj igara bez implementacije vlastitoga obrasca petlje igre. *Unity* razvojni alat, koji se obrađuje u 3. poglavlju, ima ugrađenu klasu „Monobehaviour“, koja preuzima ulogu entiteta u obrascu petlje igre. Kao i klasa „Entitet“ iz gornjeg primjera, „Monobehaviour“ definira vlastitu polimorfičnu funkciju za ažuriranje, koja se zove „Update()“.

3.12. Promatrački obrazac

Promatrački obrazac (eng. *observer pattern*) programski je obrazac koji omogućava objektu objavu promjena o svom stanju tako da sve zainteresirane strane (promatrači) budu automatski obaviještene o tim promjenama. Promatrački obrazac nastoji postići tu izmjenu informacija bez znanja promatranoga objekta za promatrače i suprotno.

Koji problem promatrački obrazac nastoji riješiti?

Zamislite da razvijate bodovni sustav za kultnu igru *Pac-Man*. U *Pac-Manu* igrač prolazi labirint i pokušava skupljati točke koje mu daju bodove, a istovremeno izbjegava četiri duha koji ga pokušavaju uhvatiti. Cilj je postići najveću količinu bodova prije nego što je igrač uhvaćen.



Slika 13. Pac-Man (slika zaslona, igra Pac-Man)

U igri postoje dvije vrste točaka:

- male točke koje daju 10 bodova
- četiri velike točke koje daju 50 bodova, i koje omogućuju na kratko vrijeme da igrač pojede duhove, a svaki pojedeni duh, također, igraču daje bodove.

Navedeno znači da u *Pac-Manu* postoje tri načina da igrač dobije bodove:

1. kada pojede malu točku
2. kada pojede veliku točku
3. kada pojede duha.

Kako je moguće implementirati takvu funkcionalnost?

Pretpostavi li se da je bodovni sustav klasa koja pamti broj bodova, ispisuje ih na ekran i prilagođava ovisno o radnjama igrača, ona se može ovako implementirati:

```
public class UpravljačBodova
{
   private int bodovi;
      public void DodajBodove(int noviBodovi)
   {
      bodovi += noviBodovi;
   }
      private void PrikaziBodove()
   {
      //Kod za prikazivanje bodova na ekranu.
   }
}
```

Prikazana klasa implementira javnu funkciju „DodajBodove()“, ali da bi joj objekti koji daju bodove igraču mogli pristupiti izvana, moraju imati referencu na instancu „UpravljačBodova“. To bi značilo da se prilikom instanciranja svakog objekta koji igraču daje bodove, mora postaviti referenca na taj objekt.

U sljedećem primjeru prikazane su pojednostavljene deklaracije tih objekata:

```
public class MalaTočka
{
   private UpravljačBodova mojUpravljacBodova;

   //Konstruktor postavlja referencu na upravljač bodova.
   public MalaTočka(UpravljačBodova upravljacBodova)
   {
      mojUpravljacBodova = upravljacBodova;
   }
      public void SkupiTočku()
   {
      mojUpravljacBodova.DodajBodove(10);
      //Ostatak funkcionalnosti skupljanja točke.
   }
```

```
}

public class VelikaTočka
{
    private UpravljačBodova mojUpravljacBodova;

    //Konstruktor postavlja referencu na upravljač bodova.
    public VelikaTočka(UpravljačBodova upravljacBodova)
    {
        mojUpravljacBodova = upravljacBodova;
        //Ostatak funkcionalnosti konstruktora.
    }
        public void SkupiTočku()
    {
        mojUpravljacBodova.DodajBodove(50);
        //Ostatak funkcionalnosti skupljanja točke.
    }
}

public class Duh
{
    private UpravljačBodova mojUpravljacBodova;

    //Konstruktor postavlja referencu na upravljač bodova.
    public Duh(UpravljačBodova upravljacBodova)
    {
        mojUpravljacBodova = upravljacBodova;
        //Ostatak funkcionalnosti konstruktora.
    }
        public void PojediDuha()
    {
        mojUpravljacBodova.DodajBodove(50);
        //Ostatak funkcionalnosti jedenja duha.
    }
```

}

To rješenje predstavlja niz problema koji kompliciraju razvoj videoigre te otežavaju proširenje funkcionalnosti programa:

- svi objekti koji instanciraju ili upravljaju tim objektima, primjerice objekt koji postavlja točke, moraju imati referencu na objekt „UpravljacBodova“, da bi ga postavili u konstruktor. Dakle, objekti koji se ne koriste s „UpravljacBodova“, moraju znati za njega, a što dovodi do rizika od nastanka potencijalnih grešaka
- ako je tijekom razvoja potrebno ukloniti objekt „UpravljacBodova“ iz bilo kojeg razloga, primjerice za potrebe testiranja neke nove ili odvojene funkcionalnosti, to nije moguće napraviti jer konstruktori tih objekata očekuju referencu na „UpravljacBodova“
- količina bodova dobivenih kada se pojede pojedini element, definiran je u deklaraciji elementa, a ne u deklaraciji „UpravljacBodova“. To znači, ako je potrebno promijeniti količine dobivenih bodova, treba napraviti promjene na trima klasama koje nisu „UpravljacBodova“ te kojima to nije primarna funkcija. Za smisleno održavanje koda bilo bi učinkovitije kada bi se podešavanje događalo unutar klase „UpravljacBodova“.

Ukratko, problem je s takvim rješenjem što ono povećava povezanost koda i uzajamnu ovisnost odvojenih elemenata programa. Odnosno, točke moraju znati za bodovni sustav i ovise o njegovu postojanju. Takvi problemi postaju izraženiji kada se zamisle drugi sustavi koje bi jedenje točaka i duhova moglo interesirati, a za koje nema smisla da točke i duhovi direktno znaju. Primjerice, audiosustav koji reproducira određene zvučne efekte kada se objekti jedu, ili sustav promjene nivoa koji provjerava je li određen broj točaka pojeden.

3.12.1. Osnova promatračkog obrasca

Elementi u videoigrama često moraju obznaniti neku promjenu svog stanja sustavima za koje ne žele biti direktno vezani referencom i biti ovisni o njima. Takve objave tipično se nazivaju porukama (eng. *messages*) ili događajima (eng. *events*) te one formiraju osnovnu ideju **promatračkog obrasca**.

Promatrački se obrazac sastoji od dvaju osnovnih elemenata:

- promatrači
- subjekti.

U ovom primjeru promatrač je „BodovniSustav“, koji čeka objavu događaja (*event*) od subjekata „MalaTočka“, „VelikaTočka“ i „Duh“.

Promatrač je zadužen za „pretplatu“ na objavu određenog događaja, a **subjekti** implementiraju sučelje za takvu pretplatu, ali tako da nemaju direktnu referencu na pretplaćene promatrače. Logika je tog obrasca ugrađena u sintaksu programskog jezika *C#* pomoću sustava *event*, a u nastavku će biti objašnjena funkcionalnost tog sustava.

3.12.2. Delegati i eventi

Delegat je tip podataka koji predstavlja referencu na metodu. Delegati omogućavaju da se metode tretiraju kao objekti, što znači da se mogu pohraniti u varijable, prenositi kao argumenti funkcija i pozivati kao normalne metode. Delegat se deklarira pomoću ključne riječi *delegate* i definira potpis metode koju će delegat referencirati. Na primjer:

```
public delegate void MojDelegat(string poruka);
```

Delegat „MojDelegat“ može referencirati metode koje primaju jedan argument tipa *string* i nemaju povratnu vrijednost. Drugim riječima, navedeni delegat stvara tip varijable koja može pohraniti referencu na bilo koju metodu bilo kojeg objekta, pod uvjetom da ta metoda prima samo jedan *string* parametar i nema povratnu vrijednost.

Nakon što je delegat deklariran, moguće je stvoriti instancu istog delegata. Na primjer:

```
MojDelegat delegat1;
MojDelegat delegat2;
```

Nakon što su delegatne varijable deklarirane, moguće je dodijeliti metodu koju će referencirati. Za potrebe ovog primjera najprije će biti implementirane dvije metode koje se mogu pohraniti u delegat jer imaju potrebne parametre i povratne vrijednosti.

```
void IspisiPrvoSlovoPoruke(string poruka)
{
    Console.WriteLine(poruka[0]);
}
void IspisiDrugoSlovoPoruke(string poruka)
{
    Console.WriteLine(poruka[1]);
}
```

Nakon što su implementirane, funkcije je moguće postaviti u delegate:

```
delegat1 = IspisiPrvoSlovoPoruke;
delegat2 = IspisiDrugoSlovoPoruke;
```

U ranijem primjeru „delagat1“ i „delegat 2“ referenciraju metode „IspisiPrvoSlovoPoruke()“ i „IspisiDrugoSlovoPoruke()“, koje odgovaraju potpisu delegata. Moguće je primijetiti kako su postavljeni u delegate bez zagrada, zato što se ispisivanjem zagrada funkcije pozivaju i vraćaju povratnu vrijednost, a nije želja dobiti njihovu povratnu vrijednost, nego njihovu referencu.

Nakon što su postavljene reference na metode u delegate, moguće ih je pozvati. Delegati se pozivaju kao normalne metode. Kada je delegat pozvan, on će izvršiti metodu koju referencira i proslijediti joj argumente. Na primjer:

```
delegat1("Pozdrav, delegat 1"); // Poziva „IspisiPrvoSlovoPoruke“ s argumentom.
delegat2("Pozdrav, delegat 2"); // Poziva „IspisiDrugoSlovoPoruke“ s argumentom.
```

Taj poziv delegata rezultirat će izvršavanjem odgovarajuće metode s proslijeđenim argumentom.

Delegati u programskom jeziku *C#* mogu sadržavati više od jedne reference na metodu, a što znači da će poziv delegata pozvati sve pohranjene metode. Za dodavanje više od jedne metode, potrebno se koristiti operatorom „+=“. Primjerice:

```
delegat1 += IspisiPrvoSlovoPoruke;
delegat1 += IspisiDrugoSlovoPoruke;
```

U sljedećem primjeru pozivanje delegata pozvat će obje funkcije s proslijeđenim argumentima:

```
delegat1("Pozdrav, delegat 1")
```

Dakle, najprije će se pozvati pohranjena metoda „IspisiPrvoSlovoPoruke()“, a onda metoda „IspisiDrugoSlovoPoruke()“.

Delegati su osnova *event* sustava u programskom jeziku *C#* jer omogućuju da subjekt sadrži delegat u koji promatrači spremaju svoje funkcije. Kada subjekt želi obznaniti neki događaj, primjerice kada točka u *Pac-Manu* želi javiti da je pojedena, onda ona kao subjekt pozove određeni delegat, a time se sve metode njezinih promatrača pohranjene u taj delegat pozivaju. Na taj način subjekt ne zna, odnosno nema referencu na sustave koji očekuju taj događaj, i nije ovisan o njihovu postojanju. Kako bi se pojednostavila implementacija takvog *event* sustava, u primjeru će se koristiti ključnom riječju *event* te ugrađenim tipom *Action*.

Action

Action je ugrađeni tip delegata u programskom jeziku *C#*, koji ne vraća vrijednost, i kojemu je moguće definirati jednu varijablu bilo kojeg tipa prilikom deklaracije varijable. Na primjer:

```
public Action delegatBezParametra;
public Action<string> delagatSaStringParametrom;
public Action<int> delagatSaIntParametrom;
```

Iz primjera je vidljivo da se tip parametra određuje njegovim unosom u uglate zagrade. Da bismo se koristili ugrađenim tipom *Action*, potrebno je programu dodati imenski prostor (eng. *namespace*) *System* na vrhu dokumenta:

```
using System;
```

Action je koristan tip u programskom jeziku *C#* jer omogućava jednostavno deklariranje delegata prigodnih za promatrački obrazac bez potrebe deklariranja novoga tipa delegata.

Ključna riječ *event*

Ključna riječ *event* u kombinaciji s delegatom sprječava pozivanje delegata izvan njegova opsega, čak i u slučajevima u kojima je delegat deklariran kao javan. To omogućava da postoji javni delegat u koji objekti izvan opsega mogu postaviti svoje metode, ali da ga istodobno ne mogu pozvati. Time *event* ključna riječ sprječava da greškom bude pozvan delegat izvana. Na primjer:

```
public class Subjekt
{
    public event Action delegatBezParametra;
    public event Action<string> delagatSaStringParametrom;
    public event Action<int> delagatSaIntParametrom;
}
```

U tom se primjeru klase izvan klase „Subjekt“ mogu pretplatiti na njezine delegate, ali ih ne mogu pozvati. To je točno kontrola pristupa kakvu se priželjkuje u implementaciji sustava *event*.

Apstraktni primjer implementacije sustava *event*

Prije povratka originalnom problemu implementacije bodovnog sustava, bit će prikazan apstraktni primjer implementacije promatračkog obrasca korištenjem *evenata* u programskom jeziku *C#*.

U sljedećem primjeru postoje dvije klase, „Subjekt“ i „Promatrac“, koje komuniciraju korištenjem *evenata*:

```
public class Subjekt
{
   public static event Action delegatBezParametra;
      public void PozoviDelegat()
   {
      delegatBezParametra?.Invoke();
   }
}
public class Promatrac
{
   public Promatrac()
   {
      Subjekt.delegatBezParametra += DogadajPrepoznat;
   }
   private void DogadajPrepoznat()
   {
      Console.WriteLine("Ova funkcija je pozvana iz Action delegata na Subjektu.");
   }
}
```

S posebnom pozornošću potrebno je promotriti sljedeće elemente u primjeru:

Static – ključna riječ u deklaraciji _eventa_

```
public static event Action delegatBezParametra;
```

Delegat *event* je deklariran korištenjem ključnom riječju *static*. Statični delegati pohranjuju sve metode u delegat koji je neovisan o pojedinoj instanci klase. Time omogućuje da se promatrači pretplaćuju na delegat kojem imaju pristup, čak i kada nemaju referencu na neku instancu klase. To u praksi znači da je moguće imati bilo koji broj instanci klase „Subjekt“, i da sve te instance dijele jedan delegat.

Metoda delagata „Invoke()“ i operator?

```
delegatBezParametra?.Invoke();
```

Tu se koristi ugrađenom metodom „Invoke()“, koja poziva delegat. Moguće je pozvati *event* delegat bez te metode, kao što je vidljivo u gornjim primjerima, ali s ovakvom sintaksom moguće je koristiti operator „?“ nakon imena delegata. Operator „?“ u ovom kontekstu provjerava postoje li pohranjene metode u delegatu, te ako ih nema, on preskače pozivanje metode „Invoke()“. Navedeno je bitno ako se pokuša pozvati prazni delegat, jer će se javiti greška, pa se čitko i jednostavno može ograditi od te mogućnosti.

U nastavku je prikazan primjer uporabe ovih klasa:

```
Subjekt subjekt = new Subjekt();
Promatrac promatrac = new Promatrac();

subjekt.PozoviDelegat();
```

Pokrene li se program, primijetit će se da je subjekt pozivanjem delegata pozvao metodu objekta „Promatrac“, bez da je ijedan objekt imao referencu na drugoga. Promatrač je dobio poruku od subjekta bez „znanja“ o njemu, i obje klase mogu imati funkcionalne instance neovisno o drugoj. To je srž promatračkog obrasca.

Konkretni primjer promatračkog obrasca

Sada je moguće iskoristiti funkcionalnost promatračkog obrasca za implementaciju bodovnog sustava u *Pac-Manu*. Subjekti će biti tri klase koje mogu biti pojedene, a s druge strane, promatrač će biti bodovna klasa:

```
//Promatrac
public class UpravljacBodova
{
   private int bodovi;
      public UpravljacBodova()
   {
      MalaTočka.Pojedena += DodajBodoveZaMaluTocku;
      VelikaTočka.Pojedena += DodajBodoveZaVelikuTocku;
      Duh.Pojeden += DodajBodoveZaDuha;
   }
   public void DodajBodoveZaMaluTocku()
   {
      bodovi += 10;
```

```
    }
        public void DodajBodoveZaVelikuTocku()
    {
        bodovi += 40;
    }
        public void DodajBodoveZaDuha()
    {
        bodovi += 200;
    }
}
// Subjekti
public class MalaTočka
{
    public static event Action Pojedena;
    public void SkupiTočku()
    {
        TockaPojedena?.Invoke();
    }
}
public class VelikaTočka
{
    public static event Action Pojedena;
    public void SkupiTočku()
    {
        Pojedena?.Invoke();
    }
}
public class Duh
{
    public static event Action Pojeden;
    public void SkupiTočku()
    {
        Pojeden?.Invoke();
    }
```

}

Kao što je vidljivo iz primjera, problemi su s prvom implementacijom riješeni: objekti više nisu ovisni jedni o drugima te se funkcionalnost bodovanja nalazi u potpunosti u opsegu klase „UpravljacBodovanja“. Koristeći se promatračkim obrascem, sada je moguće lako implementirati i druge promatrače koje zanima jesu li subjekti pojedeni, bez proširivanja klase subjekata.

3.13. Obrazac stanja

Obrazac stanja (eng. *state pattern*) način je da se implementira niz različitih i međusobno isključivih ponašanja na nekom objektu, koja se mogu dinamički izmjenjivati ovisno o uvjetima.

Koju vrstu funkcionalnosti obrazac stanja omogućava?

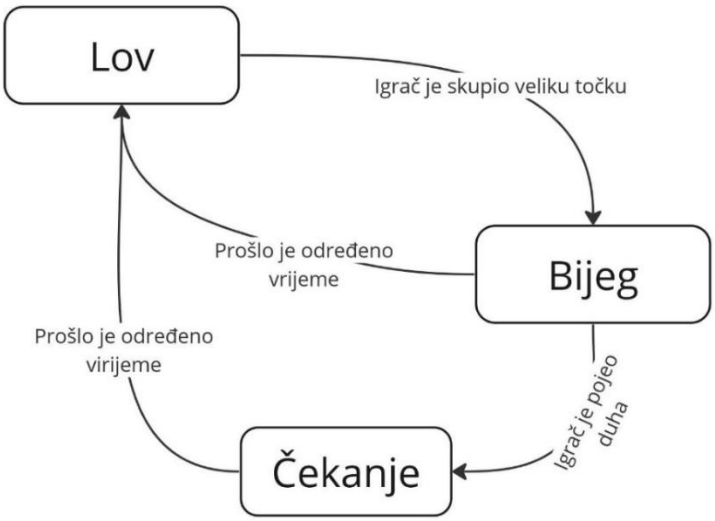
Zamislimo razvoj umjetne inteligencije za duhove u *Pac-Manu*.

U *Pac-Manu* četiri duha pokušavaju uhvatiti igrača u labirintu. U normalnim uvjetima oni love igrača, ali ako igrač pojede jednu od četiri velike točke u nivou, onda duhovi neko vrijeme bježe od igrača. Dok duhovi bježe, ako ih igrač dotakne, on ih pojede, te su duhovi vraćaju u svoju početnu poziciju u centru labirinta, gdje čekaju određeno vrijeme prije nego što opet počinju loviti igrača. To se ponašanje duhova u *Pac-Manu* može opisati kao tri međusobno isključiva „stanja“, odnosno tri različita i distinktivna ponašanja:

1. **lov:** duh pokušava doći do igrača, a ako ga dotakne, igrač je pojeden
2. **bijeg:** duh se pokušava udaljiti od igrača, a dotakne li igrača, duh je pojeden
3. **čekanje:** duh čeka u mjestu. Igrač i duh se ne mogu dotaknuti.

Osim što ta tri stanja imaju različito ponašanje, ona se također moraju izmjenjivati ovisno o uvjetima igre i trenutnom stanju duha. Svako stanje definira određene uvjete za prelazak u sljedeće stanje.

Sljedeći dijagram opisuje moguće izmjene stanja i njihove uvjete:



Slika 14. Dijagram obrasca stanja (Juračić, autorov rad)

Ključno je primijetiti da su ta stanja međusobno isključiva. Duh u *Pac-Manu* ne može istodobno loviti i bježati, ili čekati i loviti. Odnosno, duh može postojati samo u jednom od triju stanja u bilo kojem trenutku.

Kakav je odnos tih stanja s obrascem petlje igre?

U svakom bi kadru igre trenutno stanje duha trebalo ažurirati njegovo ponašanje. Odnosno, svako bi stanje trebalo implementirati vlastitu verziju metode „Azuriraj()“, koja bi se koristila relevantnim poljima u objektu „Duh“.

Sljedeći primjer pokazuje pojednostavljenu implementaciju klase „Duh“, koja se pokušava postići obrascem stanja:

```
public class Duh : Entitet
{
    private StanjeDuha mojeTrenutnoStanje;
    private Dictionary<string, StanjeDuha> mojaMogucaStanja;

    public override void Azuriraj()
    {
        mojeTrenutnoStanje.Azuriraj();
    }
}
```

Tijekom svakog kadra igre klasa „Duh“ u svojoj funkciji „Azuriraj()“ poziva konkretnu implementaciju stanja u kojem se nalazi, a spremljena je u polje „mojeTrenutnoStanje“.

Klasa „Duh“ također sadrži rječnik svojih drugih mogućih stanja, u kojima je ključ svakog elementa *string* a vrijednost određena instanca stanja duha. Svako je stanje u rječniku „mojaMogucaStanja“ reprezentirano specifičnim tipom koji nasljeđuje od zajedničke apstraktne bazne klase. Sljedeći primjer pokazuje implementaciju klasa stanja, odnosno apstraktnu baznu klasu:

```
public abstract class StanjeDuha
{
    protected Duh mojDuh;
    //Konstruktor.
    public StanjeDuha(Duh duh)
    {
        mojDuh = duh;
    }
    public abstract void Azuriraj();
}
```

Apstraktna bazna klasa uključuje konstruktor, koji postavlja referencu na objekt „Duh“ u *protected* varijablu „mojDuh“. Time se osigurava da stanje ima pristup relevantnim poljima svog duha, te da se može njima koristiti. Kao što i entiteti u obrascu igre petlje definiraju javnu polimorfičnu funkciju „Azuriraj()“, tako i apstraktna klasa stanja u tom obrascu mora definirati javnu polimorfičnu funkciju, koju će klasa „Duh“ moći pozivati. Sljedeći primjeri konkretne su implementacije apstraktne klase „StanjeDuha“:

```
public class Bijeg : StanjeDuha
{
   //Konstruktor.
   public Bijeg(Duh duh) : base(duh) {}

   public override void Azuriraj()
    {
        //Logika za stanje „Bijeg“.
    }
}
public class Lov : StanjeDuha
{
   //Konstruktor.
```

```
    public Lov(Duh duh) : base(duh) {}
    public override void Azuriraj()
    {
        //Logika za stanje „Lov“.
    }
}
public class Cekanje : StanjeDuha
{
    //Konstruktor.
    public Cekanje(Duh duh) : base(duh) {}
    public override void Azuriraj()
    {
        //Logika za stanje „Cekanje“.
    }
}
```

Svako od tih konkretnih stanja mora sadržavati u implementaciji svoje funkcije „Azuriraj()“ (koja poziva svaki kadar) uvjetnu logiku koja omogućava da se trenutno stanje objekta „Duh“ zamijeni drugim stanjem. Da bi se to postiglo, mora se implementirati funkcija za promjenu stanja u klasi „Duh“:

```
public class Duh : Entitet
{
    private StanjeDuha mojeTrenutnoStanje;
    private Dictionary<string, StanjeDuha> mojaMogucaStanja;
    public override void Azuriraj()
    {
        mojeTrenutnoStanje.Azuriraj();
    }
    public void PromijeniStanje(string imeStanja)
    {
        mojeTrenutnoStanje = mojaMogucaStanja[imeStanja];
    }
}
```

Metoda „PromijeniStanje()“ prima *string* parametar „imeStanja“, koji se koristi kao ključ u rječniku „mojaMogucaStanja“. Dohvaćeno stanje iz rječnika potom se postavlja u

varijablu „mojeTrenutnoStanje“. U sljedećem kadru objekt „Duh“ pozvat će metodu „Azuriraj()“ na novom stanju postavljenom u njezinoj varijabli „mojeTrenutnoStanje“. Kako bi funkcija „PromijeniStanje()“ mogla uspješno promijeniti stanje, rječnik „mojaMogucaStanja“ mora se popuniti stanjima pri inicijalizaciji objekta „Duh“. To se može postići pomoću konstruktora klase „Duh“:

```
public class Duh : Entitet
{
    private StanjeDuha mojeTrenutnoStanje;
    private Dictionary<string, StanjeDuha> mojaMogucaStanja;

    public Duh()
    {
        mojaMogucaStanja = new Dictionary<string, StanjeDuha>()
        {
            {"Lov", new Lov(this)},
            {"Bijeg", new Bijeg(this)},
            {"Cekanje", new Cekanje(this)}
        };
        PromijeniStanje("Lov");
    }
    public override void Azuriraj()
    {
        mojeTrenutnoStanje.Azuriraj();
    }
    public void PromijeniStanje(string imeStanja)
    {
        mojeTrenutnoStanje = mojaMogucaStanja[imeStanja];
    }
}
```

U konstruktoru, nakon što je inicijaliziran rječnik s instancama stanja, poziva se funkcija „PromijeniStanje()“, s parametrom „Lov“, što postavlja početnu vrijednost varijable „mojeTrenutnoStanje“. Zato što je funkcija „PromijeniStanje()“ javna, njome se stanja koriste za promjenu trenutnoga stanja objekta „Duh“ u slučaju da su ispunjeni uvjeti za to. Sljedeći primjer pokazuje navedenu funkcionalnost, u primjeru klase „Bijeg“:

```
public class Bijeg : StanjeDuha
{
    //Konstruktor.
    public Bijeg(Duh duh) : base(duh) {}
    public override void Azuriraj()
    {
        if (ProsloJeVrijemeZaBijeg())
        {
            mojDuh.PromijeniStanje(„Lov“);
            return;
        }
        if (IgracJePojeoDuha())
        {
            mojDuh.PromijeniStanje(„Cekanje“);
            return;
        }
        //Logika za stanje „Bijeg“.
    }
    private bool ProsloJeVrijemeZaBijeg()
    {
        //Logika koja provjerava je li prošlo vrijeme za bijeg.
    }
    private bool IgracJePojeoDuha()
    {
        //Logika koja provjerava je li igrač pojeo duha.
    }
}
```

U metodi „Azuriraj()“ prije nego što se provodi potrebna logika za to stanje, provjeravaju se uvjeti za promjenu stanja, i to dvjema privatnim logičkim (*bool*) metodama: „ProsloJeVrijemeZaBijeg()“ i „IgracJePojeoDuha()“, ali nisu definirane specifične implementacije tih metoda jer nisu relevantne za razumijevanje.

Izlazne i ulazne metode stanja

Obrazac stanja omogućava definiranje metoda koje se pozivaju pri ulasku ili izlasku iz pojedinog stanja. Te izlazne ili ulazne metode pozivaju se čim se stanje promijeni, za razliku od metode „Azuriraj()“, koja se provodi jednom svaki kadar.

Da bi se implementirale ulazne i izlazne metode za stanja, potrebno je deklarirati polimorfične metode u apstraktnoj baznoj klasi „StanjeDuha“:

```
public abstract class StanjeDuha
{
    protected Duh mojDuh;
    public StanjeDuha(Duh duh)
    {
        mojDuh = duh;
    }
    public abstract void Azuriraj();
    public abstract void UdiUStanje();
    public abstract void IzadiIzStanja();
}
```

U ovom se primjeru koristi apstraktnim metodama, ali, naravno, moguće je umjesto apstraktnih, implementirati virtualne metode. Konkretne će klase, derivirane od „StanjaDuha“, imati vlastitu implementaciju tih metoda.

Kako bi se osigurao poziv tih izlaznih i ulaznih metoda u točnom trenutku, potrebno je modificirati metodu „PromijeniStanje()“ u klasi „Duh“:

```
public void PromijeniStanje(string imeStanja)
{
    if (mojeTrenutnoStanje != null)
    {
        mojeTrenutnoStanje.IzadiIzStanja();
    }
    mojeTrenutnoStanje = mojaMogucaStanja[imeStanja];
    mojeTrenutnoStanje.UdiUStanje();
}
```

Bitno je primijetiti u ovoj modificiranoj verziji metode da prije njezina poziva provjeravamo uvjet „(mojeTrenutnoStanje != null)“. Taj će uvjet biti ispunjen samo ako je varijabla „mojeTrenutnoStanje“ inicijalizirana, odnosno ako je u tu varijablu

postavljena referenca na objekt. Taj se uvjet mora provjeriti zato što bi pozivanje metode „IzadiIzStanja()“ u nepostojećem objektu, rezultiralo greškom.
Smisao je te modificirane verzije metode „PromijeniStanje()“ pozvati funkciju „IzadiIzStanja()“ u stanju u kojem se trenutno nalazi objekt „Duh“, prije nego što se ono promijeni. Nakon što se trenutno stanje promijeni, tek se onda poziva funkcija „UdiUStanje()“.
Obrazac stanja moćan je alat iz dvaju razloga:

- omogućava programeru da razloži kompleksno ponašanje jednog objekta („Duh“) na niz objekata (podklase klase „StanjeDuha“), što pridonosi čitkosti koda
- pojednostavljuje dodavanje novih ponašanja nekom objektu jer svako novo ponašanje nije kruto vezano za implementaciju klase kojoj stanje pripada. Odnosno, postoji samo minimalna ovisnost između različitih stanja.

Prilikom implementacije bilo koje klase čija funkcionalnost zahtijeva kompleksnu uvjetnu logiku, programer se treba zapitati može li se ta uvjetna logika razložiti na niz različitih i međusobno isključivih stanja. Ako može, takva je klasa savršen kandidat za implementaciju obrasca stanja.

PITANJA ZA PONAVLJANJE:

Provjerite svoje znanje odgovorima na sljedeća pitanja:

1. Što je objektno orijentirano programiranje i kako se razlikuje od drugih programskih paradigmi?
2. Koja je razlika između klase i objekta?
3. Zašto su modifikatori pristupa važni u objektno orijentiranom programiranju?
4. Objasnite razliku između javnih (eng. *public*) i privatnih (eng. *private*) polja. U kojim slučajevima bi se koristila javna, a kada privatna polja?
5. Što su svojstva i pristupnici?
6. Koji se problemi mogu riješiti pomoću svojstava?
7. Kako konstruktori doprinose stvaranju objekta?
8. Što znači *overloading* konstruktora?
9. Objasnite razliku između referentnih i vrijednosnih tipova njihovom usporedbom.
10. Što je nasljeđivanje i koje su njegove prednosti u programiranju?

11. Kako apstraktne klase i metode doprinose modularnosti i fleksibilnosti koda?
12. Što je polimorfizam i kako doprinosi fleksibilnosti i ponovnoj upotrebi koda?
13. Objasnite koncept enkapsulacije i zašto je važan.
14. Što je promatrački obrazac i kako se može primijeniti u realnim programskim situacijama?

4. UVOD U ENGINE: SRCE DIGITALNIH SVJETOVA

U OVOM POGLAVLJU NAUČIT ĆETE:

- postaviti razvojno okruženje u programu Unity
- koristiti se glavnim komponentama za razvoj 3D okruženja u Unityju (svjetlo, zvuk, animacije, skriptiranje)
- upotrijebiti vještine i znanja iz područja C# programiranja u 3D sučelju
- integrirati pakete razvojnog alata.

Motori (eng. *engine*), u kontekstu razvoja videoigara i aplikacija, temelj su na kojem se gradi cijeli digitalni svijet. Oni nisu samo alati, već kompleksni sustavi koji programerima, dizajnerima i umjetnicima pružaju platformu na kojoj mogu oblikovati, kreirati i animirati svoje ideje u interaktivne doživljaje. U osnovi, motori upravljaju svim ključnim aspektima digitalne kreacije, od renderiranja grafike, fizike, umjetne inteligencije, do upravljanja audiovizualnim elementima.

Motori igara upravljaju nizom središnjih komponenti i sustava, svaki svojom specifičnom funkcijom. Grafički sustavi omogućavaju renderiranje tekstura, svjetlosti i objekata, dok sustavi fizike kalkuliraju interakcije, pokrete i kolizije objekata unutar virtualnog prostora. Sustavi za upravljanje zvukom orkestriraju zvučnu kulisu, dok sustavi umjetne inteligencije omogućavaju likovima i elementima da djeluju autonomno ili reagiraju na korisničke unose. Te i mnoge druge komponente rade sinkronizirano za besprijekorno korisničko iskustvo.

Uza svoje osnovne funkcije, motori također nude i različite alate koji omogućavaju timovima realizaciju svojih vizija. Ti alati uključuju, ali nisu na njih ograničeni, uređivače nivoa, animacijske studije, programerska sučelja i mnoge druge. Svi ti alati omogućuju stručnjacima različitih područja da surađuju, kreirajući složene i dinamične virtualne svjetove. Motori igara nisu samo tehnički alati, već sredstva koja premošćuju tehničke i kreativne aspekte razvoja, oživljavajući inovacije unutar digitalnog prostora. Motori

igara su, u konačnici, mostovi koji povezuju ideje s interakcijom. Oni olakšavaju kreiranje izvanrednih iskustava, koja mogu istraživati različite aspekte ljudske interakcije i narativa, te eksperimentirati unutar virtualnog konteksta. Bilo da je riječ o igrama, simulacijama ili edukativnim aplikacijama, motori pružaju osnovu na kojoj se mogu graditi bogati, imerzivni i zadivljujući digitalni svjetovi, omogućavajući kreatorima da dijele svoje priče, iskustva i ideje širom svijeta.
Postoji mnogo motora (eng. *engines*) koji omogućuju razvojnim timovima da kreiraju igre, aplikacije i različite vrste digitalnih doživljaja. Neki su od najpoznatijih:

- ***Unity***: svestran i široko korišten motor, koji podržava različite platforme i vrste projekata
- ***Unreal Engine***: poznat po svojim impresivnim grafičkim sposobnostima i realnom prikazu svjetla
- ***Godot***: besplatan i motor otvorenog koda (eng. *open-source*) sa širokim spektrom mogućnosti
- ***CryEngine***: orijentiran prema visokokvalitetnoj grafici i realnom prikazu okoliša
- ***RPG Maker***: specijaliziran za kreiranje RPG igara s naglaskom na jednostavnost korištenja.

4.1. Razvojni alat Unity

Unity je razvojni alat ili „motor“ koji se primarno koristi za kreiranje videoigara. Međutim, njegova primjena seže i dalje od toga. U posljednjih nekoliko godina *Unity* je postao popularan i za kreiranje različitih interaktivnih sadržaja, uključujući **virtualnu stvarnost** (VR), **proširenu stvarnost** (AR), simulacije, animacije, obrazovne aplikacije i mnoge druge.

Osnovne značajke *Unity* razvojnog motora:

- **višeplatformska podrška:** jedna je od najistaknutijih značajki *Unityja* sposobnost izvoza (eng. *build*) projekta u mnoge različite platforme. To uključuje osobna računala, konzole, mobilne uređaje, internetske preglednike i mnoge druge platforme

- **grafičko sučelje:** *Unity* nudi intuitivno grafičko sučelje koje omogućava korisnicima da vizualno organiziraju i uređuju objekte unutar scene, definiraju ponašanje objekata i pregledavaju svoj rad u stvarnom vremenu
- **skriptiranje:** ponašanje objekata unutar *Unity* aplikacija obično se definira pomoću skripti napisanih u jeziku *C#*. Te skripte omogućuju razvojnim inženjerima kreiranje kompleksne interakcije, logiku igre i druga ponašanja
- **fizički motor:** *Unity* uključuje fizički motor koji simulira stvarne fizičke interakcije, kao što su gravitacija, sudari ili trenje
- ***Asset Store*:** *Unityjeva* trgovina imovinom (eng. *Asset Store*) mrežna je trgovina gdje korisnici mogu kupiti i prodavati gotove resurse, uključujući modele, teksture, skripte i dodatke koji proširuju funkcionalnost motora
- **integracija VR/AR:** *Unity* je postao ključni alat za razvoj VR i AR aplikacija zbog svoje snažne podrške popularnim VR i AR uređajima te integracije s njima.

4.1.1. Instalacija Unityja

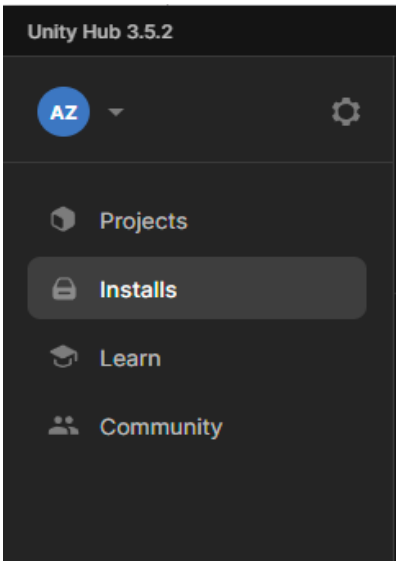
Unity je kreiran 2005. godine, i od tada je postao jedan od najpopularnijih razvojnih alata na svijetu. Njegova pristupačnost, fleksibilnost i jaka zajednica razvojnih inženjera čine ga atraktivnim izborom mnogih tvrtki i nezavisnih razvojnih timova širom svijeta.

Za instalaciju *Unityja* potrebno je otvoriti mrežni preglednik te posjetiti *Unityjevu* službenu mrežnu stranicu.

Koraci instalacije:

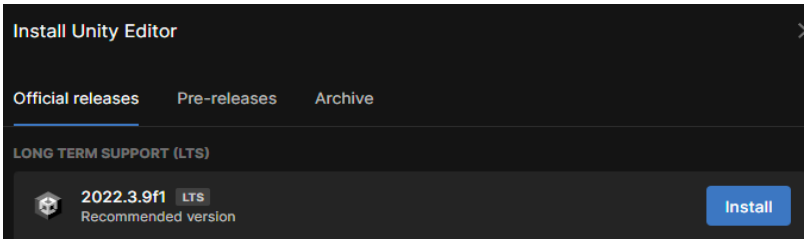
1. **preuzimanje *Unity Huba*:** na glavnoj je stranici potrebno potražiti odjeljak za preuzimanje (eng. *Download*). *Unity Hub* je centralna aplikacija koja omogućuje upravljanje različitim verzijama *Unity* uređivača, projekata i dodataka
2. **pokretanje instalacije:** nakon što je *Unity Hub* preuzet, može se pokrenuti instalacijski program te je potrebno slijediti upute na ekranu. Tijekom postupka instalacije bit će ponuđen odabir komponenti za instalaciju (poput različitih modula ovisnih o platformama, dokumentacije itd.)
3. **prijava ili registracija:** nakon instalacije *Unity Huba* potrebno je pokrenuti aplikaciju. Raspolaže li s već postojećim *Unity* računom, potrebno se prijaviti. Ako ne, potrebno se registrirati.

Unity nudi različite licence svoje platforme, uključujući besplatnu verziju ***Unity Personal*** i komercijalne verzije poput ***Unity Pro***.



Slika 16. Opcije Unity Huba (slika zaslona, Unity Hub)

Instalacija *Unity* uređivača: unutar *Unity Huba* potrebno je otići u odjeljak „Instalacije“ (eng. *Installs*). Tu je moguće odabrati i instalirati verziju *Unity* uređivača koju se želi koristiti, ili više njih ako je potrebno. Zatim je potrebno odabrati opciju ***Install Editor***, a potom verziju *Unity* uređivača koju se želi instalirati. Preporuča se preuzeti **LTS** (eng. *Long-Term Support*). Zatim pritisnuti gumb „instaliraj“ (eng. *Install*).



Slika 15. Biranje verzije Unityja (slika zaslona, Unity Hub)

4.1.2. LTS – Long-Term Support

LTS, eng. *Long-Term Support* u hrvatskom se prevodi kao „dugoročna podrška“. U kontekstu platforme *Unity*, LTS verzija predstavlja stabilnu verziju *Unity* softvera koja će primati ažuriranja i podršku tijekom duljeg vremenskog razdoblja, obično dvije godine.

Značajke i prednosti *Unity* LTS verzije sljedeće su:

- **stabilnost**: LTS verzije su namijenjene onima koji traže stabilno okruženje, posebno za projekte koji su već u produkciji ili su blizu završetka. Razvojni timovi koji koriste LTS verziju, mogu očekivati manje problema ili greške.
- **dugoročna podrška**: kroz cijelo razdoblje podrške ***Unity Technologies*** pruža ažuriranje ispravaka i sigurnosnih zakrpa za LTS verzije. Međutim, nove značajke obično nisu uključene u te ispravke radi stabilnosti verzije
- **fokus na postojanosti**: dok standardne verzije *Unityja* mogu uključivati nove i eksperimentalne značajke, LTS verzija je usredotočena na postojanost i pouzdanost, što je ključno za veće projekte i komercijalne aplikacije.

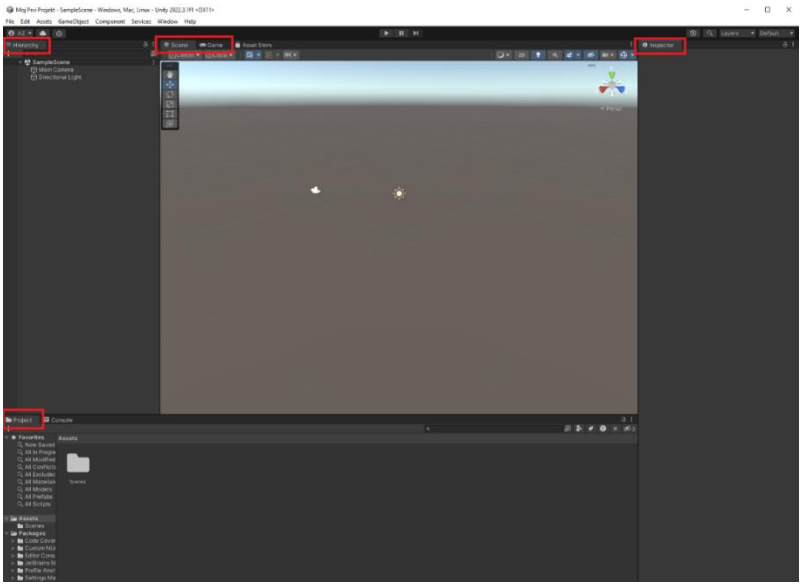
Razvojnim timovima koji traže najnovije značajke i inovacije, redovite verzije *Unityja* mogu biti privlačnije. Međutim, ukoliko je potrebno stabilno i pouzdano razvojno okruženje za duže vremensko razdoblje, utoliko je LTS verzija najbolji izbor.

Stvaranje novog projekta

Nakon što je uspješno instalirana željena verzija *Unityja* posredstvom *Unity Huba*, kreiranje novog projekta prilično je jednostavno. Na početnoj stranici, koja se naziva „Projekti“ (eng. *projects*), nalazi se gumb „Novi Projekt“ (eng. *new project*) smješten na vrhu prozora. Klikom na navedeni gumb, otvara se novi ekran, gdje se najprije odabire verzija *Unityja* koju se namjerava koristiti. Nastavno, unosi se ime novog projekta, „**Moj Prvi Projekt**“. Zatim se određuje gdje se na računalu projekt pohranjuje, a nakon toga se odabire odgovarajući predložak – **3D Core**. Nakon što smo zadovoljni unesenim informacijama, potrebno je kliknuti na „Stvori Projekt“ (eng. *create project*). *Unity* će potom obraditi zahtjev i pripremiti radno okruženje za novi projekt. Ovisno o računalu i odabranom predlošku, to može potrajati nekoliko minuta. Kada se proces završi, otvorit će se radno okruženje *Unityja*, i tada je sve spremno za rad na novom projektu.

4.2. Korisničko sučelje alata Unity

Unityjevo sučelje dizajnirano je da bi omogućilo korisnicima intuitivno kreiranje i upravljanje projektima. Ključni dijelovi toga sučelja različiti su prozori, a svaki služi specifičnoj svrsi. Evo objašnjenja glavnih prozora unutar *Unity* sučelja:



Slika 17. Korisničko sučelje alata Unity (slika zaslona, Unity)

Hijerarhija (eng. *hierarchy*)

Prozor prikazuje sve objekte trenutno prisutne u aktivnoj sceni. Slično kao struktura mape na računalu, omogućuje organizaciju objekata unutar scene. Objekti unutar hijerarhije mogu biti organizirani u odnose roditelj-dijete. Primjerice, ukoliko postoji objekt automobil i želi se postići da kotači prate njegovo kretanje, svaki kotač bi bio „dijete“ objekta automobil. Kad se automobil premjesti, svi kotači (dječji objekti) ga automatski prate. Druge akcije, tipa skaliranja, također preslikavaju svoje promjene dječjim objektima.

Prikaz scene (eng. *scene view*)

Označava prozor gdje korisnici vizualno uređuju i raspoređuju objekte unutar scene. Korisnicima omogućuje pregledavanje scene iz različitih kutova, zumiranje, rotiranje i selektiranje objekata. Osim toga, sadrži različite alate za transformaciju objekata, poput pomaka, rotacije i skaliranja.

Prikaz igre (eng. *game view*)

Prozor precizno prikazuje kako će igra izgledati i ponašati se jednom kada se izgradi i pokrene. Ovdje korisnici mogu testirati igru u stvarnom vremenu unutar *Unity* okruženja.

Prozor projekta (eng. *project window*)

Sučelje koje prikazuje sve resurse i datoteke povezane s trenutnim *Unity* projektom. Tu korisnici mogu pristupiti svim svojim modelima, teksturama, skriptama, zvučnim datotekama i drugoj imovini. Omogućuje organizaciju, pretragu i filtriranje imovine unutar projekta.

Inspektor (eng. *inspector*):

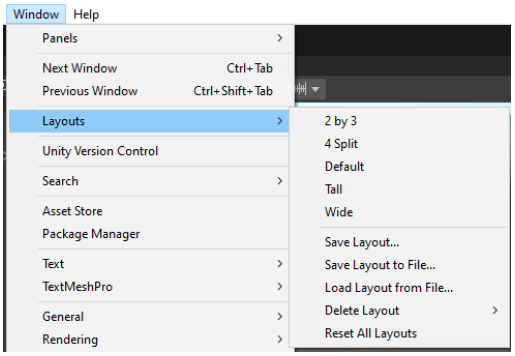
Kada se objekt u „Hijerarhiji“ ili „Prikazu scene“ odabere, „Inspektor“ prikazuje sve njegove atribute i postavke. Tu korisnici mogu uređivati svojstva objekta, dodavati komponente, mijenjati postavke renderiranja, fizička svojstva, pridružene skripte i mnoge druge detalje. „Inspektor“ se dinamički mijenja ovisno o tome koji je objekt ili imovina trenutno odabrana. Ti prozori čine osnovu *Unityjeva* korisničkog sučelja te pružaju razvojnim inženjerima sve potrebne alate i informacije za kreiranje interaktivnih aplikacija i igara. Početnicima će moguće trebati malo vremena da se naviknu na to radno okruženje (eng. *workflow*), ali s vremenom postaju vrlo učinkoviti i produktivni u korištenju tim prozorima.

4.2.1. Prilagođavanje prozora i izgleda u Unityju

Unity pruža iznimno prilagodljivo korisničko sučelje koje korisnicima omogućava prilagođavanje rasporeda prozora i alata prema vlastitim potrebama.

U nastavku će biti prikazan način prilagodbe prozora i izgleda u *Unityju*.

- **Pomicanje prozora**: svaki prozor unutar sučelja *Unity* može se preurediti povlačenjem njegove kartice. Potrebno je jednostavno kliknuti te povući karticu prozora na željenu lokaciju. Moguće je pridružiti ga drugim prozorima, postaviti ga kao zaseban plutajući prozor ili ga umetnuti između dvaju postojećih prozora.
- **Promjena veličine prozora**: da bi se promijenila veličina prozora, potrebno je postaviti pokazivač miša na rub prozora dok se ne pojavi dvostruka strelica. Nakon toga je potrebno kliknuti i povući prozor da bi se povećala ili smanjila njegova veličina.
- **Zatvaranje prozora:** da bi se zatvorio određeni prozor, potrebno je kliknuti na malu ikonu „x“ na kartici prozora.
- **Dodavanje novih prozora:** želi li se dodati nove prozore, potrebno je otići na glavni izbornik i odabrati *Window*. Tu je moguće vidjeti popis svih dostupnih prozora i alata. Klikom na željeni prozor, dodat će ga se u radno sučelje.



Slika 18. Opcije za prilagođavanje prozora (slika zaslona, Unity)

- **Korištenje predloženih izgleda:** *Unity* nudi nekoliko predefiniranih izgleda. Izgledima je moguće pristupiti klikom na ***Layout*** u desnom gornjem kutu glavnog sučelja. Tu se može izabrati između raznih izgleda, poput ***Default***, ***Tall***, ***Wide*** i drugih.

- **Spremanje i učitavanje vlastitih izgleda:** nakon što je sučelje prilagođeno prema specifičnim potrebama određenog zadatka, odnosno projekta, moguće je spremiti izgled za kasniju upotrebu. Da bi se to učinilo, potrebno je otići na *Layout* i odabrati ***Save Layout***. Da bi se prethodno spremljeni izgled učitao, potrebno je otići na *Layout*, i odabrati željeni izgled iz popisa.

Prilagodba prozora i izgleda u *Unityju* omogućava svakom korisniku da stvori optimalno radno okruženje za svoje potrebe. Bez obzira radi li se na skriptiranju, grafičkom dizajnu ili izradi nivoa, pravilno postavljeno sučelje može znatno poboljšati produktivnost.

4.2.2. Navigacija u Unityju

Navigacija *Unityjevom* scenom ključna je vještina svakoga razvojnog inženjera koji se koristi tim alatom. Kada se otvori „Prikaz scene“ (eng. *scene view*) u *Unityju*, moguće se kretati, zumirati i rotirati objekt za njegov pregled i uređivanje unutar scene.
U nastavku su objašnjene osnovne metode navigacije.

- **Pomicanje (eng. *pan*):** potrebno je držati pritisnutu tipku *Alt* ili *Option* na *Mac* računalu te se koristiti lijevom tipkom miša da bi se pomicao pogled oko scene.
- **Rotacija (eng. *orbit*):** potrebno je držati pritisnutu tipku *Alt* i koristiti se srednjom tipkom miša ili kliknuti na kotačić kako bi se rotirao pogled oko trenutne točke fokusa.
- **Zumiranje:** potrebno se koristiti kotačićem miša za zumiranje unaprijed ili unazad. Također, može se držati tipka *Alt* i koristiti desnom tipkom miša kako bi se zumiralo prema unutra ili van.

- **Fokusiranje objekta:** u slučaju da je potrebno brzo zumirati ili fokusirati određeni objekt u sceni, potrebno ga je jednostavno odabrati i pritisnuti tipku „**F**“. To će centrirati i zumirati pogled na odabrani objekt.
- **Slobodna kamera:** potrebno je držati desnu tipku miša (eng. *right mouse button*) i koristiti se tipkama W, A, S, D za slobodno kretanje kroz scenu, slično kao u mnogim videoigrama. Također, moguće se koristiti s Q i E za kretanje prema gore i dolje.

Brz odabir alata omogućuju sljedeće tipke:
Q – pomicanje
W – alat za pomak
E – alat za rotaciju
R – alat za skaliranje
T – alat za *rect transform*; koristi se uglavnom za grafičko sučelje (engl. *User interface* ili UI) elemenata.

> ⚠ **VAŽNO ZNATI**
> Na dnu prozora „prikaz scene“ (eng. *scene view*) moguće je pronaći mali gumb u obliku ručne kamere. Klikom na njega otvara se padajući izbornik s nekim od gore navedenih alata za navigaciju, kao i dodatne postavke koje utječu na izgled i ponašanje „prikaza scene“ (eng. *scene view*).

Vježba je ključna za glatku navigaciju *Unity* scenom. Što se više vremena kreće „prikazom scene“ (eng. *scene view*), to će postupak postati intuitivniji i brži.

4.3. Scene u Unityju

U *Unityju* scena predstavlja prostor gdje se kreiraju i organiziraju objekti igre ili aplikacije koju se razvija. **Scena** je osnovno radno okruženje u kojem se postavlja, uređuje i dizajnira virtualne svjetove, objekte i interakcije.
Evo detaljnijeg objašnjenja o scenama u *Unityju*:

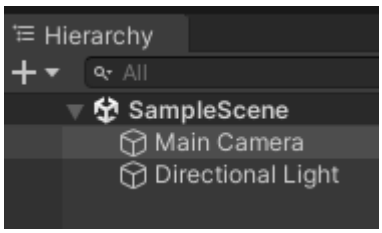
- **osnovni prikaz**: scena pruža grafički prikaz kreiranoga virtualnog svijeta. U njemu je moguće direktno dodavati, uređivati ili uklanjati objekte, postaviti

osvjetljenje, kamere, igrače, neprijatelje i sve ostale komponente koje igra ili aplikacija koju se kreira zahtijeva

- **upravljanje objektima**: unutar scene je moguće organizirati objekte u hijerarhiji. To omogućuje lakše upravljanje grupama objekata, primjerice grupa objekata koji čine određenu lokaciju ili funkcionalnost
- **višestruke scene:** u *Unityju* je moguće imati više scena unutar jednog projekta, što je korisno za segmentiranje različitih dijelova igre, kao što su različite razine ili meniji. S *Unityjevim* sustavom za upravljanje scenama, moguće je dinamički učitavati i isključivati scene dok igra radi, što omogućuje stvaranje prostranih virtualnih svjetova ili epizodnih igara
- **prijelaz između scena**: skriptiranjem je moguće kreirati prijelaze između različitih scena. Na primjer, kada igrač završi jednu razinu, može automatski učitati sljedeću scenu ili razinu
- **spremanje i učitavanje**: kada se rade promjene unutar scene, važno je redovito spremanje (pohranjivanje) izvršenih promjena. Ukoliko se želi isprobati nešto novo ili eksperimentirati s postavkama, utoliko je moguće jednostavno napraviti kopiju trenutne scene i raditi promjene na toj kopiji. To omogućuje povratak na originalnu scenu ako nešto pođe po zlu.

Scene su osnovna jedinica dizajna u *Unityju*. Sve što se vidi i interagira s njom u igri ili aplikaciji, od okruženja do likova, kreirano je i organizirano unutar jedne ili više scena. **Razumijevanje kako se koristiti i upravljati scenama ključno je za uspješan razvoj u *Unityju*.**

4.3.1. Uzorak scene (Sample scene)



Slika 19. Objekti primjerne scene (slika zaslona, Unity)

Kada se započinje novi 3D osnovni projekt u *Unityju*, u „Uzorku scene“ (eng. ***sample scene***) automatski su kreirane dvije osnovne komponente: glavna kamera (eng. ***main camera***) i usmjerena svjetlost (eng. ***directional light***).

Glavna kamera predstavlja oči gledatelja ili korisnika unutar 3D scene. Određuje što će korisnik vidjeti kad pokrene kreiranu igru ili aplikaciju. Uz pomoć njezinih postavki

moguće je kontrolirati kako će se određena scena prikazivati, bilo da se radi o perspektivi, kutu gledanja, dubini polja ili drugim vizualnim aspektima. S druge strane, usmjerena svjetlost služi kao globalni izvor svjetlosti u sceni. Ta se svjetlost često koristi za simulaciju svjetla sunca ili nekoga drugog udaljenog i velikog izvora svjetlosti. Usmjerena svjetlost utječe na sve objekte unutar scene, dajući im osvjetljenje i sjene na temelju položaja i orijentacije svjetla. Ta svjetlosna komponenta ključna je za postizanje realnih vizualnih efekata i doprinosi općem osjećaju i atmosferi 3D scene.
Navedene dvije automatski kreirane komponente osiguravaju da nova scena odmah ima osnovne elemente potrebne za vizualizaciju i osvjetljenje, omogućujući da se odmah započne s razvojem projekta. Kada se u *Unityju* klikne na glavnu kameru (eng. *main camera*) unutar „Hijerarhije“ ili „Scene“, u prozoru „**Inspektor**“ (eng. *inspector*) otkrit će se njezine komponente.

4.4. Komponente

U kontekstu računalnog programiranja i razvoja softvera, komponente su samostalne jedinice koje sadrže određene funkcionalnosti ili ponašanje. Navedene jedinice mogu se koristiti samostalno ili u kombinaciji s drugima za kompleksnu funkcionalnost. Koncept komponenti često je povezan s modularnošću i ponovnom upotrebom koda. U kontekstu razvojnih alata za igre i aplikacije poput *Unityja*, komponente imaju specifičnu definiciju. U *Unityju* svaki objekt u sceni (poznat kao *GameObject*) sastoji se od komponenata koje definiraju njegovo ponašanje i svojstva.
Na primjer:

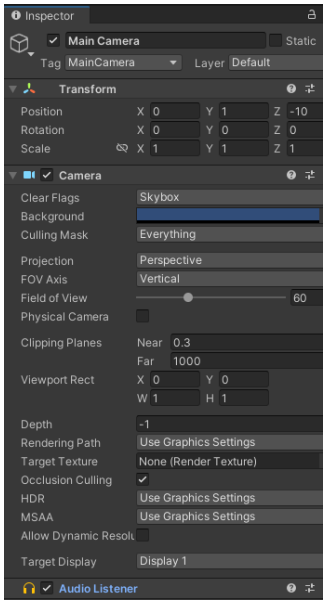
- **komponenta *Transform***: sadrži informacije o poziciji, rotaciji i veličini objekta
- **komponenta *Renderer*:** kontrolira kako će se objekt vizualno prikazivati
- **komponenta *Collider*:** definira prostornu zonu unutar koje se mogu detektirati sudari s drugim objektima
- **komponenta *Rigidbody*:** dodaje fizikalna svojstva objektu, poput gravitacije ili trenja.

Osim ugrađenih komponenata koje nudi *Unity*, korisnici mogu kreirati i svoje prilagođene komponente skriptiranjem, što omogućuje definiranje jedinstvenih ponašanja i interakcija.

Jedna je od ključnih prednosti komponenata u *Unityju* „kompozicijski“ pristup dizajnu. Naime, umjesto da se stvori složena hijerarhija nasljeđivanja, objekte je moguće jednostavno sastaviti dodavanjem ili uklanjanjem komponenata za postizanje željene funkcionalnosti. To omogućuje veću fleksibilnost i modularnost u dizajnu igre ili aplikacije.

4.4.1. Kamera

U *Unityju* je komponenta kamere (eng. *camera*) ključni alat koji simulira pogled očiju korisnika unutar kreirane scene. Ona definira što će se točno prikazati na ekranu.



Slika 20. Komponenta kamere (slika zaslona, Unity)

Evo nekoliko važnih opcija kamere i njezinih funkcija:

projekcija (eng. *projection*): ta opcija omogućuje odabir između dvaju načina prikaza:

- **perspektiva (eng. *perspective*):** pruža dubinsku percepciju, što znači da će objekti koji su bliže kameri, izgledati veće, dok će udaljeniji objekti izgledati manje
- **ortografski (eng. *orthographic*):** taj način prikaza oduzima dubinsku percepciju. Svi objekti, bez obzira na udaljenost od kamere, prikazat će se u stvarnoj veličini

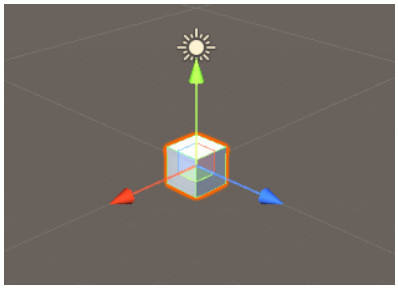
- **vidno polje (eng. *field of view*):** odnosi se na kut (izražen u stupnjevima) iz kojega kamera „gleda“. Veći kut omogućuje širi pogled, dok manji kut pruža usmjereniji, „zumiran“ pogled. Ta je opcija dostupna samo kada je postavljena perspektivna projekcija
- **isječne ravnine (eng. *clipping planes*):** dvije vrijednosti: blizu (eng. *near*) i daleko (eng. *far*) određuju koje će udaljenosti biti prikazane. Objekti koji su bliže kameri od *near* ravnine ili dalje od *far* ravnine, neće biti prikazani
- **prikazni pravokutnik (eng. *viewport rect*):** određuje dio ekrana koji kamera zauzima, omogućujući postavljanje više kamera koje dijele isti ekran
- **pozadina (eng. *background*):** omogućuje odabir boje koja će se koristiti za dijelove ekrana koji nisu prekriveni bilo kojim objektom u sceni.

Uz kameru, često se može primijetiti komponentu nazvanu „slušatelj zvuka“ (eng. ***audio listener***). Ta komponenta predstavlja „uho“ igrača unutar scene. U svakoj *Unity* sceni može postojati samo jedan „slušatelj zvuka“. Ako u sceni postoji više kamera s tom komponentom, moglo bi se naići na probleme s obradom zvuka. Slušatelj zvuka primjećuje sve izvore zvuka unutar scene i obradi ih prema njihovoj udaljenosti i položaju u odnosu na slušatelja, stvarajući 3D zvučni doživljaj.

4.4.2. Stvaranje 3D objekata

POKUŠAJTE SAMI

Treba najprije provjeriti nalazi li se u prikazu „Scena“ (eng. ***scene***). To je prozor gdje se radi većina 3D uređivanja i postavljanja. Potreban je desni klik na prozor „Hijerarhija“ (eng. ***hierarchy***), koji se obično nalazi s lijeve strane sučelja. Pokazivač miša treba pomaknuti preko opcije „3D Objekt“ (eng. ***3D Object***) u padajućem izborniku koji se pojavi te na popisu odabrati „Kocka“ (eng. ***cube***).

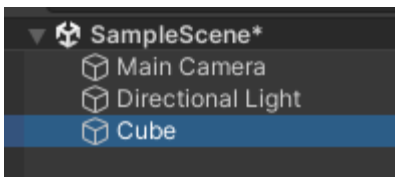


Nakon što je kocka stvorena, ona će biti automatski centrirana u sceni. Moguće se koristiti alatima za pomicanje, rotaciju i skaliranje, koji se nalaze na vrhu sučelja, za promjenu položaja, orijentacije ili veličine kocke prema potrebama.

Slika 21. 3D kocka (slika zaslona, Unity)

Manipuliranje kockom u *Unityju* može biti intuitivno jednom kada se upozna s alatima i njihovim prečacima na tipkovnici. Kako je moguće skalirati i premještati kocku koristeći se tim prečacima, bit će objašnjeno u nastavku.

Odabir kocke:



Da bi se odabrala kocka, potrebno je kliknuti na kocku unutar prozora „Hijerarhija“ (eng. ***Hierarchy***) ili direktno u prikazu „Scena“ (eng. ***Scene***). Također, koriste se prečaci alata za pomicanje i skaliranje.

Slika 22. 3D kocka u hijerarhiji (slika zaslona, Unity)

Na vrhu sučelja *Unity* alati su za manipulaciju, ali se umjesto pritiska na njih, mogu koristiti sljedeći prečaci na tipkovnici:

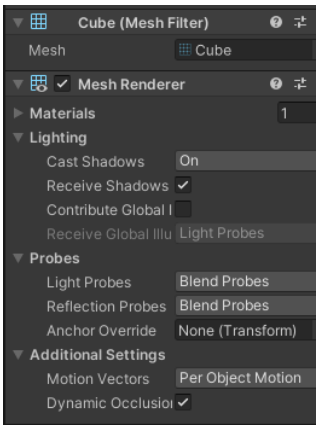
- **alat za pomicanje (eng. *Move Tool*):** potrebno je pritisnuti tipku „W“ na tipkovnici. Kada je taj alat aktiviran, na odabranom objektu pojavit će se strelice. Povlačenjem tih strelica moguće je pomaknuti kocku duž odabrane osi
- **alat za skaliranje (eng. *Scale Tool*):** potrebno je pritisnuti tipku R. Kada je taj alat aktiviran, na kocki će se pojaviti kontrolne točke za svaku os. Povlačeći te točke, može se skalirati kocku duž određene osi. Povlačenjem središnjega kockastog kontrolora može se proporcionalno skalirati kocku u svim smjerovima.

⚠ **Precizna prilagodba u prozoru „Inspektor“**

S obzirom na to da je kocka odabrana, pozornost bi trebala biti usmjerena prema prozoru *Inspector* s desne strane. U njemu je moguće uočiti komponentu pod engleskim nazivom *Transform*. Proučavajući sekciju „Pozicija“ (eng. *position*), omogućen je unos konkretnih vrijednosti za X, Y i Z osi, postavljajući time kocku na točno određenu poziciju u prostoru. Dalje, u sekciji „Skaliranje“ (eng. *Scale*), unosom specifičnih vrijednosti za X, Y i Z osi, precizno će biti skalirana kocka. Kombinacijom prečaca na tipkovnici i opcija dostupnih u prozoru *Inspector*, moguće je s lakoćom manipulirati kockama specifičnog *Unity* projekta, čime proces razvoja postaje znatno brži i učinkovitiji.

4.4.3. Mesh Renderer

Mesh Renderer komponenta je u *Unityju* ključna za vizualizaciju 3D objekata . Bez nje oblik definiran mrežom (eng. *mesh*) objekta ostao bi nevidljiv unutar scene, čak i ako



Slika 23. Komponenta „Mesh Renderer“ (slika zaslona, Unity)

objekt postoji i može interagirati s drugim objektima. U nastavku će biti detaljnije prikazan *Mesh Renderer* i njegova uloga na objektima poput kocke.

Mesh Renderer komponenta surađuje s *Mesh Filter* komponentom da bi prikazala mrežu objekta u igraćoj sceni s materijalima i teksturama. Dok *Mesh Filter* definira oblik objekta (u ovom slučaju kocke), *Mesh Renderer* je odgovoran za prikaz tog oblika na ekranu.

Materijali i sjenčari (eng. *Shader*): jedna od ključnih uloga *Mesh Renderera* određivanje je izgleda objekta, a to postiže pomoću materijala i sjenčara. Materijali definiraju vizualna svojstva objekta – njegovu boju, teksturu, sjaj, prozirnost i još mnogo toga. Ti materijali, potpomognuti sjenčarima, čine da objekti izgledaju realistično, crtano, metalno, prozirno ili postižu bilo koji drugi željeni vizualni efekt.

Na kocki je komponenta *Mesh Renderer* prisutna jer omogućuje vizualno prikazivanje kocke. Bez *Mesh Renderera*, kocka bi postojala unutar scene, ali ne bismo je mogli vidjeti.

4.5. 3D model

3D model digitalna je reprezentacija nekog objekta u trodimenzionalnom prostoru. Umjesto da bude ravna kao 2D slika, 3D model ima dubinu, širinu i visinu, što ga čini pogodnim za prikaz u virtualnim svjetovima, videoigrama, animacijama, simulacijama i mnogim drugim aplikacijama.

Osnovne karakteristike 3D modela uključuju:

- **poligone**: osnovna su građevna jedinica većine 3D modela poligoni, najčešće trokuti ili četverokuti. Složenost modela često se mjeri brojem poligona: više poligona znači detaljniji model, ali također zahtijeva više računalne snage za prikaz
- **vrhove**: točke gdje se rubovi poligona sijeku nazivaju se vrhovi (eng. *Vertex*). Oni definiraju strukturu i oblik 3D modela
- **rubove (eng. *Edges*):** linije koje povezuju vrhove i oblikuju poligone

- **teksture**: da bi se modelu dala boja, detalj ili realističan izgled, često se koristi teksturama. Tekstura je obično 2D slika koja se „omota“ oko modela. Proces mapiranja 2D teksture na 3D model naziva se **UV mapiranje**
- **normale**: vektorske linije koje izlaze iz vrhova ili poligona, koriste se za određivanje smjera iz kojeg površina modela „gleda“. To je ključno za osvjetljavanje i renderiranje jer određuje kako svjetlost interagira s površinom modela
- ***rigove* i kosti**: za animaciju modela, posebno likova, koristi se sustav kostiju ili „rig“. Kosti se postavljaju unutar modela i omogućuju mu da se kreće na realističan način. Kada se kosti pomaknu, tako se pomakne i dio modela povezan s njima.

3D modeli obično se kreiraju pomoću softvera, poput *Blendera*, *Autodesk Maya*, *3ds Maxa*, *ZBrusha* i mnogih drugih. Ti alati omogućuju umjetnicima i inženjerima da oblikuju i manipuliraju virtualnim objektima u 3D prostoru. Jednom kada je model izrađen, može se izvesti u razne formate koji se mogu koristiti u različitim aplikacijama, uključujući videoigre, animirane filmove, simulacije ili, čak, za 3D ispis.

4.5.1. Sjenčari

3D modeli i **sjenčari** usko su povezani u svijetu računalne grafike. Dok 3D model predstavlja strukturalni okvir i oblik objekta u trodimenzionalnom prostoru, sjenčari definiraju kako će se taj model vizualno prikazivati. Sjenčari utječu na boju, svjetlost, teksturu i druge vizualne aspekte modela, transformirajući osnovnu 3D strukturu u realističan ili stiliziran prikaz. Uz pomoć sjenčara 3D modeli mogu dobiti različite estetske karakteristike, poput metalnog sjaja, prozirnosti stakla ili grubosti kamena, čime se postiže željeni izgled i atmosfera u digitalnim kreacijama.

Sjenčar je poseban tip softverskog programa kojim se koristi u obradi grafičkih informacija, obično s ciljem izračunavanja boje i ostalih vizualnih svojstava piksela na zaslonu. Sjenčari se često koriste u računalnim igrama, filmovima, simulacijama i drugim aplikacijama koje koriste 3D grafiku, kako bi se postigli različiti vizualni efekti i realistični prikazi.

Sjenčare možemo klasificirati u nekoliko osnovnih kategorija:

1. ***vertex* sjenčar ili sjenčar vrhova:** obrađuje svaki vrh (eng. *vertex*) individualno. Može transformirati poziciju vrhova, promijeniti boju ili obaviti druge izračune koji utječu na završni izgled objekta u 3D prostoru
2. ***pixel* ili fragmentni sjenčar:** obrađuje svaki pojedini piksel koji će biti prikazan na zaslonu. Najčešće se koristi za izračun boje, osvjetljenja, sjenčanja i drugih vizualnih efekata koji se primjenjuju na svaki piksel
3. ***geometry* ili geometrijski sjenčar:** taj tip sjenčara omogućuje stvaranje novih geometrijskih oblika iz postojećih vrhova. Može, na primjer, transformirati jedan trokut u skupinu manjih trokuta
4. ***compute* sjenčar:** osmišljen je za izvođenje općih izračuna koristeći GPU. Iako nisu izravno povezani s prikazom, mogu se koristiti za razne zadatke, kao što su simulacije fizike ili postupci obrade slika
5. **teselacijski sjenčar (eng. *Tessellation Shader*):** omogućuje detaljniju subdiviziju površina, što može rezultirati finijim detaljima i glađim površinama.

Sjenčare obično piše programer ili tehnički umjetnik specijaliziran za grafiku. U jezicima poput **GLSL**-a (eng. *OpenGL Shading Language*) ili **HLSL**-a (eng. *High-Level Shading Language* za DirectX), sjenčari se mogu napisati za specifične grafičke potrebe.

U kontekstu razvojnih alata kao što je *Unity*, sjenčari omogućuju programerima umjetnicima kreiranje prilagođenih materijala i vizualnih efekata. Pomoću sjenčara moguće je simulirati niz efekata, od jednostavnog osvjetljenja i sjenčanja do složenih efekata, kao što su voda, staklo, koža, vatra i mnogi drugi. *Unityjeva* službena stranica za učenje nudi brojne vodiče i tečajeve koji se usredotočuju na sjenčare i vizualno programiranje. U suštini, sjenčari su ključna komponenta modernoga grafičkog *pipelinea*, omogućujući sofisticirane vizualne prezentacije i interaktivnost u digitalnom svijetu.

4.6. Unityjeva trgovina imovinom

***Unityjeva* trgovina imovinom (eng. *Asset Store*)** mrežna je trgovina koju je razvio *Unity Technologies* da bi omogućio korisnicima kupovinu i prodaju različitih resursa i alata koji se koriste unutar *Unity* razvojnog okruženja. Ti resursi, poznati pod terminom iz engleskoga govornog područja – *assets*, mogu uključivati sve, od 3D modela,

tekstura, animacija, do kompleksnih skripti, sjenčara, zvučnih efekata, glazbenih zapisa i još mnogo toga.

U nastavku će biti opisano nekoliko ključnih značajki i informacija o *Unityjevoj* trgovini imovinom.

- **Raznolikost imovine (eng. *assets*)**: trgovina nudi širok izbor imovine pogodne za različite vrste projekata, bilo da se radi o igrama, simulacijama, AR/VR aplikacijama ili bilo kojoj drugoj vrsti interaktivnog sadržaja.
- **Cjenovni raspon**: dok određena imovina može biti besplatna, druga se prodaje po različitim cijenama, ovisno o kompleksnosti, kvaliteti i drugim čimbenicima. Kupci mogu pregledavati trgovinu prema cjenovnom rasponu radi pronalaska imovine koja odgovaraju njihovu budžetu.
- **Kvaliteta i ocjene**: svaku imovinu u trgovini mogu korisnici ocijeniti i recenzirati, što pomaže potencijalnim kupcima donijeti informiranu odluku prije kupnje.
- **Integracija s *Unityjem*:** jednom kad se imovina preuzme iz trgovine, može se lako integrirati i koristiti unutar *Unity* projekta. Neka imovina dolazi s detaljnim uputama ili dokumentacijom za lakšu upotrebu.
- **Poticaj za programere:** trgovina pruža priliku nezavisnim programerima, umjetnicima i tvorcima da unovče svoje vještine i rad. Prodajom svoje imovine u trgovini, mogu zaraditi novac, dok istovremeno doprinose široj *Unity* zajednici.
- **Integracija popularnih imovina:** kada određena imovina postane posebno popularna ili se smatra izuzetno korisnom za širu zajednicu, *Unity Technologies* ponekad kupuje prava na nj. Nakon kupnje, ona se može integrirati izravno u *Unity* platformu, čineći je standardnom značajkom ili alatom unutar okruženja. To ne samo da ističe kvalitetu i vrijednost te određene imovine, već i pruža dodatnu vrijednost svim *Unity* korisnicima.

***Unityjeva* trgovina imovinom** postala je ključno mjesto mnogim *Unity* programerima širom svijeta. Pruža brz način dodavanja funkcionalnosti, vizualnoga efekta ili bilo kojega drugog resursa unutar projekta bez potrebe za izradom svega od početka. Također, potiče suradnički duh unutar zajednice jer tvorci mogu dijeliti svoje kreacije s drugima.

4.6.1. Preuzimanje imovine iz Unity trgovine

U traci za pretraživanje potrebno je unijeti ključnu riječ ***Medieval Windmill*** ili poveznicu (https://assetstore.unity.com/packages/3d/medieval-windmill-89489) da bi se pronašla željena imovina.

Slika 24. Medieval Windmill3 (slika zaslona, Unity trgovina)

Preuzimanje i kupnja

1. korak: potrebno je kliknuti na imovinu i otvoriti njezinu stranicu.
2. korak: ako imovina nije prethodno kupiljena ili preuzeta besplatno, potrebno je kliknuti na gumb ***Add to My Assets*** te slijediti upute za preuzimanje.
3. korak: potrebno jev kliknuti na opciju ***Open in Unity***. Zatim treba kliknuti na gumb ***Download*** u novootvorenom prozoru ***Package Manager***.
4. korak: nakon što je imovina preuzeta, gumb za preuzimanje promijenit će se u ***Import***. Potrebno je kliknuti na nj da bi se otvorio prozor za uvoz. Tu je moguće vidjeti sve datoteke uključene u paket imovine.
5. korak: potrebno je označiti sva polja. Zatim je potrebno kliknuti na gumb ***Import*** na dnu prozora.

Nakon uspješnog uvoza, nova imovina će se pojaviti u prozoru „Projekt“.

4.6.2. Materijal

Materijal u kontekstu računalne grafike i 3D modeliranja predstavlja kombinaciju tekstura, boja i sjenčara (eng. *shader*), koji zajedno definiraju kako će se površina objekta vizualno prikazivati. Materijali određuju izgled i karakteristike objekta, uključujući boju, sjaj, prozirnost, grubost i mnoge druge vizualne značajke. Kada se materijal primjenjuje na 3D model, on definira interakciju svjetlosti s tom površinom. Na primjer, materijal može simulirati izgled drva, metala, plastike, stakla ili bilo kojega drugog materijala u stvarnom svijetu. Uz pomoć sjenčara materijali mogu simulirati kompleksne vizualne efekte, poput refleksije, refrakcije, odbijanja svjetlosti i mnoge druge.

U razvojnim okruženjima, kao što je *Unity*, materijali se često koriste zajedno s teksturama i sjenčarima da bi se dobila željena estetika 3D objekata. Dok teksture pružaju detalje, kao što su uzorci i boje, sjenčari određuju kako će ti detalji reagirati na svjetlost i sjenu, a materijal kombinira sve te elemente za konačan izgled. Cilja li se na stvaranje površine mokre, sjajne stijene u *Unityju*, razmatranje o kombinaciji sjenčara (*shadera*), teksture i materijala ključno je za postizanje željenoga vizualnog efekta. Evo kako je moguće pristupiti tom zadatku:

- u kontekstu mokre, sjajne stijene potrebno je odabrati sjenčar koji podržava *specularity* (odnosno kako svjetlost interferira s površinama i kako se reflektira) i moguće refleksije da bi se to moglo simulirati
- za simuliranje površine stijene potrebno je koristiti teksturu koja imitira izgled kamena. Dodatno, može se koristiti i drugu teksturu, koja se usredotočuje na *specularity*, da bi se naglasili mokri dijelovi stijene koji reflektiraju svjetlost na specifičan način.

Nakon što su sjenčar i teksture odabrani, slijedi kreiranje materijala. Kada se kreira materijal, dodjeljuje mu se odabrani sjenčar, a zatim i teksture na odgovarajuća mjesta ili „mape“ kojima se sjenčar koristi (poput *albedo*, *specularity*, *normal map*, itd.). Također, prilagođavaju se druge postavke materijala, poput boje, sjajnosti i reflektivnosti, za preciznu kontrolu konačnoga izgleda.

4.6.3. Kolideri (eng. Colliders)

Kolideri su komponente u *Unityju* koje definiraju prostornu površinu unutar koje se može dogoditi fizička interakcija s drugim objektima. Svrha je kolidera detekcija sudara s drugim koliderima i omogućavanje fizičkih reakcija na te sudare.
Postoje različite vrste kolidera u *Unityju*, prilagođenih različitim oblicima i potrebama:

- ***Box Collider*:** taj kolider ima oblik pravokutnog paralelepipeda (kutije) i često se koristi za objekte poput zidova, poda ili bilo kojeg objekta s pravokutnim oblikom
- ***Sphere Collider*:** kružni kolider u obliku sfere, idealan za objekte poput loptica ili drugih sferičnih predmeta
- ***Capsule Collider*:** oblikovan kao kapsula, često se koristi za likove u igri zbog svog oblika koji se bolje prilagođava humanoidnoj formi

- ***Mesh Collider*:** taj se kolider koristi mrežom (eng. *mesh*) objekta da bi definirao oblik kolizije. Idealno za kompleksne oblike, ali je računalno zahtjevniji od osnovnih kolidera
- ***Terrain Collider*:** specifičan kolider za teren koji omogućava složene interakcije s terenskim objektima.

Kada dva kolidera dođu u kontakt, *Unity* može izračunati fizičku reakciju (ako su objekti postavljeni da reagiraju) ili jednostavno izazvati događaj. Na primjer, kada igračeva figura dodirne kolider koji predstavlja tlo, može se programirati da se igrač ne kreće dalje prema dolje ili da izazove neku drugu akciju, poput oštećenja igrača. Važno je napomenuti da sami kolideri ne uzrokuju fizičke reakcije. Za to je potrebna komponenta ***Rigidbody***. *Rigidbody* daje objektu svojstva mase, gravitacije i druga fizička svojstva, dok kolider samo definira gdje i kako se može dogoditi sudar. Kombinacija *Rigidbody* komponente i kolidera omogućava kompleksne fizičke interakcije u *Unity* igrama.

VJEŽBA: Kreiranje i vizualizacija 3D objekta u *Unityju*

Cilj je kreirati 3D objekt po vlastitom izboru, oblikovati jedinstveni materijal i aplicirati ga na kreirani objekt te se pobrinuti da je kamera ispravno pozicionirana da bi objekt bio vidljiv pri pokretanju scene. Potrebno je razmotriti razne opcije unutar *Unityja*, eksperimentirati s različitim vizualnim svojstvima i kreirati scenu koja odražava vlastitu kreativnost i tehničko razumijevanje naučenoga.

4.7. Kreiranje skripti

Kada se započinje s razvojem projekta u *Unityju*, organizacija je projekta ključna za učinkovit rad. Jedna od najboljih praksi jest grupiranje sličnih datoteka unutar posebnih mapa, a to se posebno odnosi na skripte. U nastavku će biti detaljno objašnjeni koraci izrade skripti.

- **Stvaranje mape *Scripts*:** u prozoru „Projekt“, koji se obično nalazi na dnu sučelja *Unityja*, desnom tipkom miša potrebno je kliknuti na željeni prostor ili na glavnu mapu projekta. Kada se otvori kontekstualni izbornik, odabire se ***Create > Folder***. Nova će se mapa pojaviti, a nju je odmah potrebno preimenovati u „**Scripts**“. Na taj način svi programski kodovi bit će organizirani na jednom mjestu, a što značajno olakšava navigaciju i održavanje.

- **Stvaranje i imenovanje nove skripte:** nakon što je mapa „**Scripts**“ otvorena, potrebno je kreirati novu skriptu desnim klikom unutar mape i odabrati ***Create > C# Script***. *C#* je dominantni programski jezik u *Unity* okruženju. Kada je skripta stvorena, automatski će biti nazvana „**NewBehaviourScript**“. Važno je odmah preimenovati skriptu u nešto što odražava njezinu funkcionalnost. Ime skripte treba odgovarati imenu glavne klase unutar same skripte da bi se izbjeglo bilo kakve pogreške. Potrebno je skriptu imenovati na sljedeći način: „**RotateObject**“.
- **Uređivanje i dodavanje skripte objektu:** dvostrukim klikom na skriptu otvorit će se standardni uređivač koda, poput *Visual Studija*. Unutar uređivača moguće je početi unositi kôd. Kada je uređivanje završeno, vrlo je jednostavno dodati skriptu nekom objektu unutar izrađene scene. Navedeno je moguće postići povlačenjem skripte iz mape „**Scripts**“ na željeni objekt u hijerarhijskom prozoru (eng. *Hierarchy*) ili u prozoru „Inspektor“ kada je objekt odabran.

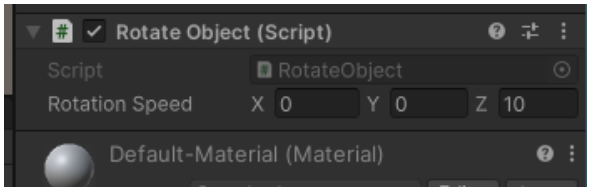
4.7.1. Rotiranje objekta

Unutar klase „RotateObject“ potrebno je definirati varijablu *Vector3*, koja će predstavljati brzinu rotacije na svakoj od osi.

```
public class RotateObject : MonoBehaviour
{
    [SerializeField] private Vector3 rotationSpeed;
```

U metodi „Update()“, potrebno je koristiti *Vector3* za kontinuiranu rotaciju objekta:

```
void Update()
{
        transform.Rotate(rotationSpeed * Time.deltaTime);
}
```



Slika 25. Komponenta „RotateObject“ (slika zaslona, Unity)

Pridruživanje skripte objektu: potrebno je povući skriptu „RotateObject“ i ispustiti je na *Cube* objekt u sceni u hijerarhijskom prozoru (eng. *Hierarchy*) ili dodati skriptu objektu u njegovu prozoru „Inspektor“.

Podešavanje brzine rotacije

Sada, s pridruženom skriptom, u prozoru „Inspektor“ postoji mogućnost prilagodbe vektoru „rotationSpeed“ za kontrolu brzine rotacije na svakoj osi (X, Y, Z). U nastavku je potrebno postaviti vrijednost Z osi na 10. Kada je scena pokrenuta, objekt s pridruženom skriptom rotirat će se na temelju definiranog vektora brzine rotacije.

Time.deltaTime vrlo je važan koncept u *Unityju*, posebno kada se radi s animacijama, pokretima ili bilo kojom vrstom kontinuirane promjene tijekom vremena. ***Time.deltaTime*** predstavlja vrijeme (u sekundama) koje je prošlo od posljednjega okvira (eng. *frame*) do trenutnog. To omogućuje da se objekti pokreću glatko i neovisno o frekvenciji osvježavanja, bilo da se igra koju izrađujemo pokreće pri 30, 60 ili više sličica u sekundi. Kada se neke vrijednosti množe s *Time.deltaTime*, zapravo se „normaliziraju“, tako da se kreću u kontinuiranu, glatkom tempu.

4.7.2. MonoBehaviour

Monoponašanje (eng. *MonoBehaviour*) osnovna je klasa iz koje sve *Unity* skripte nasljeđuju. Kada se programira u *Unityju*, monoponašanje pruža pristup brojnim funkcijama i događajima integriranima u životni ciklus objekta u igri ili okruženju koje se izrađuje. **Monoponašanje** (eng. *MonoBehaviour*) je ključna klasa u razvojnom okruženju *Unity,* a omogućuje pisanje skripti čvrsto integriranih s *Unity* motorom. Kada se stvara nova skripta u *Unityju*, ona obično nasljeđuje monoponašanje da bi se mogla koristiti njegovim funkcijama i događajima.

Nekoliko je ključnih funkcija monoponašanja.

Životni ciklus metode: skripte monoponašanja sadrže određene metode koje se automatski pozivaju tijekom različitih faza životnog ciklusa objekta. Primjeri uključuju *Awake()*, *Start()*, *Update()*, *FixedUpdate()* i mnoge druge.

Buđenje (eng. *Awake*): ta se funkcija poziva kada se skripta instancira. To se događa prije svih drugih metoda, poput *Start()*, i obično se koristi za postavljanje referenci ili druge inicijalne postavke.

Početak (eng. *Start*): *Start()* se poziva prije prvog ažuriranja okvira, ali nakon *Awake()*. Ta se metoda često koristi za inicijalizaciju varijabli ili postavljanje početnih stanja.

Prilikom omogućavanja (eng. *OnEnable*): ta se funkcija poziva kada je skripta inicijalizirana, što je uvijek prije metode *Awake()*, i svaki put kad postane aktivna ili omogućena.

Prilikom onemogućavanja (eng. *OnDisable*): poziva se kad skripta postane neaktivna ili onemogućena.

Ažuriranje (eng. *update*): *Update()* se poziva jednom po okviru. Ta se funkcija obično koristi za redovite radnje, poput provjere ulaza igrača ili pomicanja objekta.

Fiksno ažuriranje (eng. *FixedUpdate*): ta se metoda poziva u fiksnim vremenskim intervalima, neovisno o broju sličica u sekundi. Često se koristi za radnje povezane s fizikom, poput primjene sile na tijelo s rigidnim tijelom.

Kasno ažuriranje (eng. *LateUpdate*): *LateUpdate()* se poziva nakon svih *Update()* metoda u istom okviru. Korisno je za akcije koje moraju biti posljednje, poput praćenja kamere.

Prilikom uništenja (eng. *OnDestroy*): poziva se kada objekt sa skriptom treba biti uništen. Ta je metoda korisna za čišćenje ili obavljanje specifičnih akcija prije nego što se objekt u potpunosti ukloni.

Korisnički definirani događaji: osim standardnih metoda životnog ciklusa, može se definirati vlastite metode i funkcionalnosti unutar skripti koje nasljeđuju monoponašanje.

Komponentni model: u *Unityju* sve se funkcionalnosti dodaju objektima pomoću komponenti. Skripte koje nasljeđuju monoponašanje mogu se dodati objektima kao komponente, što omogućuje programerima prilagodbu ponašanja i funkcionalnosti objekata.

Komunikacija između skripti: pomoću monoponašanja skripte mogu lako komunicirati međusobno koristeći funkcije poput *GetComponent<>()* ili slanjem poruka.

Korutine: monoponašanje omogućuje korištenje korutinama, odnosno posebnim vrstama funkcija koje omogućuju prekid izvršavanja i povratak na točku prekida u kasnijim trenutcima, što je korisno za situacije poput čekanja ili postupnih animacija. Korutine su poputu malih pomoćnika koji dopuštaju izvođenje koda malo po malo, umjesto odjednom. Tako se omogućuje kašnjenje u izvršavanju koda, izvođenje koda u određenim intervalima, čekanje dok neki uvjet nije zadovoljen i više. U osnovi, korutine u *Unityju* omogućuju „pauziranje“ izvršavanja koda na određeno vrijeme ili dok neki uvjet nije zadovoljen, bez blokiranja izvođenja ostalih dijelova koda ili same igre.

4.7.3. Serialize Field

U *Unityju* je ***SerializeField*** vrlo koristan atribut koji omogućuje programerima izlaganje privatnih varijabli u *Unity* uređivaču bez potrebe da te varijable budu javne. Taj je pristup posebno važan za očuvanje načela **enkapsulacije**, dok se istovremeno omogućava jednostavno uređivanje vrijednosti varijabli u *Unity* sučelju.

Što je *SerializeField*?

U *C#* programiranju važno je očuvati enkapsulaciju. To znači da varijable trebaju biti privatne da bi se spriječilo neželjeno mijenjanje njihovih vrijednosti iz drugih dijelova koda. Međutim, u *Unityju* se često želi imati mogućnost mijenjanja vrijednosti tih varijabli u uređivaču (eng. *Editor*) za laku prilagodbu ponašanja komponenata i objekata. U tu svrhu koristi se *SerializeField*. Taj atribut omogućava da privatna varijabla postane vidljiva i uređivana u *Unity* uređivaču, dok ostaje skrivena i zaštićena od drugih klasa i skripti.

Kako se koristi *SerializeField*?

Primjenjuje se stavljanjem *[SerializeField]* iznad deklaracije varijable u skripti:

```
using UnityEngine;
public class PrimjerKlase : MonoBehaviour
{
    [SerializeField] private int broj;
    // Ovdje dolazi ostatak koda...
```

U tom primjeru broj je privatna varijabla, što znači da se ne može pristupiti izvan klase „PrimjerKlase“. Međutim, zbog atributa *SerializeField*, ta će se varijabla pojaviti u „Inspektoru“ unutar *Unity* uređivača kada se odabere objekt koji ima tu skriptu dodanu kao komponentu.
U primjeru *RotateObject* „serijalizirana“ je varijabla *Vector3*.
Zaključno, *SerializeField* je alat koji kombinira najbolje od obaju svjetova:

- omogućuje programerima očuvati sigurnost i enkapsulaciju svog koda
- pruža fleksibilnost i jednostavnost uređivanja unutar *Unity* uređivača.

Medieval Windmill

Nakon razumjevanja načina na koje funkcija ***Rotate*** i ***Time.deltaTime*** rade zajedno da bi omogućile glatko rotiranje objekata u *Unityju*, prelazi se na primjenu te funkcionalnosti na stvarnim objektima.
Treba zamisliti scenarij gdje je vjetrenjača u srednjovjekovnom okruženju i želi se kontinuirana rotacija njezinih lopatica, stvarajući dojam da je pokreće vjetar. Korištenje funkcijom *Rotate* bilo bi idealno za tu situaciju. Međutim, prije nego što se primijeni ta funkcija, potrebno je dodati vjetrenjaču u *Unity* scenu koju se priprema. A kako to učiniti? Uvozom iz *Unity* trgovine imovinom!

4.8. Prefab

U *Unityju* je ***prefab*** skraćenica za engleski termin *prefabricated object* ili na hrvatskom „unaprijed izrađeni objekt“, koji korisnici mogu spremiti s određenim postavkama i komponentama. Jednom kada je objekt spremljen kao ***prefab***, može se više puta koristiti unutar scene ili čak u različitim scenama, održavajući sve svoje postavke i komponente.
Glavne su karakteristike i prednosti *prefaba*:

- **ponovna upotreba**: jednom kada je *prefab* kreiran, može ga se lako i brzo ponovno koristiti u bilo kojem dijelu projekta bez potrebe za ponovnim postavljanjem svih svojstava
- **ujednačene izmjene:** ako se glavni *prefab* izmijeni, sve instance tog *prefaba* u projektu na kojem se radi automatski će se ažurirati. To je izuzetno korisno kada

se želi napraviti opće izmjene bez potrebe za ručnim ažuriranjem svake instance zasebno

- **organizacija**: *prefabi* pomažu u organizaciji i standardizaciji određenih objekata unutar projekta na kojem se radi, što olakšava razumijevanje i upravljanje resursima
- **složene strukture:** *prefab* može sadržavati više objekata s odnosima roditelj-dijete, što znači da je moguće imati kompleksne strukture spremljene kao jedan *prefab*.

Kako stvoriti *prefab*?

Da bi se stvorio ***prefab*** u *Unityju*, jednostavno je potrebno povući objekt iz „Hijerarhije“ u projektni prozor. Automatski će se spremiti kao *prefab* i dobit će plavu ikonu, što ga razlikuje od ostalih resursa.

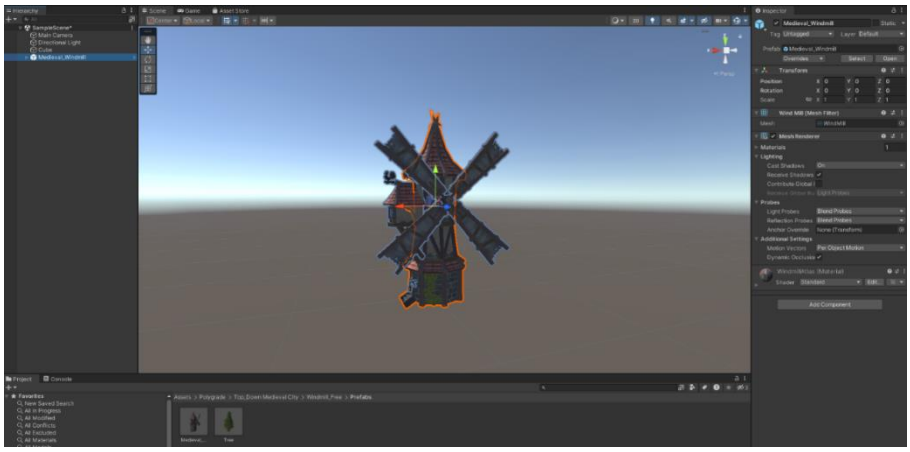
Prefabi su ključni alat u *Unity* razvojnom procesu, omogućujući razvojnim inženjerima da učinkovito i dosljedno koriste i ažuriraju objekte u različitim dijelovima njihova projekta. Na primjer, prilikom uvoza paketa, uobičajena je praksa tražiti *prefabe* za razumijevanje kako bi objekti trebali biti sastavljeni i međusobno funkcionirati unutar scene.

U ovom će dijelu biti objašnjeno kako napuniti scenu objektima iz projekta.

⚠ **Postavljanje 3D scene**

Pronalazak *prefaba*: u projektnom prozoru treba pretražiti ili doći do direktorija gdje je *prefab* pohranjen. ***Assets > Polygrade > Top_Down Medieval City > Windmill_Free> Prefabs > Medieval_Windmill***. *Prefabi* obično imaju plavu ikonu koja ih razlikuje od drugih objekata.

Povuci i ispusti: kada je pronađen *prefab* koji se želi dodati u scenu, treba kliknuti na njega i držati pritisnutu lijevu tipku miša, povući p*refab* iz projektnog prozora direktno u hijerarhijski prozor (eng. *Hierarchy*).



Slika 26. Scenski prikaz (slika zaslona, Unity)

Scena koja se izrađuje trebala bi izgledati kao na slici 4.12, s objektom (*Windmill*) na poziciji 0, 0, 0. Zatim je potrebno izbrisati stari objekt *Cube*.

Zvjezdica (*) pokraj imena scene u *Unity* uređivaču ukazuje na to da su u sceni napravljene promjene koje još nisu spremljene. To je vrlo korisna vizualna povratna informacija jer upozorava razvojni tim da postoje nespremljene modifikacije koje bi mogle biti izgubljene ako se scena ne spremi prije zatvaranja ili prebacivanja na drugu scenu.

Za spremanje promjena u sceni moguće se koristiti različitim metodama.

Moguće je kliknuti na riječ *File* u gornjem izborniku i odabrati *Save Scenes* ili *Save Project*. Drugi je način koristiti se prečacima na tipkovnici: pritisnuti „Ctrl + S“ (na *Windowsu*) ili „Cmd + S“ (na operacijskom sustavu *MacOS*) za brzo spremanje promjena u trenutačnoj sceni. Ukoliko se želi spremiti cijeli projekt, što uključuje sve resurse, postavke i scene, utoliko je moguće koristiti neku od sljedećih kombinacija tipki „Ctrl + Shift + S“ ili „Cmd + Shift + S“.

4.8.1. Pozicioniranje kamere

Postavljanje kamere ključna je komponenta prilikom kreiranja scene u *Unityju*. Nakon što je preuzet i dodan model vjetrenjače iz *Unity* trgovine imovinom u scenu koju se izrađuje, sve je postavljeno i spremno za vizualizaciju. Međutim, kada se pritisne gumb *Play* za pregled scene kroz kameru, kamera nije pravilno postavljena i ne može se vidjeti vjetrenjaču. U takvim situacijama prilagodba i postavljanje kamere tako da točno snima ono što želimo, predstavlja osnovni korak. U *Unity* uređivaču postoji mogućnost brzog postavljanja kamere tako da gleda točno ono što trenutno vidimo u scenskom prikazu (eng. *Scene View)*. To je posebno korisno kada se želi postići da kamera

odmah snima određeni objekt ili scenu iz perspektive s koje se trenutno gleda. Prečac „Ctrl + Shift + F“ omogućava tu funkcionalnost.

U nastavku je opisano postupak.

1. korak – odabrati kameru: u hijerarhijskom prozoru (eng. *Hierarchy*) potrebno je odabrati kameru koju se želi pozicionirati.

2. korak – postaviti željeni pogled: u prikazu scene (eng. *Scene View*) može se upravljati i postaviti pogled tako da se gleda točno ono što se želi da kamera snima.

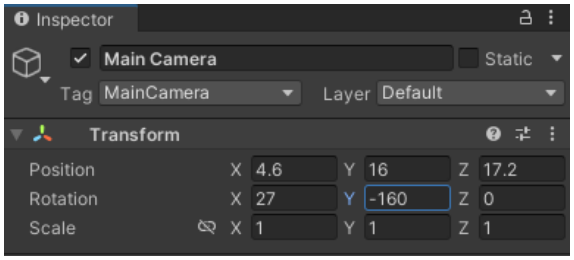
3. korak – koristiti prečac: u prikazu scene (eng. *Scene View*) potrebno je pritisnuti kombinaciju tipki „**Ctrl + Shift + F**“.
Nakon pritiska tih tipki kamera će se automatski premjestiti na poziciju i orijentaciju trenutnog pogleda u prikazu scene (eng. *Scene View*). Sada, kada se prijeđe u *Game View,* ili se pokrene scena, kamera će snimati točno ono što je postavljeno. To može značajno ubrzati radni tijek, bez potrebe za ručnim prilagođavanjem pozicije i rotacije kamere.

Slika 27. Prikaz kamere (slika zaslona, Unity)

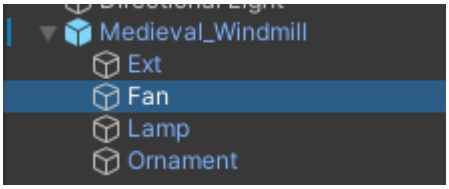
⚠ VAŽNO JE ZNATI

Pozicioniranje kamere u *Unityju* predstavlja izuzetno važnu vještinu s obzirom na to da se u 3D svijetu ona može natjerati da radi stvari koje u stvarnom životu ne bi mogla učiniti, ili bi bile vrlo teške za realizaciju. Kamera u virtualnom prostoru nije ograničena fizičkim zakonima i mogućnostima stvarnih kamera. To znači da je moguće eksperimentirati s perspektivama, kretanjima i efektima koji bi bili nemogući ili izuzetno složeni u tradicionalnim filmskim produkcijama. Na primjer, kamera može prolaziti kroz objekte, letjeti na nedostižne visine ili se kretati nevjerojatnim brzinama bez ikakvih posljedica na „opremu“ ili na „snimatelja“. Ta sloboda omogućuje stvaranje dinamičnih, intrigantnih i jedinstvenih vizualnih prikaza u 3D projektima, bilo da se razvijaju videoigre, animacije ili interaktivne prezentacije.

Potrebno je postaviti kameru na pogled prikazan na slici 4.13. Drugi je način postaviti komponentu kamere *Transform* na sljedeće vrijednosti prikazane na slici 4.14. Sada kada je kamera prilagođena i postavljena, koristeći se prečacom „**Ctrl + Shift + F**“, može se usredotočiti na vjetrenjaču. Kada se pritisne gumb *Play* u *Unityju*, moguće je vidjeti da kamera sada pravilno snima vjetrenjaču, pružajući željeni kadar.



Slika 28. Komponenta „Transform“ glavne kamere (slika zaslona, Unity)



Slika 29. Objekt „Fan“ (slika zaslona, Unity)

U nastavku je potrebno otvoriti objekt ***Medieval_Windmill*** i na objekt „Fan“ staviti skriptu ***RotateObject*** s navedenim vrijednostima. Potrebno je pritisnuti tipku ***Play*** te bi se zatim vjetrenjača trebala okretati.

4.9. Zvuk

U ovom dijelu bit će objašnjeno kako se može dodati još jednu slojevitu dimenziju vjetrenjači: zvuk. S ciljem uživljavanja i realnijega doživljaja, uvest će se zvučna datoteku koja će oponašati šum vjetrenjače. U *Unityju* zvuk ima ključnu ulogu u stvaranju snažnijega korisničkog iskustva. Bilo da se radi o pozadinskoj glazbi, zvučnim efektima ili govornim isječcima, *Unity* nudi snažne alate za upravljanje i manipulaciju audiosadržajem.

Osnovne informacije o upravljanju audiosadržajem u *Unityju*:

- **audioisječak (eng. *Audio Clip*)**: osnovna jedinica zvučnog sadržaja u *Unityju* je audioisječak. To je datoteka sa zvukom koji se želi reproducirati. Audioisječci se mogu uvesti iz različitih izvora, kao što su MP3, WAV ili OGG
- **audioizvor (eng. *Audio Source*)**: da bi se reproducirao audioisječak u projektu na kojem se radi, bit će potrebna komponenta audioizvor. Nju se dodaje na *GameObject* i dodjeljuje joj audioisječak. Ta komponenta kontrolira reprodukciju zvuka, njegovu glasnoću, paniranje i mnoge druge parametre
- **audioslušatelj (eng. *Audio Listener*)**: da bi zvuk bio čujan u scenariju, potrebno je imati komponentu audioslušatelj. Tipično, on se postavlja na glavnu kameru jer predstavlja „uši“ igrača u svijetu. U većini igara postoji samo jedan audioslušatelj
- **3D zvuk (eng. *3D sound*)**: jedna od naprednih značajki audiosustava u *Unityju* je sposobnost stvaranja 3D zvuka. To omogućuje da zvučni efekti dobiju prostornu dimenziju, odnosno da zvuče kao da dolaze iz određene lokacije u 3D prostoru. Na primjer, moguće je postaviti zvuk koraka koji će se približavati igraču kako se neprijatelj približava
- **mikseri i grupiranje:** *Unity* nudi *Audio Mixer* za složeniju kontrolu nad zvukom. Omogućuje grupiranje zvukova, dodavanje efekata i kontrolu nad različitim aspektima zvuka pomoću klizača i drugih kontrola. Na primjer, moguće je imati poseban „miks“ za pozadinsku glazbu i drugi za zvučne efekte
- **efekti**: *Unity* nudi različite zvučne efekte koji se mogu primijeniti na zvuk, kao što su *reverb*, *ekvilizacija*, kompresija i mnogi drugi
- **optimizacija**: zvukovi mogu zauzeti puno memorije, pogotovo ako je u projektu korišteno puno visokokvalitetnih zvučnih datoteka. *Unity* nudi alate za

optimizaciju, poput promjene brzine prijenosa (eng. *bit rate*) ili kompresije zvuka, da bi se osiguralo da igra ili aplikacija radi glatko.

Zvuk je ključan element bilo koje igre ili multimedijske aplikacije, pružajući dodatnu razinu realnosti i angažiranja korisnicima. S pravilnim alatima i tehnikama koje nudi *Unity*, programeri mogu kreirati bogata i živa zvučna iskustva u svojim projektima.

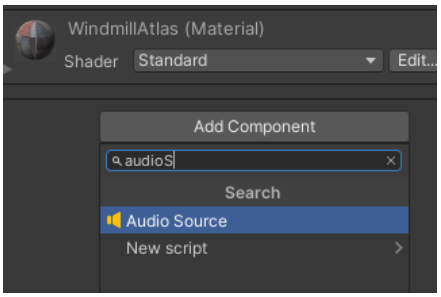
4.9.1. Uvoz zvučne datoteke

Da bi se to postiglo, potrebno je posjetiti mrežnu stranicu *Freesound*, gdje je moguće preuzeti odgovarajući zvučni zapis. Uvesti zvučnu datoteku u *Unity* projekt jednostavan je i organiziran proces. Koraci su sljedeći:

1. korak – kreiranje mape za audio (zvuk): prije nego što se uveze zvučna datoteka, preporučljivo je kreirati posebnu mapu unutar projekta da bi sve zvučne datoteke bile na jednom mjestu. Da bi se to postiglo, desnom tipkom miša potrebno je kliknuti unutar prozora *Project* u *Unity* sučelju i odabrati *Create*" > "*Folder*.

2. korak – uvoz zvučne datoteke: nakon kreiranja zvučne mape, potrebno je preuzeti zvučnu datoteku s *Freesounda* na računalo. Zatim povući i ispustiti datoteku iz mape na računalu direktno u prethodno kreiranu zvučnu mapu unutar *Unity* prozora *Project*. *Unity* će automatski prepoznati zvučnu datoteku i uvesti je s pravilnim postavkama. Kada se završi s uvozom, zvučna će datoteka biti spremna za upotrebu unutar *Unity* projekta.

4.9.2. Dodavanje i podešavanje zvuka

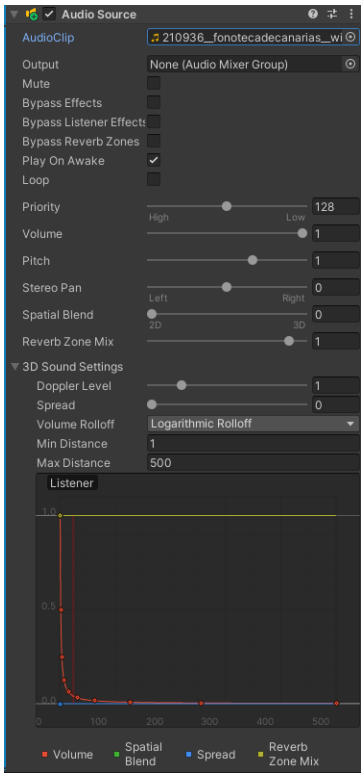


Slika 30. Dodavanje komponente (slika zaslona, Unity)

Nakon što je uspješno uvezena zvučna datoteka u *Unity*, slijedi postupak dodavanja komponente *AudioSource* na vjetrenjaču i postavljanje zvučne datoteke da se reproducira u petlji.

Odabir vjetrenjače u sceni: u hijerarhijskom prozoru (eng. *Hierarchy*) potrebno je kliknuti na objekt vjetrenjače da bi ga se odabralio. Objekt će biti označen i njegove će komponente biti vidljive u prozoru *Inspector*.

Dodavanje komponente *AudioSource*: s desne strane prozora *Inspector* potrebno je kliknuti gumb „dodaj komponentu“ (**engl. *Add Component***). U pretraživačkoj traci koja se pojavi, treba upisati *AudioSource*. Nakon što je komponenta pronađena, treba kliknuti na nju kako bi je se dodalo objektu vjetrenjače.



Slika 31. Komponenta audioslušatelja (slika zaslona, Unity)

U komponenti ***AudioSource*** koja se sada pojavljuje u prozoru *Inspector*, nalazi se opcija nazvana ***AudioClip***. Pored te opcije postoji mjesto gdje je moguće povući i ispustiti zvučnu datoteku. Zatim treba povući zvučnu datoteku iz prethodno kreirane mape „**Audio“** i ispustiti je u tu opciju.

Postavljanje zvučne datoteke s ciljem reprodukcije u petlji: ispod opcije ***AudioClip*** u komponenti ***AudioSource*** može se pronaći opcija označena kao ***Loop***. Potrebno je označiti tu opciju da bi se postavila zvučna datoteka te se kontinuirano reproducirala dok god je objekt aktivan u sceni.

Reprodukcija zvuka prilikom pokretanja scene: da bi se osigurala automatska reprodukcija zvuka kada se scena pokrene, nužno je osigurati da je opcija ***Play On Awake*** označena. Ako nije, treba kliknuti na nju za označavanje.

Sada, kada se pritisne ***Play*** u *Unityju* za testiranje scene, trebao bi se čuti zvuk vjetrenjače koji se reproducira u kontinuiranoj petlji, pružajući dodatnu razinu realizma projektu.

Trenutno, nakon što se doda zvučna datoteka u vjetrenjaču, može se čuti zvuk bilo gdje u sceni. Razlog tome je što je *AudioSource* postavljen na **2D** u postavkama prostornog zvuka. Dok je zvuk postavljen na 2D, on se reproducira jednako glasno bez obzira na udaljenost kamere ili igrača od izvora zvuka. Međutim, ako se želi postići realniji zvučni efekt, pri čemu zvuk postaje tiši kako se udaljava od vjetrenjače i glasniji kako joj se približava, trebalo bi promijeniti postavke zvuka na **3D**. Taj način stvara

prostorno zvučno iskustvo, dodajući dodatnu dimenziju realizma u kreiranu scenu. Naposlijetku, treba prebaciti ***Spatial Blend*** postavke na 3D.

4.10. Skybox

Kako bi se sceni dodalo više živosti i dubine, potrebno je koristiti ***skybox*.**

Skybox je tehnika u 3D računalnoj grafici za prikaz pozadine koja izgleda kao da okružuje cijelu 3D scenu. To je zapravo kubna tekstura koja okružuje kameru, i koja se prilagođava položaju i rotaciji kamere, pružajući osjećaj beskonačne udaljenosti. *Skyboxovi* su iznimno korisni za stvaranje ambijentalne atmosfere u sceni, bilo da se radi o plavom nebu s oblacima, svemirskoj sceni s planetima i zvijezdama, ili bilo kojem drugom dalekom okruženju. U *Unityju* se *skybox* može primijeniti na cijelu scenu u postavki *Lighting*, ali može se i dodijeliti specifičnoj kameri. Osim toga, *Unity* trgovina imovinom nudi mnoge gotove *skyboxove*, dok alati unutar *Unityja* omogućuju kreiranje vlastitih.

Slika 32. Scenski prikaz (slika zaslona, Unity)

Slijede koraci primjene *skyboxa* na scenu.

Kada se preuzimanje završi, potrebno je kliknuti na *Import* da bi se otvorio prozor za uvoz. Prikazat će se sve datoteke uključene u paket.

Važno: potrebno je sve kutije označiti da bi se uvezle sve komponente paketa, a zatim kliknuti na gumb *Import* na dnu prozora.

Nakon što je uspješno uvezen *skybox*, treba otvoriti *Window* (hrv. prozor) > *Rendering* (hrv. renderiranje) > *Lighting* (hrv. svjetlost).

U sekciji „Environment“ (hrv. okolina) treba pronaći opciju *Skybox Material* (hrv. *Skybox* materijal) i kliknuti na mali krug s desne strane da bi se otvorio prozor za odabir. Prikazat će se lista svih dostupnih *skyboxova* te je iz popisa potrebno odabrati ***SkyMidnight***.

Nakon što je željeni *skybox* odabran, automatski će se postaviti kao pozadina scene. Sada kad se scena pokrene, moći će se primjetiti da pozadina ima impresivnu kubnu teksturu *skyboxa*, koja dodaje mnogo više atmosfere i dubine u projekt. Ali isto tako se izgubilo i puno svjetla.

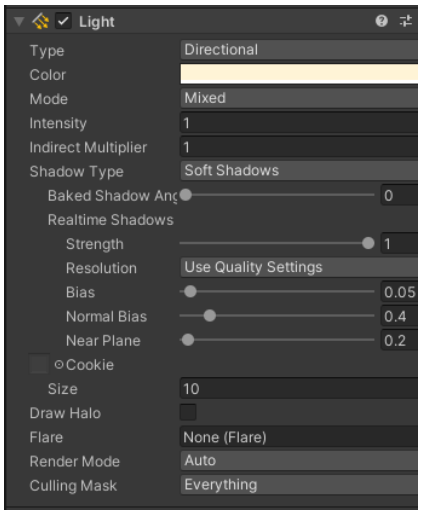
4.11. Svjetlo

U razvojnom okruženju *Unity*, svjetlo (eng. *light*) je ključna komponenta, koja omogućuje simulaciju svjetlosnih efekata u sceni. Svjetlost može dodati atmosferu, dubinu i realizam projektu. Postoji nekoliko različitih vrsta svjetala u *Unityju*, a svako od njih ima specifične postavke koje utječu na način osvjetljavanja objekata.

Vrste svjetala u *Unityju*

- **Usmjereno svjetlo (eng. *Directional Light*):** to svjetlo djeluje kao da dolazi iz beskonačne točke u jednom smjeru, poput sunca. Usmjereno svjetlo osvjetljava sve objekte u sceni i nema izvor ili pad.
- **Točkasto svjetlo (eng. *Point Light*):** izvor koji emitira svjetlost u svim smjerovima iz jedne točke u prostoru. Ima određen doseg unutar kojeg svjetlost osvjetljava objekte, a intenzitet svjetla slabi s udaljenošću od izvora.
- **Reflektorsko svjetlo (eng. *Spot Light*):** emitira svjetlost iz jedne točke u određenom smjeru s konusnim oblikom. Može se prilagoditi kut i intenzitet reflektorskog svjetla, kao i raspon unutar kojeg svjetlo utječe na objekte.
- **Površinsko svjetlo (eng. *Area Light*):** emitira svjetlost s cijele površine u određenom smjeru. Korisno je za simuliranje svjetla koje dolazi iz prozora ili većih svjetlosnih izvora.

Glavne postavke svjetlosne komponente:



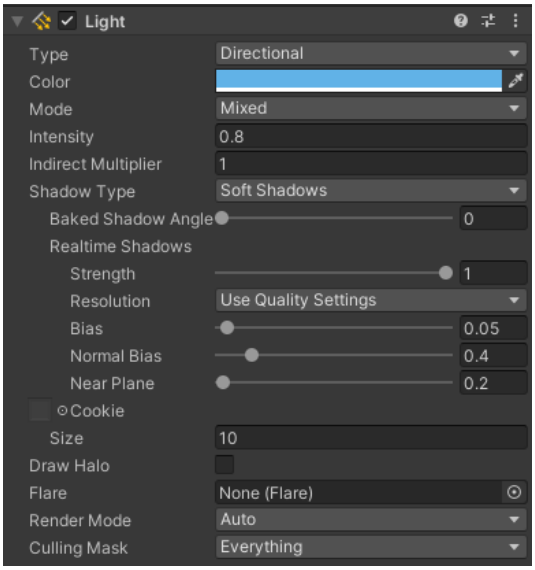
Slika 33. Komponenta „svjetlo“ (slika zaslona, Unity)

- **boja** (eng. ***color***): omogućuje odabir boje svjetla
- **intenzitet** (eng. ***intensity***): kontrolira jačinu svjetla. Veće vrijednosti posvjetljuju, dok niže vrijednosti potamnjuju
- **doseg** (eng. ***range***): za točkasta i reflektorska svjetla, ta postavka definira koliko daleko svjetlo može doći
- **kut reflektora** (eng. ***spot angle***): za reflektorska svjetla; definira kut konusa svjetla
- **sjene** (eng. ***shadows***): tom se postavkom određuje hoće li svjetlo stvarati sjene. Također je moguće prilagoditi kvalitetu i mekoću sjena.

Svjetla su temeljni dio grafičke prezentacije u *Unityju*, omogućujući razvojnim inženjerima da stvore realistične i atmosferske vizualne efekte. Kombinacijom različitih vrsta svjetala i prilagodbom njihovih postavki, može se postići širok spektar izgleda i stilova scena. Kada se želi postići noćnu atmosferu u sceni, važno je pravilno prilagoditi usmjerenu svjetlost (eng. *directional light*). Usmjerena svjetlost u *Unityju* funkcionira kao sunce u stvarnom svijetu, stoga njezino pravilno postavljanje može drastično promijeniti dojam cijele scene.

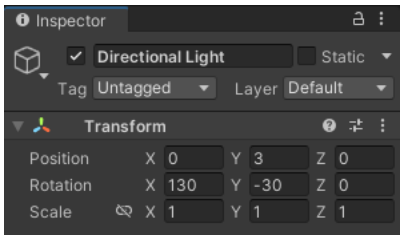
Koraci u postizanju atmosfere:

1. **odabir usmjerene svjetlosti:** u hijerarhijskom prozoru (eng. *Hierarchy*) potrebno je pronaći i odabrati usmjerenu svjetlost (eng. *Directional Light*)
2. **prilagodba intenziteta:** za postizanje noćne atmosfere treba smanjiti intenzitet svjetlosti. npr. na vrijednost 0.8. Ovisno o željenom efektu, može se poigravati tom vrijednošću
3. **promjena boje:** ako je svjetlost postavljena na bijelu ili toplo žutu (što oponaša dnevno svjetlo), potrebno ju je promijeniti da bi se dobila plavkasta nijansa koja više podsjeća na mjesečinu. U postavkama svjetlosti treba kliknuti na kvadrat boje i otvoriti paletu boja te odabrati nježnu plavu
4. **prilagodba kuta:** kut smjerne svjetlosti može utjecati na to kako sjenke padaju u sceni. Budući da je noć, možda će biti potrebno omogućiti da svjetlost dolazi iz drugačijeg kuta kako bi sjenke bile dulje ili kraće. Stoga se treba koristiti ručkama za rotaciju da bi se prilagodio kut
5. **testiranje u sceni:** nakon što su prilagodbe dovršene, treba pritisnuti *Play* za provjeru kako noćna atmosfera izgleda u pokretu. U slučaju da konačni rezultat ne zadovoljava, treba se vratiti i prilagoditi postavke dok se ne postigne željeni efekt.



Slika 36. Postavke komponente „svjetlo“ (slika zaslona, Unity)

Nakon tih prilagodbi, kreirana bi scena trebala imati mnogo više ugođaja noćne atmosfere, gdje svjetlo odražava osvjetljenje mjesečine umjesto dnevnog svjetla.



Slika 34. Komponenta „Transform“ svjetla (slika zaslona, Unity)

Slika 35. Scenski prikaz (slika zaslona, Unity)

Da bi se pravilno osvijetlila scena, potrebno je više od samo jedne svjetlosti. Međutim, važno je napomenuti da svjetla mogu biti zahtjevna za procesiranje, i mogu znatno utjecati na karakteristike igre ili aplikacije, posebno kada se koristi veliki broj svjetala s omogućenim sjenama. Svjetlo u 3D svijetu funkcionira slično kao u stvarnom svijetu. Jedan svjetlosni izvor može osvijetliti određeni prostor, ali dodavanje više izvora svjetlosti može pridonijeti bogatijoj i realnijoj atmosferi. Na primjer, uz glavnu svjetlost (poput sunca ili mjesečine) može se dodati sekundarna svjetla za naglašavanje određenih dijelova scene ili stvaranje mekših sjena.

Kada se dodaju svjetla, treba imati na umu sljedeće:

- **isključiti sjene gdje nisu potrebne**: ako svjetlo služi samo osvjetljavanju dijela scene, i sjene nisu presudne, treba ih isključiti radi poboljšanja značajki
- **koristiti *Baked Lighting* gdje je to moguće:** *Unity* nudi mogućnost „pečenja“ svjetla u teksture, što može znatno poboljšati značajke. To je idealno za statične scene ili objekte koji se ne miču
- **prilagoditi raspon svjetla:** smanjenje raspona svjetla znači da će svjetlo osvijetliti manji prostor, što može poboljšati značajke.

U svakom slučaju, potrebna je ravnoteža. Dok je važno imati dobro osvijetljenu scenu koja izgleda vizualno privlačno, također je važno osigurati da igra ili aplikacija radi glatko. Stoga je, prilikom dodavanja svjetala, nužno provjeravati značajke i prilagoditi ih prema potrebi.

4.11.1. Miješanje svjetala

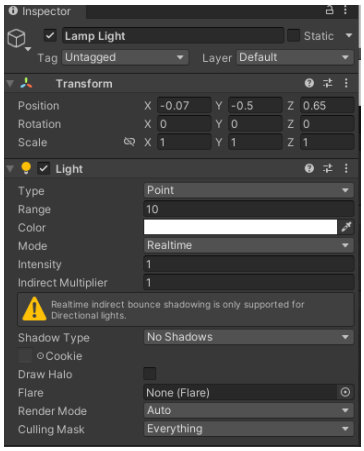
Unutar objekta ***Medieval_Windmill*** postoji drugi objekt pod nazivom ***Lamp***. Da bi se dodao novi, „prazan“ objekt (eng. *Empty Game Object*) unutar tog objekta, treba slijediti dolje navedene korake.

1. korak – odabir objekta *Lamp*: u hijerarhijskom prozoru (engl. *Hierarchy window*) unutar *Unityja* treba pronaći i odabrati objekt *Medieval_Windmill*. Nakon toga je potrebno proširiti objekt da bi se vidjeli svi njegovi potomci (objekti unutar njega), također i objekt *Lamp*. Potrebno je kliknuti na njega za odabir.

2. korak – stvaranje praznog objekta: desnim klikom miša na objekt *Lamp* otvorit će se kontekstualni izbornik. Iz tog izbornika treba odabrati ***Create Empty***. To će stvoriti nov, prazan objekt unutar objekta *Lamp*.

3. korak – preimenovanje (opcionalno): novokreirani prazni objekt automatski će dobiti ime poput *GameObject*. Ako se želi, može ga se preimenovati dvostrukim klikom na njegovo ime, ili desnim klikom te odabirom opcije ***Rename***. Za potrebe primjera u ovom priručniku nazvan je *Lamp Light*.

Praćenjem navedenih koraka uspješno je dodan nov, prazan objekt unutar objekta *Lamp*. Ti „prazni“ objekti često se koriste kao referentne točke ili „roditelji“ drugim objektima i komponentama koje će se željeti dodati kasnije. Sada kad postoji prazan objekt unutar objekta *Lamp*, želi se dodati svjetlosnu komponentu tom objektu i pozicionirati je na mjesto gdje bi izvor svjetlosti trebao biti u sceni. U nastavku je opisano kako je to moguće učiniti.

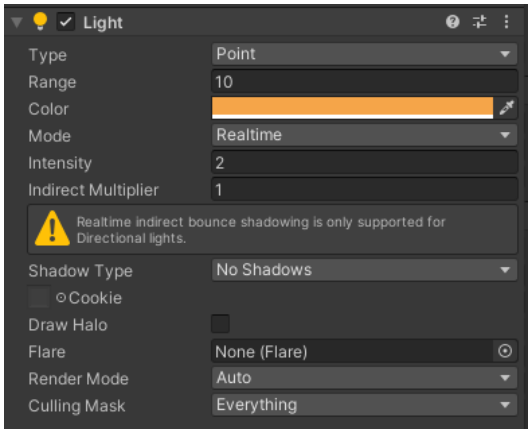


Slika 37. Komponenta „svjetlo“ (slika zaslona, Unity)

Dodavanje svjetlosne komponente: s odabranim praznim, ranije stvorenim objektom , treba pogledati inspektorski prozor (eng. *Inspector window*) s desne strane *Unity* sučelja. Zatim treba kliknuti na gumb ***Add Component*** („Dodaj komponentu“) na dnu inspektora. U pretraživaču koji se pojavi, treba upisati ***Light*** a kad se pojavi, odabrati ga. Na taj će način biti dodana komponenta svjetla tipa „točka“ ranije kreiranom praznom objektu.

Sada se, s dodanom svjetlosnom komponentom, želi podesiti njezin položaj tako da odgovara mjestu izvora svjetlosti u sceni. Da bi se to postiglo, u inspektorskom prozoru treba pronaći odjeljak *Transform*. U njemu je moguće ručno prilagoditi vrijednosti *Position* (hrv. „pozicija“) da bi svjetlo bilo na odgovarajućem mjestu. Drugi je način koristiti se alatom za pomicanje (prečac: tipka „W“ na tipkovnici).

Za stvaranje dojma da svjetlo dolazi iz lampe u obliku vatre, za nijansu i dinamiku plamena potrebno je prilagoditi nekoliko postavki.



Slika 38. Postavke komponente „Svjetlo“ (slika zaslona, Unity)

Prilagodba boje svjetla: plamen lampe obično ima toplu nijansu, koja se kreće od narančaste do crvenkaste. Da bi se prilagodila boja svjetla, potrebno je pronaći svjetlosnu komponentu unutar inspektorskog prozora te kliknuti na kutiju boje pored *Color*. Zatim treba odabrati toplu narančastu ili crvenkastu boju.

Intenzitet svjetla: plamen lampe titra i mijenja svoj intenzitet. Počinje se s postavljanjem intenziteta svjetla na nižu vrijednost, možda između 1 i 2, ovisno o tome koliku se svjetlinu želi postići.

Domet (eng. *range*): budući da svjetlo lampe obično nema velik domet, možda se želi smanjiti raspon svjetla. Prilagodi se klizač *Range* u svjetlosnoj komponenti tako da svjetlo pokriva samo željeno područje.

4.11.2. Skriptiranje svjetala

Slika 39. Scenski prikaz (slika zaslona, Unity)

Sada kada se scena pokrene, svjetlo iz lampe djeluje, međutim izgleda prilično statično. Plamen lampe obično titra i mijenja svoj intenzitet, stvarajući dinamičan i živahan efekt. Da bi se to postiglo, može se koristiti korutinu u skripti za simulaciju toga efekta.

Kreiranje skripte:

1. korak – u *Unity* uređivaču desnim klikom na projektni prozor treba odabrati ***Create***" > **"C# *Script***

2. korak – potrebno je imenovati novostvorenu skriptu ***FlickerEffect***

3. korak – dvaput kliknuti na skriptu za otvaranje u preferiranom IDE-u (npr. *Visual Studio*).

U nastavku je potrebno napisati sljedeći kod:

```
using System.Collections;
using UnityEngine;
[RequireComponent(typeof(Light))]
public class FlickerEffect : MonoBehaviour
{
    private Light lampLight;
    [SerializeField] private float minIntensity = 1.7f;
    [SerializeField] private float maxIntensity = 2.1f;
    [SerializeField] private float flickerSpeed = 0.08f;
    private void Start()
    {
        lampLight = GetComponent<Light>();
        StartCoroutine(Flicker());
    }
```

```
IEnumerator Flicker()
{
    while (true)
    {
        lampLight.intensity = Random.Range(minIntensity, maxIntensity);
        yield return new WaitForSeconds(flickerSpeed);
    }
}
```

ZADATAK: Dodavanje skripte objektu *LampLight*

Potrebno je dodati skriptu na objekt *Lamplight*!

Međutim, scena može biti još interaktivnija i dojmljivija. Stoga je sljedeći korak u razvojnom putovanju implementacija interaktivnih svjetala. Taj će element omogućiti igračima ili korisnicima izravan utjecaj na svjetlosne izvore unutar virtualnog prostora, dodajući sloj kompleksnosti i interaktivnosti u iskustvo. Bilo da se svjetlo aktivira kada igrač uđe u određeni prostor, ili igrač ima mogućnost upaliti i gasiti svjetla po volji, ti dodatni elementi omogućuju da scena postane interaktivno igralište, a ne samo vizualni prikaz. U narednim koracima razmotrit će se kako implementirati i iskoristiti interaktivne svjetlosne izvore unutar *Unityja*, dajući tako dodatnu dimenziju igri ili aplikaciji koju se izrađuje.

4.11.3. Korutine (eng. Coroutines)

Korutine su posebna vrsta funkcija u *Unityju* koje omogućuju izvođenje niza instrukcija tijekom više okvira, umjesto da se sve izvrši u jednom okviru, kao što je to slučaj s redovitim funkcijama. To je izuzetno korisno kada se želi imati kontrolu nad vremenom izvođenja određenih akcija bez potrebe za kompliciranim upravljanjem vremenom ili bez blokiranja glavne niti. Na primjer, ako se želi stvoriti kašnjenje između dviju akcija u kodu, umjesto korištenja funkcijom *Update* i brojanja vremena, moguće je jednostavno koristiti korutine s *yield return new WaitForSeconds* (sekunde) za stvaranje željenoga kašnjenja.

Osnovna upotreba korutine u *Unityju* izgleda ovako:

```
private IEnumerator MojaCoroutine()
{
        // Kod koji će se izvršiti prije kašnjenja.
        yield return new WaitForSeconds(2); // Čekanje 2 sekunde.
        // Kod koji će se izvršiti nakon kašnjenja.
}
```

Da bi se korutina pokrenula, treba koristiti *StartCoroutine*:

```
StartCoroutine(MojaCoroutine());
```

Neke korisne naredbe koje se često koriste s korutinama uključuju:

yield return null: čeka do sljedećeg okvira
yield return new WaitForSeconds(sekunde): stvara kašnjenje određenog broja sekundi
yield return new WaitForEndOfFrame(): čeka do kraja trenutnog okvira
yield return new WaitForFixedUpdate(): čeka do sljedećeg ažuriranja fiksne frekvencije.

Korutine su moćan alat koji omogućuje kreiranje složenih vremenskih sekvenci i asinkronih operacija bez potrebe za kompliciranim upravljanjem vremenom ili korištenjem dodatnih niti. Osim toga, one čine kod čišćim i lakšim za praćenje u odnosu na tradicionalne metode koje se koriste ažuriranjem i brojanjem vremena.

4.11.4. Interaktivna svjetla

Prije nego se započne s implementacijom interaktivnih svjetala, potrebno je dodati svjetlosne izvore u scenu. Za taj primjer koristit će se dostupna imovina iz *Unityjeve* trgovine, konkretno paket ***Street Lamps 2***, koji se može pronaći na sljedećoj poveznici: https://assetstore.unity.com/packages/3d/props/exterior/street-lamps-2-260395.
Kada je imovina dostupna, može se implementirati jednu od dostupnih lampi u scenu. Započninje se odabirom *prefaba StreetLampRound2B* smještenog u *Assets\SpaceZeta_StreetLamps2\Prefabs*. Njega zatim treba povući na scenu.

Koristite se alatima za transformaciju Da bi ga se adekvatno pozicioniralo i rotiralo unutar dizajna scene, treba se koristiti alatima za transformaciju. Nakon postavljanja svjetiljke u mapi „Assets“, potrebno je kreirati novu skriptu *C#* naziva *LightToggle* Zatim, treba povući skriptu *LightToggle* na objekt lampe u hijerarhijskom prozoru da bi je se povezalo. Unutar skripte napisat će se nekoliko linija koda s ciljem provjere igračevog pritiska tipki, a zatim uključiti/isključiti izvor svjetlosti. Logika koda uključuje osluškivanje pritiska tipke i mijenjanje aktivnog stanja komponente svjetla.

Slijedi prikaz jednostavne implementacije u *C#*-u.

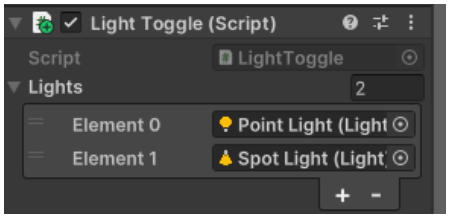
```
using System.Collections.Generic;
using UnityEngine;

public class LightToggle : MonoBehaviour
{
    [SerializeField] private List<Light> lights;  // Lista svjetala.
    private void Update()
    {
        // Provjera pritiska tipke „Enter“.
        if (Input.GetKeyDown(KeyCode.Return))
        {
            // Za svako svjetlo u listi svjetala...
            foreach (Light light in lights)
            {
                // Prebacivanje aktivnog stanja svjetla.
                light.enabled = !light.enabled;
            }
        }
    }
}
```

Natrag u *Unity* uređivaču treba odabrati objekt *StreetLampRound2B* ili odabrani objekt u hijerarhijskom prozoru.

Slika 40. Objekt „Spot Light“ (slika zaslona, Unity)

S odabranim objektom *StreetLampRound2B,* ili izabranim objektom s priloženom skriptom, potrebno je pogledati skriptu *LightToggle* u prozoru *Inspector*.



Slika 41. Komponente „Light Toggle“ (slika zaslona, Unity)

Zatim je potrebno pronaći polje koje predstavlja listu svjetala (obično će biti oznaka *Lights* i broj koji označava koliko je svjetala trenutno dodano u listu). Zadatak je povećati taj broj prema broju svjetala koje se želi kontrolirati, i dodijeliti svjetla u svako novostvoreno polje. Drugi je način direktno povući svjetla iz hijerarhijskog prozora u polje liste svjetala u prozoru *Inspector*.

Nakon toga je potrebno pritisnuti ***Play*** te pokušati pritiskom na ***Enter*** promijeniti stanje svih svjetala koja su ranije dodana u listu istovremeno. Svjetla bi se trebala uključivati i isključivati sinkronizirano sa svakim pritiskom tipke. Ta prilagodba omogućava da se jednim unosom mijenja stanje više objekata odjednom, dodajući složenost i interaktivnost u scenu u *Unityju*. To se može koristiti kao osnova za različite vrste igračkih interakcija i svjetlosnih efekata unutar projekta.

4.12. Input

U *Unityju* sustav za unos (eng. *Input*) omogućava programerima primanje ulaznih podataka od korisnika, bilo da se radi o tipkovničkim naredbama, pokretima miša, dodirima na zaslonu ili sustavima za unos s igraćih kontrolora. Sustav za unos ključan je za interakciju s igračima i upravljanje njihovim iskustvima unutar igre. *Unityjeva* klasa *Input* omogućava pristup trenutnom stanju svih podržanih ulaznih uređaja.

Tipkovnički unos: da bi se provjerilo je li određena tipka pritisnuta, može se koristiti metodom *Input.GetKey*.

```
if (Input.GetKey(KeyCode.W))
{
        // Kretanje naprijed.
}
```

Postoje i varijacije te metode, poput *Input.GetKeyDown* i *Input.GetKeyUp* koje detektiraju trenutak pritiska ili otpuštanja tipke.

Unos miša: mišem se često koristi za interakciju, bilo klikovima ili pokazivačem.

```
if (Input.GetMouseButton(0)) // Provjerava je li lijevi gumb miša pritisnut.
{
        // Izvršava određenu akciju.
```

```
}
Vector3 pozicijaMis = Input.mousePosition; // Dohvaća trenutnu poziciju kursora miša na
zaslonu.
```

Za uređaje poput džojstika, može se koristiti *Input.GetAxis* za detekciju analognoga unosa. Primjerice:

```
// Vraća vrijednost između -1 i 1, ovisno o položaju analognog kontrolora ili tipki tipkovnice.
float horizontalno = Input.GetAxis("Horizontal");
```

4.13. Dovršavanje scene

Istraživanje i kreiranje u *Unityju* nudi obilje mogućnosti za raznovrsnost i unikatan vizualni izraz unutar digitalnog svijeta. U nastavku su navedeni i opisani pojedini primjeri imovine koji se mogu iskoristiti:

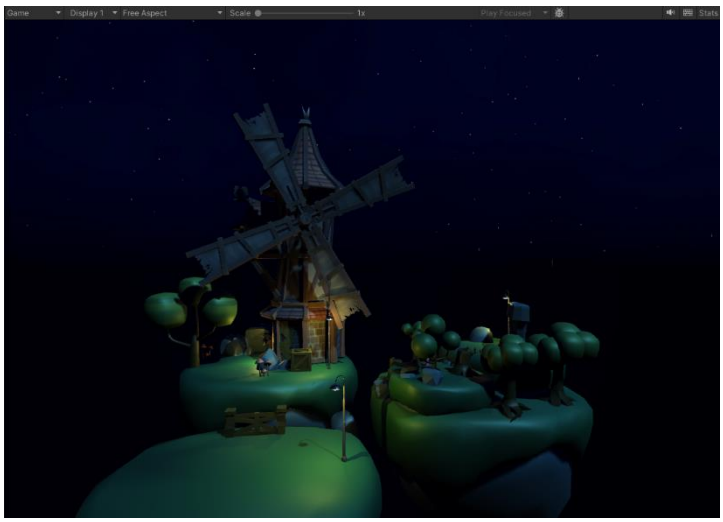
- ***Woodland Lowpoly Environment*:** ta imovina nudi bogatstvo niskopoligonalnih prirodnih elemenata koji mogu obogatiti scenu različitim prirodnim ljepotama. Poveznica na paket: Woodland Lowpoly Environment.
- ***Medieval Windmill*:** kao prethodno korištena imovina, ta vjetrenjača može biti osnova ili dopuna u kreiranju atmosfere i konteksta scene. Poveznica na paket: Medieval Windmill.
- ***Street Lamps 2*:** ovaj paket vanjskih lampi nudi širok asortiman uličnih svjetiljki koje mogu obogatiti vizualni dojam vanjskih scena ili urbanih okruženja. Poveznica na paket: Street Lamps 2.

ZADATAK:

Nakon što su sve imovine preuzete i uvezene u *Unity* projekt, može se eksperimentirati miješanjem i kombiniranjem različitih elemenata da bi se stvorila scena koja odražava osoban i jedinstven kreativni izraz. Potrebno je proučiti kako se različiti elementi – kao što su svjetlo, teksture, i boje – mogu kombinirati za željeni vizualni i emotivni dojam.

Eksperimentiranjem i istraživanjem svaka scena koju se kreira, ne samo da je prilika za razvoj vještina u oblikovanju 3D svijeta i interaktivnih elemenata, već je i prilika za otkrivanje novih stilova i koncepata.

Pogled na scenu



Slika 42. Završna scena (slika zaslona Unity)

Scena je osmišljena s ciljem kombiniranja različitih elemenata dostupnih posredstvom različitih imovina koje su prethodno istražene i preuzete. Njihovim korištenjem kreiran je jedinstven, srednjovjekovni ambijent prožet elementima prirode i rustikalnim stilom. Može se vidjeti kako vjetrenjača (*Medieval Windmill*) stoji kao centralna točka interesa, okružena bogatstvom prirodnih elemenata iz seta *Woodland Lowpoly Environment*, uključujući drveće, kamenje i različite biljke, stvarajući uravnoteženu i ugodnu vizualnu scenu.

Ulične lampe iz seta *Street Lamp Physics* dodane su radi stvaranja dodatnih točaka interesa i osvjetljenja, posebno u noćnim ili sumračnim verzijama scene. U procesu kreiranja sceneu treba razmotriti kako svaki element koji dodajemo može utjecati na cjelokupni vizualni i emotivni dojam. Svaka komponenta, od osvjetljenja do postavljanja objekata, igra ključnu ulogu u oblikovanju percepcije i iskustva korisnika.

4.14. Animacija

Animacije su u *Unityju* središnji dio većine projekata, bilo da se radi o kreiranju fluidnih pokreta likova, animiranju UI elemenata ili postizanju dinamičnih efekata u svijetu igre. Animacija u *Unityju* predstavlja tehniku koja omogućuje kreiranje pokreta i promjena u izgledu objekata ili karaktera unutar igre ili aplikacije. To se postiže nizom postupaka koji mijenjaju poziciju, rotaciju, veličinu, boje i druga svojstva objekata u vremenskom periodu. *Unity* nudi opsežan set alata za kreiranje i upravljanje animacijama.

U nastavku će biti izdvojene osnovne animacije u *Unityju*.

- **Kontrolor animatora (eng. *Animator Controller*):** to je srce sustava animacije u *Unityju*. Omogućuje kreiranje i upravljanje tijekom animacija za likove ili druge objekte. U njemu je moguće definirati različita stanja animacije i prijelaze između njih.
- **Isječak animacije *(eng. Animation Clip*):** pojedinačna je animacija, snimljena za određeni objekt. Može sadržavati informacije o pokretu, rotaciji, skaliranju i drugim svojstvima objekta tijekom određenoga vremenskog razdoblja.
- **Prozor animacije *(eng. Animation Window):*** taj je prozor središnji alat za kreiranje i uređivanje isječaka animacije. Omogućuje ručno postavljanje ključnih točaka (eng. *keyframes*) za različite atribute objekta, kreirajući tako animaciju.
- **Render mreže s kožom *(eng. Skinned Mesh Renderer):*** za 3D likove koji se trebaju animirati s kosturom (*skeleton*), *Skinned Mesh Renderer* komponenta je koja omogućuje deformaciju mreže u skladu s animacijom kostura.
- **Stabla miješanja *(eng. Blend Trees):*** unutar kontrolora animatora stabla miješanja omogućuju fluidne prijelaze između različitih animacija na temelju ulaznih parametara, poput brzine ili smjera. To je posebno korisno za likove koji imaju različite animacije kretanja.
- **Komponenta animatora *(eng. Animator Component):*** dodana na *GameObject*, komponenta animatora povezuje objekt s kontrolorom animatora i omogućuje reprodukciju animacija u radnom vremenu.
- **Avatar *(eng. Avatar):*** to je reprezentacija likova kostura (*skeleton*) u *Unityju*. Omogućuje mapiranje animacija na likove, čak i ako su animacije izvorno kreirane za drugačiji model.
- **Humanoidni i generički kosturi *(eng. Humanoid & Generic Rigs):*** *Unity* podržava različite vrste kostura, od kojih su najistaknutiji humanoidni (ljudski oblik) i generički (bilo koji oblik). To je korisno kada se želi koristiti iste animacije na različitim modelima.
- **Kontrole za reprodukciju *(eng. Playback Controls):*** kad se radi s animacijama u *Unityju*, može se koristiti kontrolama za reprodukciju kako bi se pregledale i testirale animacije u realnom vremenu.
- **Događaji animacije *(eng. Animation Events):*** omogućuju poziv određenih metoda ili funkcija u točno određenim točkama unutar animacije.

Sustav animacije u *Unityju* pruža iznimnu fleksibilnost i kontrolu, omogućujući programerima i umjetnicima prenijeti svoje vizije u život. Bilo da se animiraju složeni likovi, jednostavni objekti ili dinamički UI elementi, alati koji su na raspolaganju osigurat će da kreacije izgledaju fluidno i profesionalno.

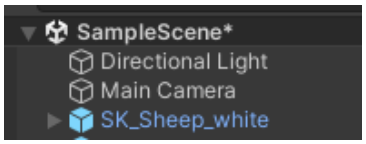
4.14.1. Oživljavanje 3D scene sa animacijom

S ciljem finalizacije scene i dodavanja još slojeva kompleksnosti i uživljenosti, implementirat će se animacije koristeći se s nekoliko životinjskih modela dostupnih u *Unityjevoj* trgovini imovinom. Korištenjem imovine *Animated Goat and Sheep 3D Low Poly Free*, dodat će se kretanje i dinamika sceni. U nastavku će biti opisani koraci.

Slika 43. Scenski prikaz (slika zaslona, Unity)

- Potrebno je posjetiti navedenu poveznicu i odabrati opciju ***Download*** ako imovina nije već dodana u *Unity* biblioteku, ili ***Open in Unity*** da bi se direktno pristupilo imovini u *Unity* sučelju.
- Nakon uvoza imovine, treba potražiti objekt imenovan ***SK_Sheep_white***.
- Zatim je potrebno povući i ispustiti taj *prefab* objekt direktno na scenu ili u hijerarhijski prozor, pozicionirajući i prilagođavajući ga prema željenom estetskom i funkcionalnom kontekstu scene.

Slika 45. Projektni prikaz „SK_Sheep_white objekta“ (slika zaslona, Unity)



Slika 44. „SK_Sheep_white u hierarhiji“ (slika zaslona, Unity)

Kontrola animacija

Da bi se kreirao vlastiti animacijski kontrolor, potrebno je učiniti sljedeće:

Za početak je potrebno kreirati novi animacijski kontrolor desnim klikom u projektnom prozoru te odabirom *Create*" > "*Animator Controller*. Dvostruko kliknuti na kontrolor za otvaranje prozora *Animator*; potrebno je odabrati objekt *SK_Sheep_white* u hijerarhijskom prozoru. U prozoru *Inspector* treba pronaći

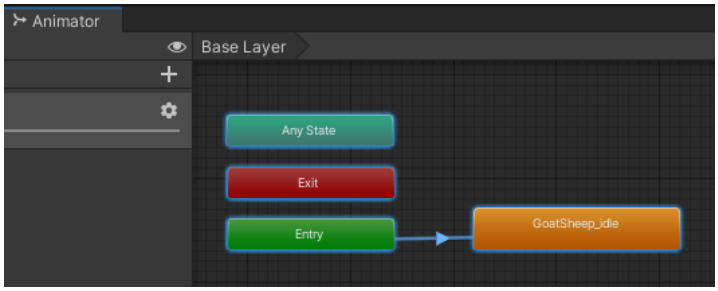
komponentu *Animator*. U polju *Controller* komponente *Animator* zatim treba povući i ispustiti novokreirani animacijski kontrolor. Postojeća animacijska skripta se uklanja s ovce klikom na *Remove Component* u opcijama skripte *Animation Controller*. Kreirana ovca sada koristi novi animacijski kontrolor za sve svoje animacije, omogućujući kontrolu nad animacijama unutar *Unityja*.

4.14.2. Dodavanje animacija

Za vizualnu kontrolu i dodavanje animacije za karakter u komponenti ***Animator***, važno je koristiti prozor ***Animation***. U nastavku će biti objašnjeno kako se to može učiniti s pomoću animacije ***GoatSheep_idle***, te kako se to odnosi na vizualno upravljanje stanjima.

Za dodavanje animacije potrebno je dvaput kliknuti na novostvoreni animator.
Zatim, unutar prozora ***Animator*** treba potražiti sučelje koje omogućuje povlaćenje i ispuštanje (eng. *drag and drop*) animacije. Nakon toga je potrebno otvoriti prozor ***Project*** i pronaći animaciju ***GoatSheep_idle***. Može se pronaći na putu ***Assets\UrsaAnimation\LOW POLY CUBIC – Goat and Sheep Pack\Animation\Goat&Sheep***.

Naposlijetku je potrebno jednostavno povući i ispustiti animaciju ***GoatSheep_idle*** u prozor ***Animation***.



Slika 46. Animacijski kontrolor stanja (slika zaslona, Unity)

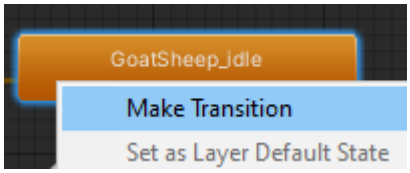
Vizualni kontrolor stanja, ili ***Animator*** omogućuje ne samo pregled svih dostupnih animacija za određeni objekt, već i kreiranje animacijske sekvence i uvjeta za prijelaz između različitih animacijskih stanja. Osim toga, pruža snažne alate za kontrolu i fino podešavanje animacijskog ponašanja objekata i likova u *Unityju*.

4.14.3. Skriptiranje animacija

Animacijsko putovanje nastavlja se dodavanjem novih animacija unutar *Animator Controllera*. Usredotočit će se na dvije nove animacije: ***GoatSheep_stand_to_sit*** i ***GoatSheep_sit_to_stand***. Te animacije, kako im i imena sugeriraju, omogućavaju kreiranoj ovci prijelaz iz stojećega u sjedeći položaj, i obrnuto.

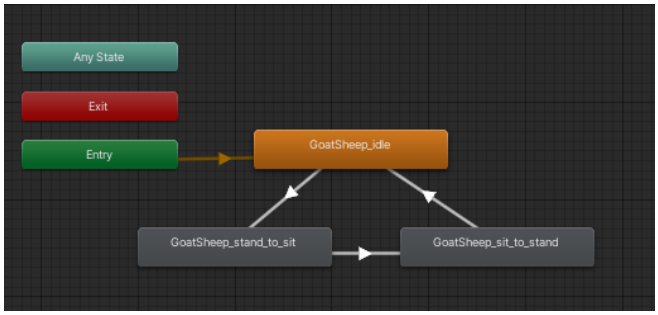
Da bi se dodala animacija u *Animator,* potrebno je slijediti ove korake:

1. unutar prozora *Animator*, uz već postojeću animaciju ***GoatSheep_idle***, treba povući i ispustiti animacije ***GoatSheep_stand_to_sit*** i ***GoatSheep_sit_to_stand*** iz prozora *Project*. Nove animacije trebaju biti povezane s postojećom animacijom *idle* da bi *Animator* znao kada prelaziti iz jedne animacije u drugu



Slika 47. Stvaranje tranzicije (slika zaslona, Unity)

2. da bi se to učinilo, potrebno je kliknuti desnom tipkom miša na animaciju ***GoatSheep_idle*** u prozoru *Animator* i stvoriti prijelaz (eng. ***Make Transition***) koji vodi do animacije ***GoatSheep_stand_to_sit***



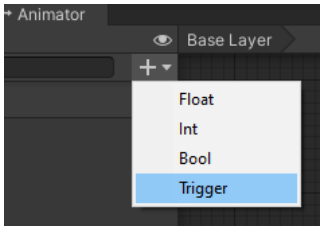
Slika 48. Postavke animacija u „Animatoru“ (slika zaslona, Unity)

3. da bi se stvorio prijelaz iz ***GoatSheep_stand_to_sit*** do ***GoatSheep_sit_to_stand*** i zatim natrag do ***GoatSheep_idle***, potrebno je ponoviti isti proces.
4. sada, kada se pritisne ***Play*** u *Unity* uređivaču, može se primjetiti da ovca automatski prolazi svim animacijama u petlji, od stojećeg položaja, preko sjedenja, pa do ponovnoga stojećeg položaja.

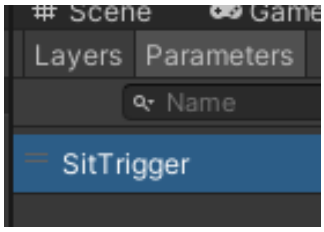
Da bi se to postiglo, koristit će se s *triggers* unutar *Animatora*, koji omogućavaju pokretanje određenih animacija kad god se želi, umjesto da se automatski pokreću u petlji.

Nakon toga potrebno je otvoriti *Animator Controller* te se vratiti u prozor *Animator* i odabrati *Animator Controller* kreirane ovce.

Dodavanje parametra:

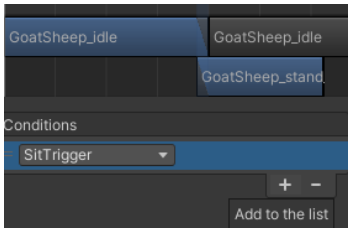


Slika 50. Parametri animatora (slika zaslona, Unity)



Slika 49. Parametar „Sit Trigger“ (slika zaslona, Unity)

Potrebno je kliknuti na karticu ***Parameters*** unutar prozora *Animator* i dodati novi ***Trigger*** parametar. Može ga se nazvati, na primjer, ***SitTrigger***.



Slika 51. Postavljanje okidača (slika zaslona, Unity)

Postavljanje uvjeta tranzicije:

Potrebno je kliknuti na strelicu kreirane tranzicije između ***GoatSheep_idle*** i ***GoatSheep_stand_to_sit*****.** Na kartici „Uvjeti“ u prozoru „Inspector“, treba kliknut na „+“ da bi se dodao uvjet.

Koristeći se padajućim izbornikom, treba odabrati ***SitTrigger*** kao okidač.

Da bi se kreiranoj animiranoj ovci osigurao glatki prijelaz između različitih stanja, uvest će se novi okidač koji će omogućiti tranziciju iz sjedećega u stojeći položaj. Potrebno je slijediti dolje navedene korake, preslikavajući ono ranije učinjeno sa ***SitTrigger***, da bi se stvorio ***StandTrigger*** i pravilno ga se primijenilo unutar *Animatora*.

Dok je odabran *Animator Controller* za ovce, potrebno je doći do prozora *Animator*.

U prozoru *Animator*, na dnu, može se pronaći kartica *Parameters* („parametri“). Potrebno je kliknuti na nju da bi se proširio odjeljak parametara. Zatim treba kliknuti na „+“ ikonu da bi se dodao novi parametar te odabrati *Trigger* iz padajućeg izbornika. Novi okidač treba imenovati kao *StandTrigger*. U nastavku, u prozoru *Animator* treba pronaći i kliknuti na prijelaznu strelicu koja ide iz stanja *GoatSheep_stand_to_sit* do *GoatSheep_sit_to_stand*. U desnom oknu pojavit će se postavke prijelaza. Tu se mogu vidjeti različite opcije za uređivanje svojstava prijelaza. Potrebno je kliknuti na „+“ ikonu u sekciji *Conditions* (hrv. uvjeti) da bi se dodali novi uvjeti. Iz padajućeg izbornika treba odabrati *StandTrigger* kao uvjet za prijelaz.

Sada kad se *StandTrigger* aktivira pomoću skripte, ovca bi trebala napraviti tranziciju iz animacije *GoatSheep_stand_to_sit* u animaciju *GoatSheep_sit_to_stand*. Na taj način se kontroliraju animacijski prijelazi kreiranog lika s većom preciznošću te se omogućuju kompleksnije sekvence animacija. Ti okidači mogu se aktivirati unutar skripte da bi dinamički kontrolirali animacijsko stanje ovce tijekom izvođenja igre. U

sljedećem koraku kreira se skripta koja će se zvati *GameManager*, i koja će upravljati animacijama svih ovaca u sceni u odnosu na status svjetla. Ideja je imati listu svih animatora ovaca i funkciju koja provjerava je li svjetlo uključeno ili isključeno. Ako je svjetlo isključeno, sve ovce će sjesti, a ako je svjetlo uključeno, sve ovce će ustati.

U *Unityju* u *Assets* direktoriju treba kliknuti desnom tipkom miša i odabrati *Create* > C# *Script*.

Skriptu treba imenovati kao *GameManager* i otvoriti je u IDE-u za uređivanje koda.

Kreiranje „Liste Animatora“

U skripti *GameManager* treba definirati listu objekata “Animator” koja će sadržavati reference na sve animatore ovaca u sceni te kreirati funkciju koja provjerava status svjetla (uključeno/isključeno) i, ovisno o statusu, aktivira odgovarajući animacijski okidač (*SitTrigger* ili *StandTrigger*) za sve ovce.

```
using System.Collections.Generic;
using UnityEngine;
public class GameManager : MonoBehaviour
{
   [SerializeField] private List<Animator> sheepAnimators = new List<Animator>();

   public void ControlSheepAnimations(bool isLightOn)
   {
        string triggerToActivate = isLightOn ? "StandTrigger" : "SitTrigger";

        foreach (Animator sheepAnimator in sheepAnimators)
        {
                        sheepAnimator.SetTrigger(triggerToActivate);
        }
   }
}
```

Da bi se uključio *GameManager* u kreiranoj *Unity* sceni, potrebno je slijediti nekoliko jednostavnih koraka.

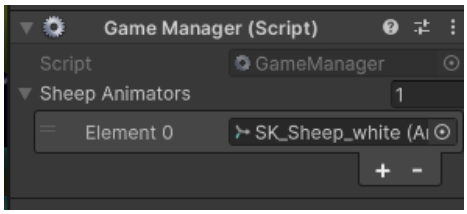
1. Prvo, u sceni treba kreirati prazan *GameObject*. To je moguće učiniti na dva načina: na vrhu zaslona u *Unity* uređivaču treba kliknuti na *GameObject*, a zatim odabrati *Create Empty* iz padajućeg izbornika. Drugi je način desnim klikom u

hijerarhijskom prozoru odabrati opcije *Create Empty*. Taj novostvoreni objekt bit će centralni upravljački objekt za različite aspekte igre i animacija unutar scene.

2. Nakon toga, ključno je pravilno imenovati *GameObject* kako bi se održavala organizacija i jasnoća unutar projekta. U hijerarhijskom prozoru treba pronaći novostvoreni prazni *GameObject*, koji će obično biti automatski nazvan kao *GameObject* ili *Empty*. Zatim ga je potrebno preimovati u *GameManager* da bi njegova uloga i funkcija bila jasna svima koji pregledavaju projekt. Preimenovanje se može obaviti desnim klikom i odabirom opcije *Rename* ili jednostavno pritiskom na *Enter* kada je objekt odabran, nakon čega se može upisati željeno ime.
3. Konačno, potrebno je dodati skriptu kreiranom objektu *GameManager*. S još uvijek odabranim objektom, treba preći na prozor *Inspector* s desne strane. Zatim kliknuti na gumb ***Add Component***. U polje za pretragu koje se pojavi, treba upisati *GameManager* ili ime koje je pripisano kreiranoj skripti te dodati je na nju klikom. Ako je skripta već napisana i dodana u projekt, ona će se pojaviti među dostupnim komponentama.

Da bi se objekt ***SK_Sheep_white*** povezao s ***GameManager***, potrebno se pridržavati sljedećih koraka.

Lociranje ovce: potrebno je pronaći objekt *SK_Sheep_white* u hijerarhijskom prozoru.



Slika 52. Postavke komponente „Game Manager“ (slika zaslona, Unity)

Metoda **povuci i ispusti**: treba kliknuti i držati objekt *SK_Sheep_white*, a zatim ga povući u jedno od dostupnih polja unutar liste *sheepAnimators* ili niza u komponenti *GameManager* u prozoru *Inspector*.

Taj postupak omogućuje skripti *GameManager* da pristupi i upravlja komponentom ovce *Animator*. U slučaju da je potrebno kontrolirati više ovaca ili drugih objekata, jednostavno se može proširiti listu ili niz unutar skripte *GameManager* i ponoviti postupak povlačenja i ispuštanja za svaki objekt. Prije nego što se započne s prilagođavanjem skripte *LightToggle* za integraciju kontrola animacija ovaca, postojat će kratki pregled o konceptu *Tagova* u *Unityju* jer će se njima koristiti u poboljšanom skriptiranju.

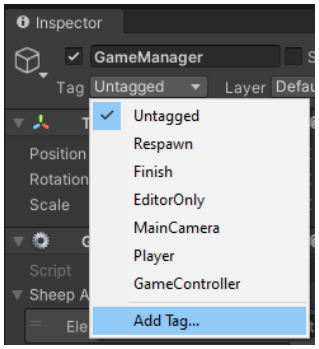
4.15. Oznake (eng. tags)

U *Unityju* **oznake** (eng. *tags*) pružaju način identifikacije određenih objekata unutar projekta. One su poput etiketa koje je moguće dodijeliti objektima da bismo ih lako identificirali ili grupirali prema određenim kriterijima. Dok se složeniji sustavi identifikacije često oslanjaju na **slojeve** (eng. *layers*) ili prilagođene skripte, oznake pružaju jednostavan i brz način za kategorizaciju i referencu objekata.

Slijedi nekoliko ključnih točaka o oznakama u *Unityju*.

- **Jednostavna identifikacija:** moguće je dodijeliti oznaku objektu i zatim koristiti skriptu za provjeru ima li određeni objekt tu oznaku. Na primjer, u slučaju postojanja više neprijatelja u igri, svima je moguće dodijeliti oznaku „Neprijatelj“, i zatim koristiti tu oznaku kako bi se provjerilo je li igračev metak pogodio neprijatelja.
- **Unaprijed definirane oznake:** *Unity* dolazi s nekoliko unaprijed definiranih oznaka, kao što su *Player*, *MainCamera* i *Finish*. To može olakšati postavljanje osnovnih funkcionalnosti, poput identificiranja igračeve figure ili glavne kamere.
- **Dodavanje prilagođenih oznaka**: moguće je dodavati i vlastite oznake prema potrebi projekta. Da bi se to učinilo, potrebno je odabrati objekt, zatim otići na padajući izbornik *Tag* u prozoru *Inspector*, odabrati *Add Tag* i dodato nove oznake na popis.
- **Skriptiranje s oznakama:** oznake se često koriste u *C#* skriptama za provjeru ili promjenu ponašanja objekata. Na primjer, može se koristiti metodu ***CompareTag*** (*string tag*) da bi se provjerilo ima li objekt određenu oznaku.

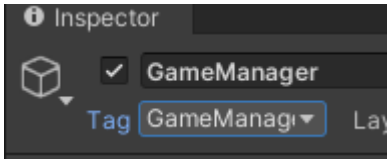
4.15.1 Stvaranje nove oznake



Slika 53. Dodavanje oznake (slika zaslona Unity)

1. Potrebno je u *Unity* uređivaču odabrati bilo koji *GameObject* u kreiranoj sceni (ne mora biti objekt koji se namjerava označiti). To se može učiniti klikom na njega u prikazu scene ili odabirom iz hijerarhijskoga prozora (*Hierarchy*).
2. Na vrhu prozora *Inspector* može se vidjeti padajući izbornik *Tag*. Potrebno je kliknuti na nj i odabrati *Add Tag*… („Dodati oznaku“).

3. Zatim treba kliknuti na malu ikonu „+“ (plus) da bi se stvorila nova oznaka. Nakon toga, treba imenovati novu oznaku *GameManager* i pritisnuti *Enter* ili kliknuti negdje drugdje da bi se spremilo ime.



Slika 54. Postavljanje oznake / Screen Shot iz Unity-a

4. Nakon što je uspješno stvorena oznaka *GameManager* u *Unity* sučelju, vrijeme je da je se primijeni na odgovarajući *GameObject,* u ovom kontekstu – na objekt naziva *GameManager*.
5. U ovom koraku potrebno je pristupiti prozoru *Inspector* i otvoriti padajući izbornik *Tag* je smješten na vrhu prozora.

6. U tom izborniku sada bi se trebala uočiti novoformirana oznaka *GameManager*. Odabirom te oznake, dodijelit će je se svojem objektu, omogućavajući time lakšu identifikaciju i interakciju s njime u skriptama i različitim funkcionalnostima u *Unity* uređivaču. To je posebno korisno za organizaciju i efikasno upravljanje objektima unutar scene ili u logici igre koju se implementira.

4.16. Pokretanje animacija

Ukoliko se skriptu *LightToggle* želi prilagoditi, to se čini pronalaskom skripte *GameManager* i pozivom određene funkcije koja će kontrolirati animacije ovaca. Ta bi funkcija trebala promijeniti stanje animacija na temelju statusa svjetla.

Za realizaciju toga, ažurirana skripta *LightToggle* trebala bi izgledati ovako:

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

public class LightToggle : MonoBehaviour
{
    [SerializeField] private List<Light> lights;  // Lista svjetala.
    private GameManager gameManager; // Referenca na GameManager.

    private void Awake()
    {
        var gmObj = GameObject.FindGameObjectWithTag("GameManager");
```

```
            gameManager = gmObj.GetComponent<GameManager>();
    }
private void Update()
{
        // Provjera pritiska tipke Enter.
        if (Input.GetKeyDown(KeyCode.Return))
        {
                var enabled =  false;
                // Za svako svjetlo u listi svjetala.
                foreach (Light light in lights)
                {
                                // Prebacivanje aktivnog stanja svjetla.
                        light.enabled = !light.enabled;
                        enabled = light.enabled;
                }

            // Pozivanje funkcije unutar skripte GameManager, koja kontrolira animacije
 ovaca.
            // Prosljeđivanje trenutnog stanja svjetla kao argument.

              gameManager.ControlSheepAnimations(enabled);
        }
    }
}
```

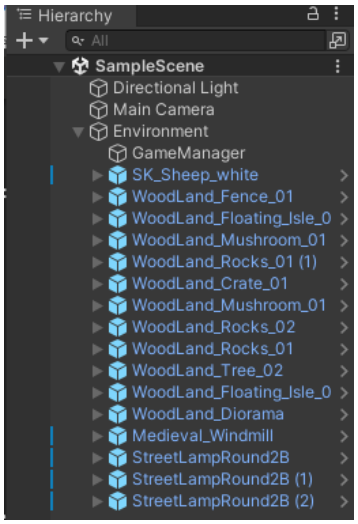
Pritisak na tipku *Enter* aktivira svjetla i pokreće animacije ovce, mijenjajući njezine pozicije i stanja u skladu sa statusom svjetla, pružajući osnovnu interaktivnost u *Unity* sceni. Interakcija, koja uključuje korištenje animacijskih okidača i kontrolu svjetla u skripti, može se nadograditi za kompleksnije animacije i ponašanja u budućnosti. Drugi je način kontrole animacije koristeći se *float* varijablom u *Animator Controlleru* za upravljanje prijelazima između različitih animacijskih stanja, poput hodanja i trčanja, na osnovu brzine lika, omogućujući tako fluidne i realistične animacijske prijelaze i dodatnu kontrolu nad dinamikom pokreta.

4.17. Unity paketi

Scenu koja je izrađena tijekom implementacije svih prethodno obrađenih koncepata i postupaka, može se koristiti kao osnovu za **AR** (proširena stvarnost) i **VR** (virtualna stvarnost) dijelove priručnika. To znači da će se okruženje stvoreno kao *Unity* paket morati izvesti, a to će biti učinjeno u nastavku.

Izvoz scene i svih povezanih resursa u *Unity* paket omogućava jednostavan prijenos ili podjelu vlastitog rada s drugima ili ponovno korištenje njime u drugim projektima, čuvajući sve skripte, teksture, modele, scene i druge resurse u jednoj kompaktnoj datoteci. U narednim koracima detaljno će se proći kroz postupak izvoza projekta kao *Unity* paketa kako bi se osiguralo da sve razvijeno bude sačuvano i spremno za daljnju upotrebu u AR i VR razvojnim fazama priručnika.

U *Unity* uređivaču navigacija na ***GameObject*** u gornjoj traci izbornika i odabir ***Create Empty*** omogućit će kreiranje novog praznog objekta. Drugi je način desnim klikom u „Hijerarhiji“ i odabirom *Create Empty* postići isto. Novostvoreni *GameObject*, koji će inicijalno možda biti nazvan *GameObject* ili *Empty*, treba biti preimenovan u ***Environment***. Preimenuje se desnim klikom na objekt i odabirom *Rename*.



Slika 55. Svi objekti u hierarhiji scene / Screen Shot iz Unity-a

Nakon toga predstoji zadatak premještanja objekata unutar scene. Sve objekte koji su prethodno kreirani, i koji čine scenu s izuzetkom direktnog svjetla i glavne kamere, trebalo bi premjestiti ispod objekta ***Environment***. Da bi se to učinilo, jednostavno treba povući i ispustiti svaki objekt na ***Environment*** u hierarhijskom prozoru. Sve te komponente sada postaju dječji objekti novostvorenog objekta ***Environment***, održavajući na taj način čistoću i organiziranost hijerarhijske strukture.

Završni je korak u ovom procesu stvaranje ***prefaba***. Najprije u prozoru *Project* treba kreirati novu mapu nazvanu ***Prefabs***. Ta mapa služi za smještaj *prefab* objekata, osiguravajući urednost i jednostavnost navigacije. Nakon što se odabere *Environment GameObject* u hijerarhiji, treba ga povući i ispustiti unutar nove mape ***Prefabs*** u prozoru *Project*. Na taj način objekt *Environment* postaje *prefab*.

Izvoz *Unity* paketa

Potrebno je doći do *prefaba* ***Environment*** u prozoru ***Project*** u *Unity* uređivaču. Taj bi *prefab* trebao sadržavati sve objekte i postavke koje su prethodno konfigurirane i želi ih se izvesti kao *Unity* paket. Kada se pronađe *prefab* ***Environment***, desnim klikom miša treba kliknuti na njega za otvaranje kontekstnoga izbornika. U tom izborniku treba potražiti i odabrati opciju ***Export Package***. Važno je napomenuti da ta opcija možda neće biti dostupna u svim verzijama *Unityja*, i ovisi o određenim postavkama i dodacima.

Nakon što se odabere ***Export Package***, pojavit će se prozor u kojem se može vidjeti i uređivati koje će sve resurse paket uključiti. Potrebno je proći listom i osigurati da su svi potrebni resursi odabrani, a nepotrebni isključeni.

Klikom na ***Export*** pita se kamo spremiti *Unity* paket. Naposlijetku treba odabrati lokaciju na računalu, unijeti ime za datoteku, i paziti na korištenje *.unitypackage* ekstenzijom datoteke. Da bi se završio proces izvoza, potrebno je kliknuti *Save*.

PITANJA ZA PONAVLJANJE

Provjerite svoje znanje odgovorima na sljedeća pitanja:

1. Na koji se način može kreirati novi 3D objekt u *Unity* okolini?
2. Što je *prefab* u *Unityju,* i kako se njime koristi prilikom razvoja igre?
3. Koja je osnovna svrha korištenja *Animator Controllera* u *Unity* projektu?
4. Koje su osnovne komponente potrebne za izradu materijala koji simulira mokru, sjajnu površinu stijene u *Unityju*?
5. Objasnite odnos dijete – roditelj (eng. *Child-Parent*) u *Unity* hijerarhijskom prozoru.
6. Na koji se način može kreirati i primijeniti *C#* skriptu na objekt u *Unityju*?
7. Kako koristiti korutine u *Unityju* i zašto su one korisne u upravljanju asinkronim operacijama?
8. Kako se može implementirati funkcionalnost u *Unityju* koja omogućava igraču interagiranje s objektima u sceni (npr. uključivanje svjetla)?

9. Kako treba postaviti kameru u *Unity* scenu da je usredotočena na određeni objekt ili poziciju?
10. Što je *skybox* i kako se implementira?

5. ALTERNATIVNA OKRUŽENJA

U OVOM POGLAVLJU NAUČIT ĆETE:

- postaviti razvojnu okolinu u Unityju za alternativnu stvarnost
- razviti sadržaj prikladan za AR, VR i MR okoline koristeći Unity
- koristiti se komponentama za razvijanje mobilnih aplikacija
- razviti vještine stvaranja interaktivnih korisničkih iskustava u AR-u, VR-u i MR-u, uključujući UI i UX razmatranja.

U posljednjem desetljeću razvoj tehnologije doveo je do revolucionarnih promjena u načinu doživljavanja i integriranja s digitalnim svijetom. Tri se ključne tehnologije ističu kao predvodnice toga digitalnog preporoda: **proširena stvarnost (AR)**, **virtualna stvarnost (VR)** i **mješovita stvarnost (MR)**. Iako se pojmovima ponekad koristi naizmjenično, svaka od tih tehnologija pruža jedinstven način interakcije i percepcije digitalnih informacija. U ovom poglavlju razmotrit će se što svaka od tih tehnologija predstavlja, njihove ključne razlike i primjere hardvera koji ih podržavaju.

Proširena stvarnost (eng. *Augmented Reality*, AR)

Proširena stvarnost tehnološki je koncept koji korisnicima omogućava nadopunu stvarnog svijeta digitalnim informacijama. To se postiže tako da se u stvarni svijet postavljaju digitalni objekti poput slika, zvuka ili drugih informacija. Jedan od najpoznatijih primjera uređaja koji koristi proširenu stvarnost upravo su mobilni uređaji, poput pametnih telefona i tableta, koji su se dokazali kao efikasne platforme za proširenu stvarnost u aplikacijama poput Pokémon Go i IKEA Place.

Proširena stvarnost omogućuje digitalno obogaćivanje stvarnog okruženja. Kombinacijom stvarnoga i virtualnog, AR nudi korisnicima dodatne informacije i

interaktivnost u realnom vremenu. Proširena stvarnost svoju primjenu pronalazi, između ostalog, u sljedećim područjima:

- **navigacija i putovanje**: mobilne se aplikacije koriste AR-om za prikaz informacija o smjerovima ili značajkama oko korisnika. Na primjer, aplikacija može prikazivati strelice na cesti da bi korisniku pokazala gdje treba skrenuti

- **trgovina i oglašavanje**: trgovine se mogu koristiti AR-om da bi omogućile kupcima „isprobati“ proizvode virtualno prije kupnje, poput isprobavanja odjeće ili obuće

- **obrazovanje**: učenici se mogu koristiti AR-om za vizualizaciju složenih koncepta, poput anatomije ljudskog tijela ili 3D struktura molekula.

Virtualna stvarnost (eng. Virtual Reality, VR)

Za razliku od proširene stvarnosti, koja samo dodaje digitalne slojeve stvarnom svijetu, virtualna stvarnost stvara potpuno novi, digitalni svijet u kojem korisnici mogu biti potpuno uronjeni. U virtualnoj stvarnosti korisnik je izoliran od stvarnog svijeta i umjesto toga vidi i interagira isključivo s digitalnim okruženjem. Da bi se postiglo takvo iskustvo, potrebna je specifična oprema. Primjerice, **VR naočale** kao što su *Oculus Rift*, *HTC Vive* i *Sony PlayStation* VR su uređaji koji omogućavaju tu vrstu iskustva. Osim toga, postoje specijalizirani **kontrolori** koji korisnicima omogućavaju razne načine interagiranja s virtualnim svijetom. VR-om je moguće doživjeti situacije, mjesta i aktivnosti koje inače ne bi bile dostupne ili moguće u stvarnom životu. Virtualna se stvarnost može primijeniti, između ostalog, u sljedećem:

- **videoigre:** igrači mogu biti potpuno uživljeni u igru, osjećajući kao da su stvarno unutar nje interagirajući s virtualnim okruženjem

- **obuka i simulacije**: profesionalci poput pilota, kirurga ili vatrogasaca mogu se koristiti VR-om za simulaciju stvarnih scenarija i vježbanje svojih vještina u sigurnom okruženju

- **terapija i rehabilitacija**: VR može biti korišten za liječenje fobija ili PTSP-a tako da pacijenti proživljavaju kontrolirane scenarije koji im pomažu suočiti se sa svojim strahovima.

Mješovita stvarnost (eng. Mixed Reality, MR)

Mješovita stvarnost hibridna je tehnologija koja kombinira elemente i proširene, i virtualne stvarnosti. U okruženju mješovite stvarnosti, digitalni i stvarni svijet mogu koegzistirati i reagirati. Primjerice, digitalni objekti mogu biti postavljeni u stvarnom prostoru i reagirati na promjene u tom prostoru. Jedan je od najinovativnijih uređaja koji pruža iskustvo mješovite stvarnosti ***Microsoft HoloLens***. Drugi je primjer ***Magic Leap One***, koji korisnicima omogućava da vide holografske projekcije u stvarnom svijetu. MR-om korisnici mogu manipulirati virtualnim objektima u stvarnom okruženju, pružajući dublji sloj interakcije i integracije.

- **Dizajn i inženjering:** inženjeri i dizajneri mogu se koristiti MR-om za vizualizaciju i mijenjanje 3D modela u stvarnom prostoru.

- **Medicina:** liječnici se mogu koristiti MR naočalama da bi vidjeli 3D snimke unutar tijela pacijenta u stvarnom vremenu dok obavljaju operaciju.

- **Kolaborativno radno okruženje:** timovi se mogu koristiti MR-om za zajednički rad na projektima u digitalnom prostoru, čak i ako su fizički udaljeni, dijeleći digitalne objekte i informacije u stvarnom okruženju.

Svaka od tih tehnologija, iako se razlikuje svojom primjenom, pruža korisnicima mogućnost doživljaja i interakcije s digitalnim informacijama na nov i uzbudljiv način. Ti su primjeri samo vrh ledenog brijega kada je riječ o mogućnostima i primjenama tehnologija alternativne stvarnosti.

5.1. Proširena stvarnost

Proširena stvarnost (eng. *Augmented Reality* – AR): na raskrižju je stvarnoga i virtualnog, stvarajući simbiozu koja pokreće maštovit doživljaj svijeta, povećavajući ga i obogaćujući. Ta se tehnologija sastoji od digitalnih slojeva informacija, bilo da se radi o slikama, zvukovima, tekstovima ili podatcima, a koji su iznad konkretne, fizičke stvarnosti, stvarajući isprepletanost dvaju svjetova i novu dimenziju interakcije. Pogledom kroz **leću proširene stvarnosti**, svijet postaje interaktivna pozornica na kojoj se stvarnost i digitalni sadržaji međusobno prožimaju, stvarajući jedinstvena, obogaćena iskustva. Korisnici nisu samo promatrači, oni su sudionici, protagonisti nove stvarnosti koja se oblikuje pred njihovim očima, odvijajući se u realnom vremenu i prostoru, omogućavajući istraživanje, kreiranje, i komunikaciju na načine koji su prije bili nezamislivi.

AR nosi sa sobom ne samo inovativne perspektive u smislu zabave i igara već i ogromni potencijal u brojnim industrijskim, obrazovnim, medicinskim, i mnogim drugim sektorima. Na primjer, AR u obrazovanju može omogućiti učiteljima i studentima istraživanje proširenih modela i eksperimentiranje. U svijetu trgovine AR pruža jedinstvene mogućnosti istraživanja proizvoda i personalizirano iskustvo, dok inženjere i dizajnere može opremiti alatima za vizualizaciju i manipulaciju proširenih prototipova i modela. Međutim, iza svih očaravajućih iskustava koja AR tehnologija omogućava, stoji i izazov razumijevanja i odgovornog oblikovanja tih isprepletenih stvarnosti. Za primjer je uzet *Pokémon Go*, popularna AR igra koja se koristi GPS-om za postavljanje virtualnih bića u stvarni svijet. Iako je igra donijela inovativni pristup igrama na otvorenom i potaknula fizičku aktivnost, susrela se i s brojnim izazovima i kritikama. Primjerice, neki su se igrači našli u opasnim situacijama zanemarujući svoju okolinu u potrazi za virtualnim bićima. Također, neki su povijesni spomenici i osjetljiva mjesta bili uključeni kao lokacije u igri, što igrači ponekad nisu odgovarajuće poštovali. Razvojni tim se suočio s izazovima u uravnoteženju privlačnog igračkog iskustva, poštivanju privatnosti i sigurnosti igrača te poštivanju javnih i privatnih prostora. To ilustrira koliko je važno razmišljati o etičkim i društvenim implikacijama AR tehnologija i iskustava koja one stvaraju, te koliko je ključno pažljivo planiranje i implementacija zaštite i smjernica unutar AR aplikacija i iskustava.

Kako oblikovati iskustva koja su ne samo tehnički impresivna već i etički, socijalno i psihološki osvještena? Kako stvarati proširene svjetove koji su pristupačni, inkluzivni i koji obogaćuju svakodnevicu, a ne odvode od temeljne stvarnosti?

Te i mnoge druge teme predstavljaju fascinantno putovanje na kojem se istražuje krećući se domenom proširene stvarnosti. U sljedećem poglavlju bit će obrađeni temeljni koncepti AR-a, uključujući rad s kamerama, prepoznavanje i praćenje objekata, te kreiranje i postavljanje virtualnih objekata u stvarni prostor.

5.1.1. Sidrišta (eng. anchors)

Sidrišta (eng. *anchors*) u proširenoj stvarnosti (AR)

U kontekstu proširene stvarnosti (AR), **sidrišta** (eng. *anchors*) su ključna komponenta koja omogućuje virtualnim objektima da ostanu postojano i dosljedno pozicionirani u odnosu na stvarni svijet. Da bi se razumjela važnost sidrišta, najprije je potrebno razumjeti osnovne izazove AR-a.

Kada se virtualni objekt postavi u AR okruženje, želi se osigurati da taj objekt ostane na točno određenom mjestu u stvarnom svijetu, čak i kad se korisnik kreće ili mijenja perspektivu kamere. Na primjer, ako se virtualna vaza postavi na stvarni stol, želi se postići da vaza ostane na tom stolu bez obzira na to kako se korisnik kreće oko stola. Ovdje dolazi do izražaja uloga sidrišta, a u nastavku će biti objašnjeno kako sidrišta funkcioniraju u AR-u.

- **Referentna točka**: sidrište djeluje kao referentna točka u stvarnom svijetu na koju je „prikvačen“ virtualni objekt. To osigurava da virtualni objekt ostane postojano vezan za odabranu točku bez obzira na promjene perspektive ili položaja kamere.
- **Praćenje i ažuriranje:** AR sustavi neprestano prate i ažuriraju poziciju sidrišta u odnosu na stvarni svijet pomoću tehnika kao što je **SLAM** (eng. *Simultaneous Localization and Mapping*). Navedeno osigurava da, čak i kad se korisnikovo okruženje mijenja, virtualni objekti ostaju dosljedno pozicionirani.
- **Višestruka sidrišta:** u složenijim AR aplikacijama moguće se koristiti višestrukim sidrištima kako bi se upravljalo položajem i orijentacijom više virtualnih objekata u različitim dijelovima stvarnog prostora.

U svakodnevnoj uporabi sidrišta omogućuju aplikacijama kao što su AR igre, alati za dizajn interijera ili aplikacije za navigaciju pružiti korisniku uvjerljivo i stabilno AR

iskustvo. Točnim i efikasnim sidrenjem virtualnih objekata u stvarnom svijetu, korisnici mogu interagirati s virtualnim sadržajem na način koji se čini prirodnim i uvjerljivim. U svijetu proširene stvarnosti (AR), sidrišta igraju ključnu ulogu u povezivanju digitalnoga i stvarnog okruženja. Ona služe kao referentne točke koje omogućuju digitalnim objektima da se „prilijepe“ za specifične lokacije ili objekte u stvarnom svijetu.
U nastavku će biti izdvojeno nekoliko glavnih tipova sidrišta koji se često koriste u AR-u.

- **Položajna sidrišta (eng. *World Anchors*)** – povezuju se s konkretnom lokacijom u stvarnom svijetu. Dok se korisnikov uređaj kreće, položajna sidrišta osiguravaju da virtualni objekti ostanu na istom mjestu u odnosu na stvarni svijet.
- **Sidrišta slike (eng. *Image Anchors*)** – vrsta su položajnog sidrišta koja se koristi prepoznatljivim slikama kako bi odredila i zadržala svoj položaj u AR okruženju.
- **Točkovna sidrišta (eng. *Point Anchors*)** – fiksiraju se na specifične točke u stvarnom svijetu. Često se koriste za postavljanje statičnih objekata u prostoru.
- **Planarna sidrišta (eng. *Plane Anchors*)** – prate ravne površine, kao što su podovi ili stolovi, i omogućuju virtualnim objektima da se dinamički prilagođavaju promjenama u stvarnom okruženju.
- **Prilagodljiva sidrišta (eng. *Adjustable Anchors*)** – omogućuju korisnicima ručnu prilagodbu položaja i orijentaciju virtualnih objekata prema svojim potrebama.
- **Izvorno sidro (eng. *Origin Anchor*)** – početna referentna točka svakog AR iskustva. Kada se AR aplikacija pokrene, izvorno sidro definira početnu točku (0, 0, 0) AR svijeta.

Unutar područja proširene stvarnosti (AR), terminologija se brzo razvija i mijenja. Navedeno je posljedica brzoga tehnološkog napretka, različitih inovacija i pristupa koji se kontinuirano istražuju, kao i raznih tvrtki i istraživača koji doprinose razvoju toga područja. Kako se AR tehnologija stalno mijenja i evoluira, tako se pojmovi i nazivi prilagođavaju novim otkrićima i metodama. Zbog toga trenutačno ne postoji univerzalno prihvaćena ili standardizirana terminologija koja obuhvaća sve aspekte AR-a. Dva izvora ili stručnjaka mogu koristiti različite termine za isti koncept ili funkciju. To može stvoriti izazove, posebno onima koji su novi u ovom području, ili onima koji prelaze s jedne platforme ili alata na drugi.

Programerima, dizajnerima i istraživačima u AR-u ključna je fleksibilnost i prilagodljivost kada je riječ o terminologiji. Preporučljivo se uvijek osloniti na najnovije relevantne izvore informacija, pratiti industrijske trendove i, gdje je to moguće, konzultirati se s kolegama i zajednicom kako bi se postigla jasnoća i dosljednost u komunikaciji.

5.1.2. AR Podloga (eng. AR Foundation)

AR podloga okvir je unutar *Unity* razvojnog okruženja namijenjen za izradu aplikacija proširene stvarnosti (AR). Glavna je prednost AR podloga što omogućuju razvojnim inženjerima pisanje jedinstvene aplikacije koja će raditi na više AR platformi, uključujući ***ARCore*** (za Android uređaje) i ***ARKit*** (za iOS uređaje).

U suštini, umjesto da programeri moraju pisati zasebne kodove za svaku platformu, **AR podloga** omogućuje pisanje koda samo jednom, koji će zatim biti kompatibilan s većinom glavnih AR platformi. Osim što štedi vrijeme, to također smanjuje mogućnost grešaka jer programeri ne moraju održavati više verzija svoje aplikacije. AR podlogom, *Unity* pruža niz alata i značajki koje olakšavaju implementaciju ključnih AR funkcionalnosti, kao što su detekcija ravnih površina, praćenje pokreta, prepoznavanje slika, i mnogih drugih. Zahvaljujući AR podlozi, razvojni inženjeri mogu iskoristiti puni potencijal ***ARCorea*** i ***ARKita***, dviju vodećih tehnologija u svijetu proširene stvarnosti, bez potrebe za dubokim razumijevanjem svake od njih zasebno. Umjesto toga, mogu se usredotočiti na stvaranje inovativnih AR iskustava u familijarnom *Unity* okruženju.

Funkcionalnosti AR podloge:

- **univerzalni pristup:** AR podloga omogućava programerima pisanje kod jednom a korištenje u više AR platformi, uključujući ***ARKit*** (*Apple*), ***ARCore*** (*Google*), ***Magic Leap***, i druge
- **komponente za AR:** pruža set osnovnih komponenti potrebnih za kreiranje AR iskustava, uključujući praćenje lica, praćenje slike, praćenje ravnih površina, i mnogo više
- **integracija s *Unity* uređivačem:** programeri se mogu koristiti brojnim alatima *Unity* uređivača za dizajn, testiranje i debugiranje svojih AR aplikacija
- **proširivost:** iako AR podloga pruža osnovne funkcionalnosti za AR, također je dizajnirana da bude proširiva, što omogućava programerima dodavanje prilagođenih funkcionalnosti ili korištenje dodatcima trećih strana.

Osnovne komponente AR podloge

AR podloga, kao *Unityjev* okvir za razvoj AR aplikacija, sadrži niz komponenti koje omogućuju različite funkcionalnosti povezane s proširenom stvarnošću.

Ključne su komponente AR podloge:

- **AR sesija (eng. *AR Session*):** ta komponenta upravlja životnim ciklusom AR iskustva, uključujući inicijalizaciju i zaustavljanje AR sesija
- **porijeklo AR sesije (eng. *ARSessionOrigin*):** predstavlja referentnu točku ili izvorište svim AR objektima unutar scene. Tipično je povezano s kamerom za pravilno postavljanje i skaliranje AR sadržaja u odnosu na stvarni svijet
- **upravitelj AR kamere (eng. *ARCameraManager*):** omogućuje pristup videoprikazu iz kamere i upravlja njezinim postavkama
- **upravitelj AR točkastog oblaka (eng. *ARPointCloudManager*):** ta komponenta vizualizira točkasti oblak koji *ARCore* ili *ARKit* koristi za razumijevanje okoline
- **upravitelj AR ravnina (eng. *ARPlaneManager*):** detektira i prati ravnine u stvarnom svijetu, poput podova ili stolova
- **upravitelj AR sidrišta (eng. *ARAnchorManager*):** omogućuje postavljanje sidrišta na specifične točke u stvarnom svijetu da bi se virtualni objekti mogli točno postaviti u prostoru
- **upravitelj praćenja slika AR (eng. *ARImageTrackingManager*):** omogućuje prepoznavanje i praćenje 2D slika u stvarnom svijetu
- **upravitelj AR pucanja zraka (eng. *ARRaycastManager*):** omogućuje izvođenje zraka (*raycasts*) iz točke u virtualnom prostoru prema stvarnom svijetu, što se često koristi za postavljanje objekata u AR
- **upravitelj AR okolišnih sonda (eng. *AREnvironmentProbeManager*):** detektira informacije o svjetlu i refleksijama iz okoline za pružanje realističnoga osvjetljenja virtualnih objekata
- **upravitelj AR lica (eng. *ARFaceManager*):** služi za praćenje lica koristeći prednju kameru na uređajima koji podržavaju prepoznavanje lica.

To su samo osnovne komponente koje AR podloga nudi. S obzirom na to da se AR tehnologija brzo razvija, moguće je da će se u budućnosti dodavati nove komponente

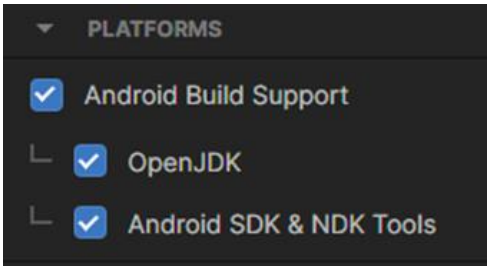
i funkcionalnosti. Ako se koristi AR podloga za razvoj, preporučuje se redovito provjeravati dokumentaciju *Unityja* za informacije o svim novostima i ažuriranjima.

5.1.3. Instalacija Android modula

Postavljanje *Unityja* za izradu Android aplikacija zahtijeva dodatne module radi pravilnoga funkcioniranja i kompatibilnosti s Android platformom. Ti moduli omogućuju *Unityju* komunikaciju s Android SDK-om i izradu .apk ili .aab datoteke, koje se mogu instalirati na Android uređaje.

Da bi se postavio *Unity* za razvoj Android aplikacija, potrebno je slijediti korake:

1. **korak – pokretanje *Unity Huba*:** to je centralizirano mjesto za upravljanje svim instalacijama *Unityja*, projektima i dodatcima
2. **korak – navigacija do instalacije (eng. *Installs*):** na lijevoj bočnoj traci *Unity Huba* treba kliknuti na *Installs*
3. **korak – dodavanje modula:** ovdje se može vidjeti popis svih instaliranih verzija *Unityja*. Potrebno je pronaći verziju trenutnoga projekta i kliknuti na tri vodoravne točke s desne strane te odabrati „Dodaj module“ (eng. *Add Modules*)
4. **korak – odabir *Android Build Support*:** u popisu dostupnih modula treba pronaći ***Android Build Support*** i označiti ga. Također, potrebno je instalirati podmodule poput ***Android SDK & NDK Tools*** i ***OpenJDK*** ako već nisu instalirani, da bi se osiguralo raspolaganje svim potrebnim alatima za razvoj Android aplikacija.



Slika 56. Android moduli (slika zaslona, Unity Hub)

5. **korak – instalacija modula:** nakon što su svi potrebni moduli odabrani, treba kliknuti na gumb ***Next*** ili ***Install*** za početak instalacije. Ovisno o internetskoj vezi, preuzimanje i instalacija mogu potrajati neko vrijeme.
6. **korak – provjera konfiguracije:** nakon što su svi potrebni moduli uspješno instalirani, sve je spremno za izradu Android aplikacija. Uvijek je dobra praksa provjeriti postavke projekta i osigurati da su sve specifične za Android konfigurirane prema potrebama prije nego što se počne s izgradnjom aplikacije.

5.1.4. Stvaranje AR projekta

U nastavku će se započeti s kreiranjem projekta proširene stvarnosti (AR) i bit će korišten ranije kreirani ***Environment*** *Unity* paket. Ukoliko se iz bilo kojeg razloga nije moguće koristiti spomenutim paketom, utoliko se može nastaviti i bez njega, koristeći se jednostavnim 3D *Cube* objektom za demonstraciju osnovnih principa.

Kreiranje novog projekta s *ARCore* predloškom

Potrebno je kliknuti na ***Projects*** unutar *Unity Huba*, a zatim na ***New***.

Korištenje predloška: iz dostupnih predložaka za novi projekt treba odabrati ***ARCore***. Ukoliko *ARCore* predložak nije vidljiv u opcijama, moguće je da ga se mora preuzeti i instalirati pritiskom na gumb ***Download Template***.

Imenovanje i lokacija: potrebno je dodati ime projekta, „**Moj AR Projekt**“, i odrediti kamo ga se želi spremiti.

Kreiranje projekta: potrebno je kliknuti ***Create*** kako bi se stvorio AR projekt.

Slijedeći navedene korake, predložak *ARCorea* bi bio postavljen i spreman za dodatnu konfiguraciju i razvoj. Odabrani predložak osnovni je kostur projekta, omogućujući implementaciju specifičnih funkcionalnosti i značajki vezanih uz određenu platformu ili tip projekta, u ovom slučaju, *ARCore* za razvoj proširene stvarnosti.

Korištenje predloškom *ARCore* u *Unity Hubu* predstavlja praktičan početni korak u razvoju projekata proširene stvarnosti (AR) zbog niza prethodno postavljenih konfiguracija i već integriranih paketa neophodnih za AR razvoj. No, razumijevanje suštine postavljanja i konfiguriranja AR projekta od osnovne 3D postavke može obogatiti tehničko znanje i vještine, pružajući fleksibilnost i sigurnost ručnoga konfiguriranja AR projekata u budućnosti.

Kada se kreira nova scena u *Unityju* koristeći se *ARCore* predloškom, unaprijed će biti nekoliko ključnih komponenti automatski postavljeno i konfigurirano za razvoj proširene stvarnosti (AR). Taj predložak osigurava temeljni skup objekata i postavki koje omogućuju automatsko započinjanje s izgradnjom AR aplikacija na temelju platforme *ARCore*.

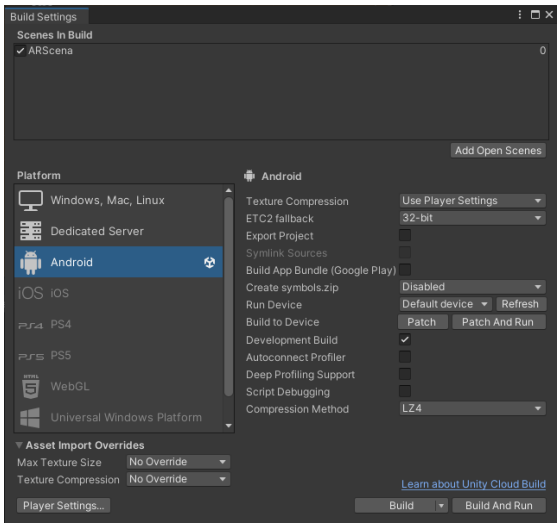
Usmjereni svjetlosni izvor (eng. *Directional Light*)

- **AR sesija (*AR Session*)** komponenta je koja upravlja i održava sve relevantne AR podatke i stanje sesije tijekom trajanja AR aplikacije. To uključuje praćenje položaja uređaja, upravljanje AR resursima i životnim ciklusom, kao i pohranu i ažuriranje podataka AR sidrišta i ravnina. U osnovi, *AR Session* je ključna komponenta koja omogućuje interakciju između *Unity* aplikacije i *ARCore* funkcionalnosti na uređaju.
- ***AR Session Origin*** ključan je za upravljanje prostornim referencama unutar AR scene. Taj objekt obično sadrži kameru koja predstavlja korisnikov pogled na AR svijet. *AR Session Origin* koristi se za skaliranje i transformaciju prostornih podataka iz *ARCorea* u koordinatni prostor *Unityja*. Pomaže u pravilnom pozicioniranju i skaliranju virtualnih objekata u odnosu na stvarni svijet, omogućavajući tako učinkovite i točne interakcije između virtualnih i fizičkih elemenata.

Nakon što je ***Directional Light*** odabran, jednostavno treba pritisnuti tipku ***Delete*** na tipkovnici ili desnim klikom miša odabrati opciju ***Delete*** iz kontekstnog izbornika. To će ukloniti objekt iz scene koju se kreira. Razlog zašto je uklanjanje *Directional Light* prihvatljivo i poželjno u ovom kontekstu, leži u činjenici da ***Environment*** *prefab* već sadrži integrirane svjetlosne elemente.

5.1.5. Konfiguriranje AR postavki za željenu platformu

Jednom kada je aplikacija ili igra razvijena, gradnja ili *building* aplikacije u *Unityju* znači stvaranje izvršne datoteke koja se može pokrenuti na ciljanoj platformi, bilo da je riječ o računalu, mobilnom uređaju, mreži ili nekoj drugoj platformi. U nastavku će biti objašnjeno kako izvesti proces izgradnje aplikacija korak po korak.



Slika 57. Postavke izgradnje za Android (slika zaslona, Unity)

Proces započinje navigacijom u *Unityju* prema ***File*** > ***Build Settings***. Ovdje je potrebno odabrati željenu platformu (primjerice, *Android*) za izradu AR aplikacije te zatim odabrati ***Switch Platform***. Trenutačno otvorenu scenu se može uključiti unutar sekcije ***Scenes in Build*** klikom na ***Add Open Scenes*** te označavanjem ***Development Build***.

Slika 58. Postavke XR dodataka (slika zaslona, Unity)

U nastavku je potrebno otići do ***Edit*** > ***Project Settings*** i odabrati ***XR Plug-in Management*** iz izbornika s lijeve strane. Na kartici ***Android*** zatim treba potvrditi kućicu pored ***Google ARCore*** da bi se osigurala podrška *ARCorea* za AR aplikaciju koju se izrađuje.

Vulkan API

Vulkan, kao suvremeni grafički API (eng. *Application Programming Interface*), predstavlja značajan iskorak u odnosu na prethodnike poput *OpenGL ES*-a. Njegova arhitektura dizajnirana je s ciljem pružanja visokih performansi i optimizirane uporabe hardvera. Uvođenjem *Vulkana*, *Google* je s *Androidom 7.0 Nougat* želio potaknuti razvoj aplikacija koje bi mogle bolje iskoristiti grafičke resurse uređaja, pružajući korisnicima glađe i vizualno impresivnije iskustvo. Međutim, unatoč prednostima koje *Vulkan* nudi, njegova integracija nije uvijek jednostavna ni optimalna za sve uređaje. Hardverska raznolikost Android ekosustava znači da neki uređaji, posebno stariji

modeli ili oni niže klase, nemaju potrebne specifikacije ili ažurirane GPU upravljačke programe potrebne za podršku *Vulkanu*. To može rezultirati da aplikacije koje koriste *Vulkan* ne funkcioniraju ispravno ili ne pružaju očekivane značajke na tim uređajima.

Osim samih tehničkih prepreka, postoje i praktične dimenzije koju programeri moraju uzeti u obzir: **stabilnost** i **kompatibilnost**. Implementacija novog API-ja kao što je *Vulkan* može dovesti do neočekivanih problema ili poteškoća koje nisu prisutne kada se koristi stariji, ali dokazano stabilni API poput *OpenGL ES*-a. Ovisno o ciljanoj publici i uređajima na kojima aplikacija treba raditi, programeri moraju pažljivo razmotriti hoće li koristiti *Vulkan* ili će se osloniti na provjerene tehnologije da bi osigurali nesmetan rad aplikacije. U konačnici, dok *Vulkan* nudi značajne prednosti u određenim scenarijima, njegova primjena zahtijeva pažljivo razmatranje i testiranje da bi se osigurala optimalna korisnička iskustva na različitim Android uređajima.

UPUTE ZA DEAKTIVACIJU

Ako je *Vulkan* aktivan, da bi ga se deaktiviralo potrebno je pratiti sljedeće korake.

1. korak – otvoriti ***Edit*** > ***Player Settings*** > ***Other***.

2. korak – unutar ***Other*** potrebno je potražiti opciju ***Graphic APIs*** u odjeljku ***Rendering*** te je zatim deaktivirati.

3. korak – s označenim *Vulkanom* treba pritisnuti ikonu „-“ da bi ga se izbrisalo iz popisa. Na taj način *Vulkan* API neće biti aktivan u aplikaciji.

Android API

Android API (eng. *Application Programming Interface*) predstavlja skup softverskih sučelja koje operativni sustav Android nudi. Ta sučelja omogućuju programerima pristup specifičnim funkcijama i uslugama Android uređaja u njihovim aplikacijama. Na primjer, želimo li da aplikacija pristupi kontaktima na uređaju, koristit ćete se određenim dijelom *Android* API-ja namijenjenim upravljanju kontaktima. Svaka nova verzija Androida donosi i novu verziju API-ja s dodatnim funkcionalnostima, poboljšanjima i promjenama. *ARCore*, *Googleova* platforma za proširenu stvarnost, ima određene zahtjeve koji se tiču hardvera i softvera. Te zahtjeve, kao što su napredni senzori,

algoritmi za praćenje pokreta i optimizirane značajke, ne možemo pronaći na starijim verzijama Androida. Upravo zato *Android 7.0 Nougat* i noviji operativni sustavi nude bolju podršku za *ARCore*, osiguravajući pouzdana i učinkovita iskustva proširene stvarnosti (AR). Da bi se osiguralo ispravno funkcioniranje *ARCore* aplikacije, preporučuje se postaviti minimalnu razinu Android API-ja na 24.

UPUTE ZA POSTAVLJANJE

Želi li se postaviti tu minimalnu razinu za projekt, u *Unityju* je potrebno otvoriti ***Player Settings***, zatim otići do ***Other*** te odabrati ***Identification***. Tu je moguće postaviti ***Minimum API Level*** na ***Android 7.0 Nougat***.

5.1.6. APK i AAB

APK (eng. *Android Package Kit*) je format datoteke koji koristi operacijski sustav Android za distribuciju i instalaciju mobilnih aplikacija i srednjeg sloja. U suštini, APK datoteka funkcionira slično kao što **.exe datoteka** radi na *Windows* računalima. Kada se aplikacija preuzima iz trgovine *Google Play* ili bilo koje druge Androidove trgovine, zapravo se preuzima APK datoteka na uređajv te se zatim automatski instalira.

Osim aplikacija, APK datoteke mogu sadržavati i ostale komponente potrebne za ispravno funkcioniranje aplikacije, poput slika, skripti ili drugih resursa. APK datoteke omogućuju korisnicima da zaobiđu trgovinu *Google Play* i instaliraju aplikacije direktno na vlastiti uređaj pomoću bočnog učitavanja *sideloadinga*. Međutim, važno je biti oprezan prilikom instaliranja APK datoteka s nepoznatih izvora zbog potencijalnih sigurnosnih rizika, kao što su *malware* i druge štetne aplikacije. Uvijek je preporučljivo preuzimati i instalirati APK datoteke samo iz pouzdanih izvora, tipa *Google Play Storea* i vlastitih aplikacija.

AAB (eng. *Android App Bundle*) je noviji format pakiranja aplikacija koji je predstavio *Google* kako bi optimizirao način distribucije *Android* aplikacija u trgovini *Google Play*. Za razliku od APK-a, koji predstavlja konkretnu aplikaciju koja se instalira na uređaj, AAB je kolekcija svih mogućih APK-ova koje određena aplikacija može stvoriti. Sadrži sve komponente aplikacije (kod, resurse, manifeste), ali omogućuje *Google Playu* stvarati i dostaviti APK specifičan za određenu kombinaciju karakteristika korisničkih

uređaja (npr. arhitektura procesora, rezolucija zaslona, jezik itd.). To korisnicima rezultira manjom veličinom preuzimanja i instalacije.

Dok je APK tradicionalni format za distribuciju i instalaciju aplikacija, AAB predstavlja novi pristup koji omogućuje bolju optimizaciju i efikasnost. Kada razvojni programer postavi vlastitu aplikaciju u trgovinu *Google Play* koristeći AAB format, ona se koristi njime da bi dinamički generirala i poslala optimizirani APK za svaki pojedini uređaj. Taj pristup osigurava korisnicima preuzimanje samo onih dijelova aplikacije koji su im specifično potrebni, čime se smanjuju veličina preuzimanja i resursi potrebni za instalaciju.

5.1.7. Omogućavanje razvojnog načina na Android uređaju

Prije nego što se započne s razvojem aplikacija za Android ili testiranjem određene funkcionalnosti na stvarnom uređaju, često je potrebno omogućiti „**Razvojni način**“ (eng. *Development Mode*) na uređaju Android. Time se pristupa nizu razvojnih alata i opcija specifičnih programerima.

> ⚠ **VAŽNA NAPOMENA**
>
> Sljedeći koraci opisuju opći postupak za većinu Android uređaja. Međutim, ovisno o proizvođaču i verziji Androida, koraci se mogu malo razlikovati. Ako sljedeće upute ne odgovaraju uređaju kojim se koristi, preporučuje se potražiti specifične upute za model uređaja.

Koraci omogućavanja „Razvojnog načina“:

1. korak – **pristup informacijama o uređaju:** na Android uređaju treba otvoriti „**Postavke**“ (eng. *Settings*)
2. korak – potrebno se pomaknuti prema dolje i odabrati „**O telefonu**“ (eng. *About phone*) ili sličnu opciju
3. korak – **otključavanje opcija za razvojne programere:** unutar informacija o telefonu potrebno je pronaći „**Broj *builda***“ (eng. *Build number*)
4. korak – potrebno je pritisnuti na „**Broj *builda***“ nekoliko puta (obično 7) dok se ne zaprimi poruka koja informira o statusu razvojnog programera. Možda će biti potrebno unijeti PIN ili lozinku uređaja.

Pristup opcijama za razvojne programere

Za početak ovog procesa, potrebno se vratiti na glavni izbornik „**Postavke**“. U njemu bi se sada trebala vidjeti nova opcija nazvana „**Opcije za razvojne programere**“ (eng. *Developer options*). Potrebno ju je odabrati.

Aktivacija „Razvojnog načina“

U „**Opciji za razvojne programere**“ treba potražiti i omogućiti „**USB ispravljanje pogrešaka**“ (eng. *USB Debugging*) i druge relevantne opcije potrebne za razvoj.

POKRETANJE NA UREĐAJU

Želi li se postaviti tu minimalnu razinu projekta, u Unityju je potrebno otvoriti ***Player Settings***, zatim doći do ***Other*** te odabrati ***Identification***. Ovdje se može postaviti ***Minimum API Level*** na ***Android***.

U prozoru ***Build Settings*** treba potražiti i odabrati „Osvježi“ (eng. ***Refresh***) unutar sekcije „Uređaj za pokretanje“ (eng. ***Run Device***). Ta radnja omogućava *Unityju* detektirati sve trenutno spojene uređaje. Nakon osvježavanja unutar iste sekcije pojavit će se padajući izbornik s popisom povezanih uređaja. Ako je Android uređaj ispravno povezan, trebalo bi ga se moći pronaći na tom popisu. Ako nije vidljiv, treba pogledati upute koje slijede ispod.

Za pokretanje testiranja aplikacije treba odabrati željeni uređaj iz popisa. Time vlastiti uređaj postavljamo kao „Uređaj za pokretanje” (eng. *Run Device*). Nakon toga, potrebno je slijediti upute odabirom ***Build and Run***. Nakon što se odredi željeno mjesto pohrane izlazne datoteke, i kako ju se želi imenovati, *Unity* će sam pripremiti i aktivirati aplikaciju na odabranom uređaju.

Ako uređaj nije prepoznat, preporučuje se poduzeti sljedeće.

Isključiti Android uređaj s USB kabela, potom se uputiti u „Postavke“ te pristupiti „Opcijama za razvijatelje“. Ondje je potrebno isključiti, a potom i ponovno uključiti opciju USB ispravljanje pogrešaka (eng. *USB Debugging*). Obavezno je potrebno dodirnuti i „Opozovi pristup USB ispravljanju pogrešaka“ (eng. *Revoke USB Debugging Access*). Nakon tih koraka treba se povezati uređaj s računalom pomoću USB-a te kada sustav pošalje upit za dopuštenje prijenosa podataka, odgovoriti s „Da“ (eng. *Yes*).

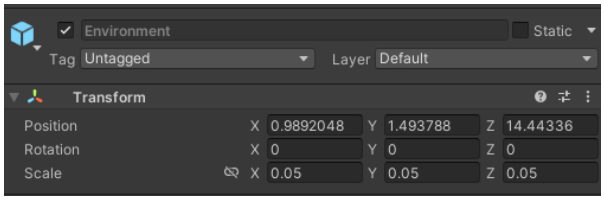
Sada bi trebala biti vidljiva AR scena u akciji! Izrađivana aplikacija bi trebala automatski koristiti kameru uređaja za prikaz AR sadržaja na ekranu.

5.1.8. Uvoz Unity paketa

U nastavku će biti integriran prethodno kreirani paket ***Environment*** unutar projekta.

1. **Uvoz paketa:** potrebno je odabrati opciju ***Import Package"*** > ***"Custom Package*** unutar izbornika ***Assets***. Ta opcija omogućava uvoz vlastitih, prethodno kreiranih paketa.
2. **Lociranje i selekcija paketa:** nakon što je ***Custom Package*** odabran, pojavit će se prozor za istraživanje datoteka. Tu je potrebno pronaći i odabrati prethodno kreiranu datoteku *.unitypackage* koju se želi uvesti. U ovom slučaju, traži se paket ***Environment***.
3. **Pregled i uvoz:** kada se paket odabere, otvorit će se prozor koji omogućava pregled svih dostupnih resursa. Tu se može odabrati komponente koje se želi uvesti, potom treba kliknuti ***Import*** za dodavanje resursa u projektni radni prostor.
4. **Provjera i organizacija imovine:** nakon uspješnog uvoza, u direktoriju ***Assets*** u *Unity* uređivaču treba provjeriti jesu li svi potrebni elementi iz ***Environment*** paketa uspješno dodani u projekt. Tu se može dodatno organizirati i rasporediti imovinu prema vlastitim preferencijama unutar različitih mapa i podmapa da bi se održavala urednost i konzistentnost projekta.

U procesu kreiranja AR aplikacija vrlo je važno osigurati da su svi objekti i okolina pravilno skalirani u odnosu na stvarni svijet. Zbog različitih načina na koje virtualni i stvarni prostor mogu biti interpretirani unutar aplikacije, može se dogoditi da virtualna okolina ili objekti budu preveliki ili premali kada se postave u AR prostor. U tom slučaju, može se primijetiti da je uvezeni *Environment prefab* značajno prevelik kad je instanciran u AR prostoru koji se gradi, stoga treba prilagoditi njegovu veličinu.



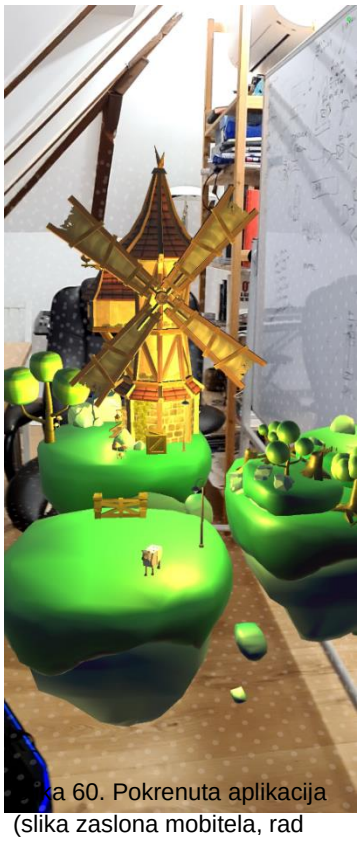
Potrebno je kliknuti na *prefab* za njegov odabir.

U prozoru „Inspektor“ treba pronaći polje ***Scale*** unutar komponente *Transform*.

Slika 59. Postavke objekta „Environment“ (slika zaslona, Unity)

Unosi se vrijednosti 0.05 u X, Y, i Z polja skale, čime se smanjuje veličina *prefaba* na 5 % originalne veličine.

5.1.9. Implementacija dodira i instanciranje Prefaba u AR



ka 60. Pokrenuta aplikacija (slika zaslona mobitela, rad autora)

Za kreiranje dojmljive AR aplikacije bit će implementirana funkcionalnost dodira za mobilne uređaje i omogućeno instanciranje *prefaba* okoline (*environment prefab)* na ravne površine u stvarnom svijetu. Treba omogućiti korisnicima postavljanje virtualne okoline u stvarni prostor jednostavnim dodirivanjem ekrana. S lijeve se strane može vidjeti kako *prefab* okoline izgleda u stvarnom svijetu

Korištenje dodirom umjesto tipkovnicom: u prethodnim scenama korišten je tipkovnički unos za upravljanje svjetlima. S obzirom na to da mobilni uređaji nemaju fizičke tipke za takve interakcije, koristit će se dodir ekrana kao metoda unosa.

Potrebno je kreirati skriptu ***PlaceObjectOnPlane*** i stavite je na objekt ***AR Session Origin*** u „Hijerarhiji“. Cilj je kreirati skriptu koja će, dodirujući ekran, omogućiti postavljanje objekta na ravnu horizontalnu površinu. Taj proces, u osnovi, uključuje kombinaciju detekcije dodira na ekranu uređaja i AR tehnologiju za identificiranje stvarnog svijeta, prepoznavanje površina, i stavljanje virtualnih objekata u taj kontekst.

Unesite sljedeći kod na vrh skripte:

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;
```

Kombiniranjem tih biblioteka u kodu, omogućuje se pristup svim potrebnim funkcijama i tipovima podataka za razvijanje efikasne i funkcionalne AR aplikacije. Uvijek neka su te biblioteke uključene na vrhu AR skripti u *Unityju* da bi se izbjegle potencijalne greške pri kompilaciji i osigurao optimalan rad aplikacije. Ispod toga u glavno tijelo skripte treba dodati sljedeći kod:

```
public class PlaceObjectOnPlane : MonoBehaviour
```

```
{
    // Varijabla koja će držati naš prefab okoline.
    [SerializeField] private GameObject environmentPrefab;
    // Varijabla za upravljanje AR raycastingom.
    private ARRaycastManager arRaycastManager;
    // Lista za pohranu rezultata raycasta.
    private List<ARRaycastHit> hits = new List<ARRaycastHit>();
    // Varijabla koja će pohranjivati instancirani objekt.
    private GameObject placedObject;

    // Metoda koja se poziva prije prvog okvira.
    private void Awake()
    {
        // Dohvaćanje komponente ARRaycastManager s ovog GameObjecta.
        arRaycastManager = GetComponent<ARRaycastManager>();
    }
```

GetComponent<ARRaycastManager>()*:** metoda *GetComponent* koristi se za dohvaćanje referenci na komponente koje su pridružene *GameObjectu* na kojem se skripta izvodi. U ovom slučaju, koristi se za dohvaćanje komponente ***ARRaycastManager koja omogućava upravljanje ***raycastingom*** unutar AR prostora. R*aycasting* omogućava korisnicima na prirodan način umetnuti i manipulirati virtualnim entitetima unutar AR prostora, povezujući digitalni i fizički svijet na jedinstven i zanimljiv način. Daljnji detalji i objašnjenja bit će predstavljeni kasnije u priručniku.

Zatim, u metodu *Update* treba dodati kod za praćenje dodira:

```
    private void Update()
    {
        // Provjerava imamo li dodir na ekranu, ako ne, izlazi iz metode Update.
        if (Input.touchCount == 0)
            return;
        // Dohvaćanje prvog dodira na ekranu.
        Touch touch = Input.GetTouch(0);
```

```
        // Provjerava je li faza dodira početak dodira.
        if (touch.phase != TouchPhase.Began)
            return;
        // Izvođenje raycasta od pozicije dodira i pohranjivanje rezultata u hits.
        if (arRaycastManager.Raycast(touch.position, hits, TrackableType.Planes))
        {
            // Dohvaćanje pozicije i rotacije prvog pogotka.
            Pose pose = hits[0].pose;

            // Provjerava je li objekt već postavljen.
            if (placedObject == null)
            {
                // Ako nije, instancira objekt na dobivenoj poziciji i rotaciji.
            placedObject = Instantiate(environmentPrefab, pose.position, pose.rotation);
            }
            else
            {
                // Ako jest, postavlja postojeći objekt na novu poziciju i rotaciju.
             placedObject.transform.SetPositionAndRotation(pose.position,
ose.rotation);
            }
        }
    }
}
```

U *Unity* uređivaču treba odabrati *GameObject* koji sadrži skriptu *PlaceObjectOnPlane* i povezati *prefab* okoline (eng. *environment*) povlačeći ga u odgovarajuće polje (eng. *field*) skripte, kao na slici. Time se osigurava da se virtualna okolina može instancirati u stvarnom svijetu u AR iskustvu. Zatim, za testiranje aplikacije treba navigirati na *File" > "Build Settings* u *Unity* uređivaču, potvrditi da je ciljana platforma i scena uključena u *build*, a potom odabrati ***Build And Run***. Dodirujući ekran, *prefab* okoline trebao bi se odmah postaviti na detektiranu površinu, poštujući njezinu poziciju i orijentaciju.

5.1.10. Dodirni unos

Dodirni unos (eng. *Touch Input*) u *Unityju* pruža mogućnost interakcije s mobilnim uređajima koji imaju osjetljive zaslone na dodir. Bilo da se razvija bilo koja druga vrsta aplikacije za pametne telefone i tablete, upravljanje dodirom ima ključnu važnost u kreiranju intuitivnog korisničkog iskustva. U *Unityju* svaki dodir na ekranu predstavlja se kao instanca ***Touch*** klase. Kada korisnik dodirne zaslon, može se pristupiti raznim informacijama o tom dodiru, uključujući poziciju, fazu (npr. početak, kretanje, kraj dodira) i pritisak.

Da bi se saznalo koliko prstiju trenutno dodiruje zaslon potrebno je koristiti ***Input.touchCount***. Zatim se može koristiti ***Input.GetTouch*** za informacije o određenom dodiru:

```
if (Input.touchCount > 0)
{
        Touch dodir = Input.GetTouch(0); // Dohvaća informacije o prvom dodiru.
        Vector2 pozicijaDodira = dodir.position; // Dohvaća trenutnu poziciju dodira.
}
```

Faze dodira: svaki objekt *Touch* ima svojstvo *phase*, koje označava trenutnu fazu dodira. To može biti posebno korisno za detekciju različitih tipova gesti koje se mogu pratiti u kodu na sljedeći način:

```
    switch(dodir.phase)
{
        case TouchPhase.Began:
        // Ovdje se obrada započinje kada korisnik prvi put dodirne zaslon.
        break;
        case TouchPhase.Moved:
        // To se izvršava kada korisnik premiješta prst po zaslonu.
        break;
        case TouchPhase.Ended:
        // Ovdje se obrada završava kada korisnik prestane dodirivati zaslon.
        break;
}
```

Višestruki dodiri: jedna je od prednosti rada s dodirnim uređajima mogućnost detektiranja višestrukih dodira istodobno, što je idealno za geste poput štipanja ili rotiranja. Evo kako se u kodu mogu pratiti dva dodira:

```
if (Input.touchCount == 2)
{
    Touch prviDodir = Input.GetTouch(0);
    Touch drugiDodir = Input.GetTouch(1);
    // Tu možete obraditi geste poput štipanja ili rotiranja.
```

5.1.11. Raycasting

Raycasting je proces slanja imaginarnih zraka ili *raya* iz određene točke u određenom smjeru i provjere gdje i s čime te zrake dolaze u interakciju ili se sudaraju u 3D prostoru. U kontekstu računalne grafike i igraćih motora, kao što je *Unity*, *raycasting* je često korištena tehnika za detekciju kolizija, određivanje vidljivosti ili interakciju s objektima u 3D svijetu.

U *Unityju* se *raycasting* često koristi u različitim scenarijima:

- **Detekcija kolizija:** na primjer, da bi se utvrdilo je li put između igrača i neprijatelja prepriječen zidom ili nekim drugim objektom.
- **Interakcija igrača:** kada igrač uperi pokazivač ili retikul na neki objekt u igri i klikne mišem, *raycast* može odrediti koji je objekt odabran.
- **Umetanje logike AI:** na primjer, za omogućavanje neprijateljima da „vide“ igrača ili određuju gdje će hodati.

Kako bi se izvršio *raycast* u *Unityju*, koristi se metodom „Physics.Raycast()“. Ta metoda vraća „istina“ ako zraka susretne neki objekt s koliderom, i „neistina“ ako ne. Ako *raycast* detektira koliziju, informacije o toj koliziji (poput točke dodira, udaljenosti i površine s kojom se sudarilo) mogu se pohraniti u varijabli tipa ***RaycastHit***.

Primjer jednostavnog *raycastinga*:

```
RaycastHit hit;
if (Physics.Raycast(transform.position, transform.forward, out hit, 10f))
{
    Debug.Log("Zraka je pogodila objekt: " + hit.collider.name);
}
```

U tom primjeru zraka se šalje iz trenutne pozicije objekta prema njegovu prednjem smjeru do maksimalne udaljenosti od 10 jedinica. Ako zraka pogodi neki objekt s koliderom unutar te udaljenosti, informacije o sudaru pohranjuju se u varijabli *hit*. *Raycasting* je iznimno snažan alat u razvoju igara i simulacija, omogućujući detekciju i interakciju s objektima u virtualnom svijetu na intuitivan i efikasan način.

5.1.12. AR prepoznavanje lica

Osim osnovnih koncepata, kao što su instanciranje objekata u AR prostoru i interakcija s fizičkim svijetom putem *raycastinga* postoji mnoštvo drugih značajki i komponenti koje se mogu istražiti kako bi se stvorilo bogatije i interaktivnije AR iskustvo. Jedna je od takvih komponenti ***AR Face Manager***, koji omogućava detekciju i praćenje ljudskih lica u stvarnom vremenu pomoću kamere mobilnog uređaja. Ta komponenta posebno je korisna za stvaranje interaktivnih iskustava koja se temelje na ljudskom licu, kao što su filtri za lice, animirani avatari ili aplikacije za virtualnu probu proizvoda.
AR *Face Manager* koristi prednju kameru mobilnog uređaja za praćenje lica u vidnom polju, omogućavajući razvojnim inženjerima da na njih prikažu 3D modele ili da pokreću određene funkcije i događaje na temelju izraza lica. Osim toga, prati lica korisnika u stvarnom vremenu i omogućuje prikaz 3D sadržaja „pričvršćenog“ za lice korisnika ili koristiti prepoznavanje izraza lica za pokretanje raznih funkcionalnosti.

> ⚠ **VAŽNO JE ZNATI!**
> Za razvoj AR aplikacija koje uključuju prepoznavanje lica, često je nužno prilagoditi postavke privatnosti i tražiti od korisnika određene dozvole pri pokretanju aplikacije.

U nastavku će biti opisani koraci postavljanja *AR Face Managera*

1. korak – dodati AR *Face Manager* na AR *Session Origin* objekt u *Unity* sceni.
2. korak – postaviti prednju kameru uređaja kao izvor ulaznih podataka za AR da bi se mogle pratiti informacije o licu.
3. korak – kreirati ili uvesti 3D model (npr. masku) koji će se koristiti u AR iskustvu. Skripta bi trebala upravljati logikom praćenja lica i interakcije s 3D modelom.
4. korak – obvezno provjeriti dodatne resurse i dokumentaciju radi iskorištavanja kompleksnijih mogućnosti i alata dostupnih u *Unity* AR *Foundationu*.

Primjer skripte, koji demonstrira kako raditi s događajima vezanim uz praćenje lica:

```
using UnityEngine;
using UnityEngine.XR.ARFoundation;
public class FaceTrackingHandler : MonoBehaviour
{
   [SerializeField] private ARFaceManager arFaceManager;
   [SerializeField] private GameObject facePrefab;

        private void OnEnable()
        {
                arFaceManager.facesChanged += OnFacesChanged;
        }

        private void OnDisable()
        {
                arFaceManager.facesChanged -= OnFacesChanged;
        }

        private void OnFacesChanged(ARFacesChangedEventArgs eventArgs)
        {
                foreach (var face in eventArgs.added)
                {
                        // Logika za nova praćena lica.


                        face.gameObject.AddComponent<YourCustomFaceBehaviour>()
                ;
                }

                foreach (var face in eventArgs.updated)
                {
                        // Logika za ažurirana praćena lica.
                }
```

```
            foreach (var face in eventArgs.removed)
            {
                // Logika za uklonjena praćena lica.
            }
        }
}
```

5.1.13. Prepoznavanje slika u proširenoj stvarnosti (AR)

Prepoznavanje slika u proširenoj stvarnosti (AR) odnosi se na sposobnost AR sustava da identificira i obrađuje slike iz stvarnog svijeta, omogućujući softveru da sidri virtualne objekte na prepoznate slike u okolini korisnika. To je ključni aspekt stvaranja uključivih i interaktivnih AR iskustava. U *Unityjevoj* AR podlozi (*AR Foundation*), sposobnost prepoznavanja i praćenja 2D slika u okolini osigurava se pomoću *AR Tracked Image Managera*. Kada je riječ o prepoznavanju slika u AR-u, sustav koristi kameru kako bi identificirao unaprijed definiranu sliku u korisnikovoj okolini. Nakon što identificira sliku, može prekriti digitalni sadržaj preko nje, stvarajući iskustvo mješovite stvarnosti, gdje se fizički i digitalni svjetovi stapaju.

Za implementaciju prepoznavanja slika koristeći *Unity* AR podlogu, treba početi s dodavanjem komponente AR *Tracked Image Manager* na AR *Session Origin* objekt. Taj *manager* odgovoran je za identifikaciju slika u korisnikovoj okolini i instanciranje odgovarajućih AR *Tracked Image* objekata. AR sustav mora razumjeti koje slike prepoznati. To se postiže stvaranjem knjižnice referentnih slika. Navedeno se može učiniti na način da se:

1. dođe na ***Assets*** > ***Create*** > ***XR*** > ***Reference Image Library***
2. nakon što se stvori knjižnica referentnih slika, dodaju se slike koje se želi da aplikacija prepozna specificirajući teksturu i fizičku veličinu svake slike

Napomena! Potrebno je provjeriti jesu li odabrane slike visokog kontrasta i imaju li dovoljno jedinstvenih detalja da budu lako prepoznatljive.

3. poveže knjižnica referentnih slika s AR *Tracked Image Managerom*. Treba specificirati knjižnicu referentnih slika u svojstvu *Serialized Library* AR *Tracked Image Managera*. Zatim je potrebno odlučiti što bi se trebalo dogoditi jednom

kada se slika prepozna. Na primjer, želi se da se 3D model pojavi na vrhu prepoznate slike. To se postiže tako da:

- se stvori *prefabe* koji bi trebali biti prikazani po prepoznavanju slike
- implementira logika u skriptama da bi se obradili događaji pokrenuti kada se slika prepozna.

Unityjev AR Foundation pruža mogućnosti poput *trackedImagesChanged,* pomoću kojih se može upravljati onim što se događa kada se slike detektiraju, ažuriraju ili izgube.

Primjer skripte za rukovanje prepoznatim slikama:

```
using UnityEngine;
using UnityEngine.XR.ARFoundation;

public class ImageRecognitionHandler : MonoBehaviour
{
    [SerializeField] private ARTrackedImageManager arTrackedImageManager;

    private void OnEnable()
    {
        arTrackedImageManager.trackedImagesChanged +=
OnTrackedImagesChanged;
    }

    private void OnDisable()
    {
        arTrackedImageManager.trackedImagesChanged -=
OnTrackedImagesChanged;
    }

    private void
OnTrackedImagesChanged(ARTrackedImagesChangedEventArgs eventArgs)
    {
        foreach (var trackedImage in eventArgs.added)
        {
            // Logika za novopraćene slike.
```

```
            }

            foreach (var trackedImage in eventArgs.updated)
            {
                // Logika za ažurirane praćene slike.
            }

            foreach (var trackedImage in eventArgs.removed)
            {
                // Logika za uklonjene praćene slike.
            }
        }
}
```

U području razvoja proširene stvarnosti (AR), osim praćenja lica, prepoznavanja slika i detekcije ravnina, nekoliko je drugih komponenti integralno i vrijedno ih je spomenuti.

- **Prepoznavanje objekata** omogućava AR sustavu identificiranje i praćenje stvarnih objekata, poput igračaka, opreme ili raznih predmeta, i preklapanje digitalnog sadržaja na njima. Može se koristiti u raznim primjenama kao što su igre, gdje stvarni objekt postaje interaktivni element igre, ili u industrijskim primjenama za preklapanje uputa na opremu.
- **Lokacijski bazirani AR** koristi geografske podatke o lokaciji (poput GPS-a, akcelerometra i podataka kompasa) za sidrenje digitalnog sadržaja u fizičkim prostorima. *Pokemon Go* izvrstan je primjer, koristi se podatcima o lokaciji za postavljanje virtualnih stvorenja na stvarne lokacije.
- **Procjena svjetlosti** omogućava AR aplikaciji procjenjivanje trenutačnih uvjeta osvjetljenja u okolini. To pomaže razvojnim programerima prilagoditi osvjetljenje virtualnog sadržaja da bi se on bolje uklopio u stvarni svijet, pridonoseći realizmu.
- **Procjena dubine** pruža AR sustavu podatke o dubini i konturama stvarnog svijeta. To je ključno u aplikacijama u kojima virtualni objekti trebaju na uvjerljiv način interagirati s fizičkim okruženjem, poput skrivanja iza stvarnih objekata ili prilagođavanja fizičkim površinama.

Poticajno je razvojnim programerima i entuzijastima unutar AR, VR i MR prostora istraživati sve te aspekte, kako bi dublje shvatili i iskoristili širok spektar mogućnosti koje tehnologije proširene stvarnosti nude.

ZAKLJUČNO O RAZVOJU AR-a

Proširena stvarnost (**AR**) nezaustavljivo se razvija, nudeći jedinstvene i sveprisutne načine istraživanja interakcija između digitalnoga i fizičkog svijeta. Razvojem AR aplikacija uočavamo nepregledne mogućnosti kreiranja interaktivnih i realističnih iskustava, gdje korisnici mogu istraživati, učiti i zabaviti se na nove i uzbudljive načine. Ovladavanje osnovnim i naprednim konceptima AR-a, uključujući praćenje, prepoznavanje oblika, i upravljanje objektima, otvara vrata širokom spektru aplikacija u različitim sektorima, od edukacije do oglašavanja.

Širina i fleksibilnost AR tehnologija omogućuju njihovu primjenu u raznim industrijama i kontekstima. Na primjer, u obrazovanju AR može donijeti revolucionarne metode podučavanja, nudeći vizualno bogate i interaktivne materijale. U kontekstu oglašavanja, AR pruža neusporedivo iskustvo proizvoda i brendova, kreirajući personalizirane prezentacije koje mogu duboko rezonirati s ciljanom publikom.

Tijekom razvoja AR projekata, treba razmotriti integraciju s drugim tehnologijama da bi se obogatilo korisničko iskustvo. Povezivanje AR-a i interneta stvari (IoT) može pružiti korisnicima mogućnost za interakciju i kontrolu pametnih uređaja unutar njihove stvarne okoline. Razvoj miješane stvarnosti (MR), koja kombinira elemente AR-a i VR-a, može korisnicima pružiti još dublja i fascinantnija iskustva.

5.2. Virtualna stvarnost

Virtualna stvarnost (**VR**) predstavlja digitalno iskustvo koje korisnike uvodi u simulirano okruženje, različito od stvarnoga svijeta. Koristi se računalnom tehnologijom za stvaranje umjetnoga okruženja s kojim se korisnici mogu interaktivno baviti na način koji se osjeća stvarno, koliko i fizički svijet oko njih. Ključni element VR-a je omogućavanje korisnicima da postanu dio virtualnog svijeta, nudeći im mogućnost interakcije s virtualnim elementima pomoću različitih uređaja i senzora. Temeljna tehnologija koja stoji iza virtualne stvarnosti uključuje VR naočale, praćenje pokreta,

grafičko renderiranje i stvaranje uvjerljivih i realističnih vizualnih prikaza. **VR naočale** omogućavaju korisniku da bude uronjen u virtualno okruženje, dok **senzori** i **kontrolori** omogućavaju interakciju unutar tog okruženja. Različite tehnologije, kao što su senzori pokreta, haptička povratna informacija i 3D zvuk, surađuju u kreiranju bogatstva iskustava koje korisnik može osjetiti u virtualnom svijetu.

VR je našao primjenu u raznim industrijskim sektorima, zabavi, obrazovanju, zdravstvenoj skrbi, vojsci i mnogim drugima. U videoigrama korisnici mogu uroniti u kompleksne svjetove i interagirati s njima na jedinstvene načine. U medicini VR se koristi za terapije, treninge i simulacije operacija. U arhitekturi i dizajnu stručnjaci koriste VR za vizualizaciju budućih projekata i kreacija u trodimenzionalnom prostoru prije nego što postanu fizička stvarnost.

Iako virtualna stvarnost nudi impresivna iskustva, ona također donosi izazove, poput razvoja sadržaja, tehnoloških ograničenja i problematike povezane s korisničkim iskustvom, kao što su VR bolesti. Budućnost VR-a vjerojatno će donijeti razvoj lakših, ergonomičnijih uređaja, poboljšanja glede realističnosti simulacija, te širenje primjena u različite industrije i sektore. Virtualna stvarnost otvara vrata novim, uzbudljivim svjetovima i iskustvima, izazivajući granice naše stvarnosti i načina na koji je percipiramo i interagiramo s njome. Kako tehnologija nastavlja napredovati, možemo očekivati sve uživljujuće, realističnije i pristupačnije VR iskustvo, koje će oblikovati budućnost mnogih industrija i svakodnevnog života.

5.2.1. Priprema VR naočala

Za potrebe razvoja projekta koji će biti prikazan u nastavku poglavlja, koristit će se ***Oculus Rift S***, jedan od vodećih VR uređaja koji pruža kvalitetno i uronjeno iskustvo virtualne stvarnosti. *Oculus Rift S* predstavlja spoj modernog dizajna, udobnosti i visokokvalitetnih tehničkih specifikacija, što ga čini izvrsnim izborom za razne VR aplikacije.

1. Prvi je korak u postavljanju *Oculus Rift S*-a priprema računala. Važno je da računalo ispunjava minimalne tehničke zahtjeve koje je *Oculus* postavio za *Rift*

S. Oculus na svojoj službenoj stranici pruža detaljne informacije o preporučenim specifikacijama.

2. Nakon što je potvrđeno da računalo ispunjava zahtjeve, treba prijeći na instalaciju softvera. Za to je potrebno posjetiti službenu web stranicu Oculus-a i preuzeti najnoviju verziju softvera za *Oculus Rift S*. Nakon toga treba slijediti upute za instalaciju koje se pojavljuju na ekranu. Navedeni softver omogućit će pristup *Oculus* trgovini, postavkama uređaja i raznim drugim značajkama.

3. Sljedeći je korak povezivanje *Oculus Rift S* uređaja s računalom. Potrebno je povezati USB kabel s USB 3.0 portom na računalu, a zatim *DisplayPort* kabel s odgovarajućim portom. Kada je sve pravilno povezano, softver će prepoznati uređaj.

4. Jedan od najvažnijih koraka u postavljanju VR uređaja konfiguracija je igraćeg prostora. Softver će voditi postupkom označavanja i postavljanja prostora kako bi se osiguralo sigurno i slobodno kretanje unutar VR-a. Uz pomoć kontrolora koji dolaze s uređajem, potrebno je označiti granice prostora prateći upute na ekranu.

5. Nakon što je prostor definiran, potrebno je kalibrirati kontrolere. Zatim uključiti, slijediti upute na ekranu i sinkronizirajti ih s *Oculus Rift S*-om. Završno, potrebno je staviti VR naočale na glavu i prilagoditi ih.

VAŽNO ZAPAMTITI

Da bi se izbjegle potencijalne nesreće, nužno je osigurati prostor bez prepreka.

5.2.2. Postavljanje VR projekta

Na početku procesa potrebno je kreirati novi *Unity* projekt putem *Huba* odabirom opcije ***New*** i dodjeljivanjem imena projektu. Nakon odabira kreiranja novog projekta, treba prijeći na odjeljak „Predložak“ (eng. ***Template***) i izabrati **VR *Core*** iz dostupnih predložaka. Taj predložak dolazi s predinstaliranim postavkama i alatima koji su

optimizirani za razvoj VR aplikacija, uključujući prethodno postavljene postavke kamere i kontrolora te različite elemente sučelja korisnika prilagođene VR-u.

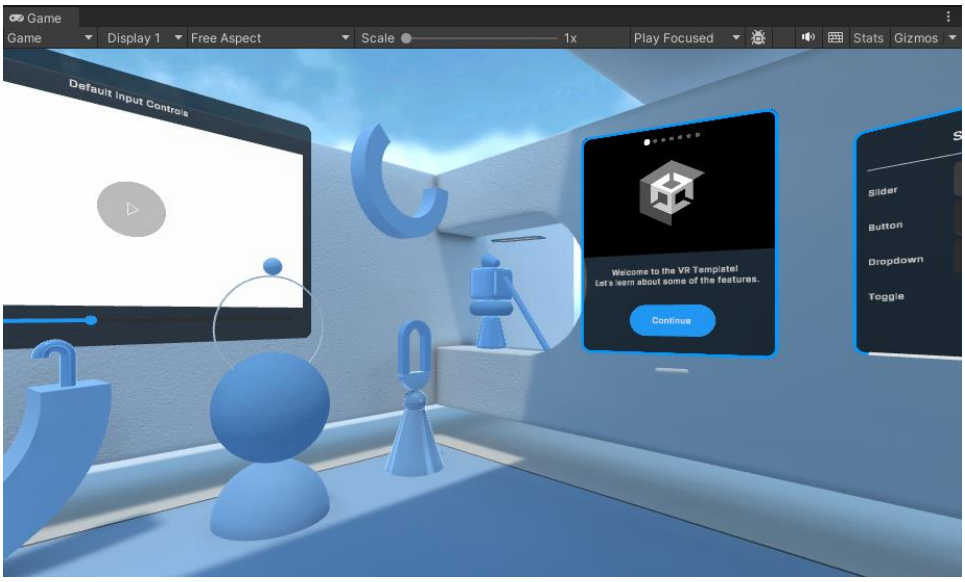
Kada se kreira *Unity* projekt koristeći se VR *Core* predloškom, može se primjetiti da dolazi s prethodno postavljenim uzorkom scene (eng. ***Sample Scene***). Ta scena služi kao početni primjer za brže upoznavanje s osnovama razvoja VR aplikacija unutar *Unity* okruženja. Scena uključuje osnovne objekte i konfiguracije koje demonstriraju osnovne aspekte i mogućnosti VR razvoja. U kontekstu VR aplikacija, mehanizme kontrole kretanja često karakterizira njihova intuitivnost i fluidnost kako bi se omogućilo korisnicima da se lako kreću kroz virtualni prostor. Uobičajeno je u VR aplikacijama:

- **lijevi *stick* / analogna palica:** koristi se za osnovno kretanje korisnika kroz virtualni prostor. Pomakom lijeve analogne palice, korisnik može hodati ili se pomicati kroz okolinu, ovisno o postavkama

- **desni stick / analogna palica:** pored standardnog kretanja, desna analogna palica često se koristi za dodatne načine kretanja ili interakciju s prostorom. U kontekstu VR-a, popularna metoda jest teleportacija.

Teleportacija je čest mehanizam kretanja u VR aplikacijama, osobito koristan u situacijama gdje standardno kretanje može izazvati korisniku nelagodu ili dezorijentaciju. U nastavku je opisan uobičajeni način funkcioniranja.

Korisnik pokreće desnu palicu ili je koristi za odabir destinacije. Dok je drži u željenom smjeru, na tlu se pojavljuje indikator koji prikazuje mjesto na koje će se korisnik teleportirati. Pustivši palicu ili pritiskom dodatnog gumba, korisnik se teleportira na odabranu lokaciju. Ispod vidimo početnu scenu *Unity* VR *Core* predloška.



Slika 61. Primjer scene „VR Corea“ (slika zaslona, Unity)

5.2.3. Lokomocija u virtualnoj stvarnosti (VR)

Lokomocija u VR-u odnosi se na bilo koju metodu koja korisnicima omogućava kretanje virtualnim okruženjem. Efikasna lokomocija ključna je za stvaranje dubokoga i udobnog korisničkog iskustva u VR-u. Ta je tema često pod povećalom jer loše strategije lokomocije mogu izazvati nelagodu ili kinetozu (bolest kretanja) u korisnika.

Uobičajene **tehnike lokomocije** u VR-u

- **Teleportacija**, tehnika instantnog premještanja korisnika iz jedne točke prostora u drugu, često se koristi u svrhu minimiziranja kinetoze. U kontekstu njezine implementacije, posebno je naglašena u aplikacijama poput VR turizma ili arhitektonske vizualizacije. Esencijalna joj je smislenost prostorne navigacije i istovremeno, minimiziranje pojave senzornih konflikata korisnika s virtualnim okruženjem.

- **Hodanje na mjestu** omogućuje korisnicima da svojim stvarnim pokretima hodanja direktno utječu na napredovanje unutar virtualne stvarnosti. Primarni cilj takvog oblika lokomocije leži u pružanju intuitivnoga i prirodnog načina kretanja VR-om, bez zahtjeva za velikim fizičkim prostorom. Tipično, taj bi se način kretanja mogao pronaći u VR aplikacijama koje naglašavaju imerziju i fizičku interakciju, poput određenih VR *fitness* aplikacija.

- **Lokomocija** pomoću palčane palice predstavlja tehniku gdje se korisnici koriste analognom palicom ili dodirnom pločom na svojim VR kontrolorima za kretanje virtualnim prostorom. Ta je metoda često prisutna u VR igrama i iskustvima gdje se fluidna navigacija smatra ključnom, posebice u kontekstima gdje je korisnikov fizički prostor za kretanje ograničen.

- **Kretanje po tračnicama ili lokomocija po tračnicama** uvodi korisnike u iskustvo gdje se kreću duž unaprijed definiranog puta ili „tračnice“, eliminirajući potrebu za aktivnim odlukama o smjeru. Ta je metoda često korištena u narativno vođenim VR iskustvima ili VR vožnjama, pružajući vodič kroz specifične priče ili rute radi usredotočenosti na predviđeni sadržaj.

- **Fizička lokomocija** uključuje direktan prijevod fizičkih pokreta korisnika u pokrete unutar VR svijeta, nudeći snažan osjećaj prisutnosti i imerzije. Taj način kretanja može se tipično vidjeti u VR iskustvima koja teže izrazitom uživljavanju i mogu pružiti siguran kontekst za fizičko kretanje korisnika, kao što su VR *escape* sobe ili specifične simulacije treninga.

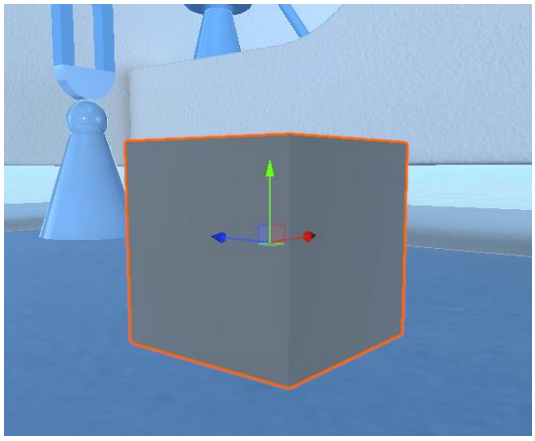
Razumijevanje i odabir prave tehnike lokomocije ima ključnu važnost jer utječe na iskustvo i doživljaj korisnika. Potrebna je ravnoteža između udobnosti korisnika i praktičnosti implementacije. Razni faktori, poput osjetljivosti korisnika na kinetozu (vrtoglavica), dostupnog fizičkog prostora i namjene, moraju se uzeti u obzir da bi se pružilo pristupačno i uključivo VR iskustvo.

5.2.4. Interakcija s objektima u okolini virtualne stvarnosti (VR)

Interakcija s objektima u okolini virtualne stvarnosti (VR) ključna je za stvaranje uživljenih i privlačnih korisničkih iskustava. Koristeći ***XR Toolkit*** unutar *Unityja*, programeri mogu iskoristiti prethodno izrađene komponente poput *XRGrabInteractable* da bi olakšali interakcije s virtualnim objektima pomoću **VR kontrolora.**

***XRGrabInteractable*: pregled**

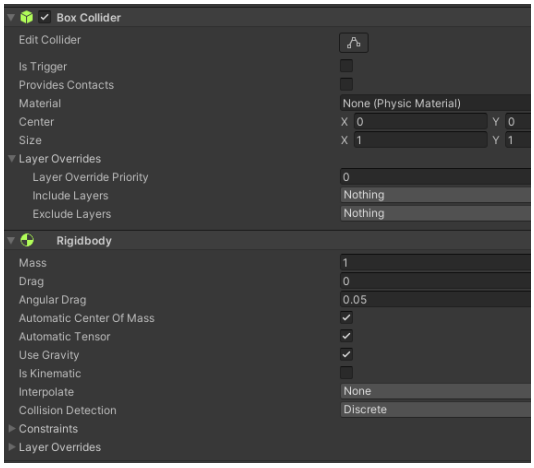
Komponenta ***XRGrabInteractable*** dio je *Unityjeva* XR *Toolkita* i dizajnirana je da omogući objektima biti dohvatljvima i pomičnim unutar VR scene. Jednostavnije rečeno, omogućava objektima interagiranje s VR kontrolorima za ruke, pa korisnici prepoznaju kada je objekt „zgrabljen“, „pomaknut“ ili „otpušten“.



Slika 62. 3D kocka (slika zaslona, Unity)

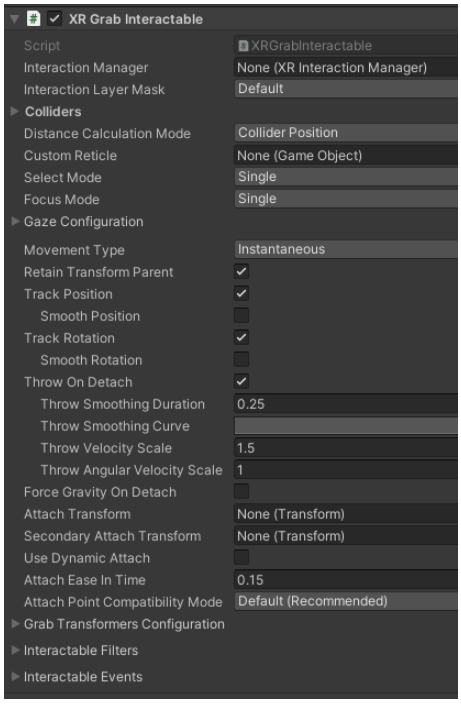
Prvi segment koraka uključuje dodavanje kocke. To je moguće ostvariti unutar *Unityjeve* hijerarhije izbornika tako da se kreira novi 3D objekt, specifično kocku (***GameObject > 3D Object > Cube***). Kocka mora biti opremljena komponentama ***Rigidbody*** i ***Collider*** kako bi interakcija bila moguća i fizički točna. *Collider* omogućuje detekciju interakcija, dok *Rigidbody* dopušta kocki da bude pod utjecajem fizike.

Rigidbody je komponenta dostupna unutar *Unity* razvojnog okruženja koja omogućava *GameObjectu* (objektu unutar igre, odnosno scene) da bude pod utjecajem fizike motora. Kada *GameObject* ima dodanu komponentu *Rigidbody*, on postaje dio fizičke simulacije i automatski će reagirati na gravitaciju, sudare, ili bilo koju drugu interakciju u sceni.



Slika 63. Komponente kocke (slika zaslona, Unity)

S odabranom kockom koristi se *Inspector* za dodavanje nove komponente naziva ***XR Grab Interactable***. Ta će komponenta biti ključna u korisničkoj mogućnosti interagiranja s kockom u VR prostoru. Ako je kocka konfigurirana s komponentom *XR Grab Interactable*, kao što je prethodno objašnjeno, kada se približi i upotrijebi gumb za hvatanje, koji se nalazi sa strane kontrolora, bit će ostvarena interakcija s kockom!



Slika 64. Komponente kocke (slika zaslona, Unity)

XRGrabInteractable osmišljena je da bi omogućila jednostavnu interakciju s objektima u VR i AR iskustvima. Kada se doda *GameObjectu*, oprema objekt sposobnošću da bude uhvaćen i da se njime manipulira pomoću XR kontrolora. U nastavku su izdvojena neka od svojstava dostupnih u komponenti *XRGrabInteractable*.

Sloj maske *(eng. Layer Mask*): određuje koji će se slojevi smatrati važećima za interakciju. Koristi se za ograničavanje sposobnosti interaktora da interagiraju s objektima na određenim slojevima.

Kolideri (eng. ***Colliders*****)**: lista kolidera koji se smatraju dijelom ovog interaktivnog objekta za određivanje važećih interakcija. Ako nisu navedeni kolideri, koristi se bilo kojim koliderima prisutnim u objektu ili njegovoj djeci.

Događaji (eng. ***Events*****)**

- ***On First Hover Entered***: pokreće se prvi put kada interaktor uđe u stanje lebdenja s interaktivnim objektom.
- ***On Hover Entered*****:** pokreće se svaki put kada interaktor uđe u stanje lebdenja s interaktivnim objektom.
- ***On Last Hover Exited*****:** pokreće se posljednji put kada interaktor izađe iz stanja lebdenja s interaktivnim objektom.
- ***On Select Entered*****:** pokreće se kada se interaktivni objekt odabere (zgrabi).
- ***On Select Exited***: pokreće se kada se interaktivni objekt otpusti.

Praćenje pozicije (eng. *Track Position*): određuje hoće li objekt pratiti položaj ruke/kontrolora kada je zgrabljen.
Praćenje rotacije (eng. *Track Rotation*): određuje hoće li objekt pratiti rotaciju ruke/kontrolora kada je zgrabljen.

Vrijeme ubacivanja prilikom pričvršćivanja (eng. *Attach Ease In Time*): vrijeme koje je potrebno interaktivnom objektu da dođe na mjesto kada je zgrabljen. Vrijednost 0 čini da objekt trenutno prijeđe na ruku ili kontrolor, dok više vrijednosti stvaraju glađi prijelaz.

Bacanje pri otpuštanju (eng. *Throw on Detach*): ukazuje hoće li se objekt baciti kada se otpusti.
Skala brzine bacanja (eng. *Throw Velocity Scale*): modificira brzinu objekta kada se baci. Vrijednost veća od 1 povećat će brzinu bacanja, dok će vrijednost manja od 1 smanjiti.

Gravitacija (eng. *Gravity*): ukazuje hoće li interaktivni objekt biti pod utjecajem gravitacije kada se otpusti.
Održavanje brzine (eng. *Keep Velocity*): određuje hoće li interaktivni objekt održavati svoju brzinu kada se otpusti.

Razumijevanje i manipuliranje tim svojstvima važno je za oblikovanje iskustva interakcije u vašem XR projektu. Mogu se prilagoditi različitim scenarijima interakcije i osigurati da manipulacija objektom u vašem virtualnom okruženju bude intuitivna.

5.2.5. Stvaranje materijala

Materijali u *Unityju* koriste se za definiranje izgleda površine objekta. Oni određuju kako se svjetlost odbija ili propušta kroz površinu, čime se definira boja, sjaj, transparentnost i drugi vizualni aspekti objekta. Evo kako je moguće stvoriti materijal u *Unityju*:

- proces se započinje otvaranjemprozora ***Project*** u *Unity* sučelju. To je mjesto gdje se čuvaju svi resursi, uključujući skripte, teksture, scene i, naravno, materijale

- desnim klikom unutar mape se određuje mjesto spremanja materijala, a zatim treba otvoriti kontekstni izbornik. Zatim treba otići na ***Create"*** > ***"Material***. Pojavit će se novi materijal u direktoriju resursa

- kada se stvori novi materijal, potrebno mu je pripisati ime. Preporučuje se da ga se odmah preimenuje u nešto što opisuje njegovu namjenu ili izgled da bi se lakše upravljalo resursima kasnije. U ovom primjeru nazvat ćemo ga *Grabbed Material*

- odabranim novostvorenim materijalom, na desnoj strani u prozoru „**Inspektor**“ (eng. *Inspector*), mogu se vidjeti razne opcije za konfiguriranje. Tu se može postaviti boju, sjaj, odabrati teksturu, postaviti transparentnost i mnoge druge postavke. Preporučljivo je eksperimentirati s tim opcijama radi postizanja željenog izgleda.

5.2.6. Skriptiranje VR interakcija

U nastavku će biti obrađen proces kodiranja interakcije uz upotrebu komponente ***XRGrabInteractable*** unutar *Unityja*. Ta komponenta pruža temeljni skup alata za kreiranje interaktivnih objekata koje korisnici mogu lako uhvatiti i njima manipulirati u VR prostoru. No, što ako se želi dodati složenije reakcije objekata na interakcije? Na primjer, mijenjanje izgleda objekta kad ga korisnik uhvati. U nastavku će biti dan primjer koda koji demonstrira upravo to, mijenjajući materijal objekta kada je zgrabljen.

```
using UnityEngine;
using UnityEngine.XR.Interaction.Toolkit;

public class CubeInteract : MonoBehaviour
{
    private XRGrabInteractable grabInteractable; // Komponenta koja omogućuje interakciju
 hvatanja.
    public Material grabbedMaterial; // Materijal koji će se primijeniti kada je kocka
 zgrabljena.
    private Material originalMaterial; // Izvorni materijal kocke radi mogućnosti povrata.
```

```
    private Renderer cubeRenderer; // Render kocke radi mogućnosti promjene materijala.
    private void Awake()
    {
        grabInteractable = GetComponent<XRGrabInteractable>();
        cubeRenderer = GetComponent<Renderer>(); // Dohvaćanje komponente rendera.
        originalMaterial = cubeRenderer.material; // Pohrana izvornog materijala.
    }
    private void OnEnable()
    {
        // Dodavanje slušatelja za događaje hvatanja i puštanja.
        grabInteractable.selectEntered.AddListener(ChangeMaterialOnGrab);
        grabInteractable.selectExited.AddListener(RevertMaterialOnRelease);
    }
    private void OnDisable()
    {
        // Uklanjanje listenera kad skripta nije aktivna.

    grabInteractable.selectEntered.RemoveListener(ChangeMaterialOnGrab);

    grabInteractable.selectExited.RemoveListener(RevertMaterialOnRelease);
    }
    // Metoda koja mijenja materijal kocke kad je zgrabljena.
    private void ChangeMaterialOnGrab(SelectEnterEventArgs args)
    {
        cubeRenderer.material = grabbedMaterial;
    }
    // Metoda koja vraća izvorni materijal kocki kad je puštena.
    private void RevertMaterialOnRelease(SelectExitEventArgs args)
    {
        cubeRenderer.material = originalMaterial;
    }
```

}

U tom primjeru koda kreirana je jednostavna *Unity* skripta koja omogućuje korisnicima da vide vizualnu povratnu informaciju kad zgrabe objekt, mijenjajući materijal objekta dok je zgrabljen, i vraćajući ga natrag kad je objekt pušten.

Sada je potrebno aplicirati skriptu na objekt *Cube*, i u polje *grabbedMaterial* povući materijal izrađen u prethodnom poglavlju. Zatim treba stisnuti *Play* i zgrabiti kocku!

5.2.7. Unity događaji

U prethodnom primjeru korišten je oblikovni obrazac promatrača kako bi se registrirali za događaj. Međutim, može se primjetiti da su ti događaji izloženi u „Inspektoru“ kao *select* događaji u *XR Grab Interactable*, primjerice *Select Entered*. Taj obrazac omogućuje određenom objektu (u ovom slučaju skripti *CubeInteract*) da „sluša“ ili promatra za specifične događaje (u ovom slučaju događaje *selectEntered* i *selectExited* unutar komponente *XRGrabInteractable*) i izvršava specifične funkcije kada se ti oni dogode (kao što su *ChangeMaterialOnGrab* i *RevertMaterialOnRelease*). No, postoji i elegantniji način za postizanje sličnog rezultata unutar *Unityja*, koji može biti osobito koristan u situacijama gdje je kodiranje minimalno ili gdje se želi pružiti dizajnerima ili onima koji nisu programeri, fleksibilnost za lako definiranje ponašanja unutar *Unity* sučelja. Navedeno se može postići *Unity* događajima.

***Unity* događaji** (eng. *Unity Event*) omogućuju definiranje događaja u *Unity* sučelju i dodjeljivanje funkcija koje treba izvršiti kada se događaj aktivira, sve to bez potrebe za dodatnim kodiranjem. To ne pojednostavljuje samo proces postavljanja osnovnih interakcija, već također omogućava veću suradnju između programera i dizajnera, smanjujući potrebu izmjena koda za svaku malu prilagodbu.

Prvi korak u pojednostavljenju interakcija koristeći *Unity* događaje jest modificiranje skripte koja je prikazana ranije. Ta skripta ***CubeInteract*** i dalje sadrži metode koje mijenjaju materijal kocke. Sada, umjesto slušanja i reagiranja na događaje unutar skripte, te će metode biti pozivane direktno posredstvom *Unity* sučelja pomoću *Unity* događaja. Vrlo je važno osigurati da funkcije *ChangeMaterialOnGrab* i

RevertMaterialOnRelease u skripti budu označene kao *public*. To je ključno jer ako su funkcije postavljene kao privatne (eng. *private*) ili nisu eksplicitno označene, predefinirano će biti privatne. One neće biti vidljive i neće ih se moći povezati s *Unity* događajima.

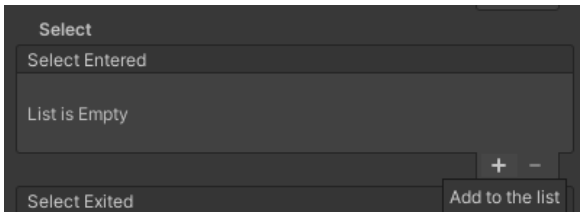
```
using UnityEngine;
public class CubeInteract : MonoBehaviour
{
    public Material grabbedMaterial; // Materijal koji će se primijeniti kada je kocka zgrabljena.
    private Material originalMaterial; // Izvorni materijal kocke, kako bismo ga mogli vratiti.
    private Renderer cubeRenderer; // Render kocke, kako bismo mogli mijenjati materijal.

    private void Awake()
    {
        // Inicijalizacija varijabli.
        cubeRenderer = GetComponent<Renderer>(); // Dohvaćanje komponente rendera.
        originalMaterial = cubeRenderer.material; // Pohrana izvornog materijala.
    }

    // Metoda koja mijenja materijal kocke kad je zgrabljena.
    public void ChangeMaterialOnGrab()
    {
        cubeRenderer.material = grabbedMaterial;
    }
    // Metoda koja vraća izvorni materijal kocki kad je puštena.
    public void RevertMaterialOnRelease()
    {
        cubeRenderer.material = originalMaterial;
    }
}
```

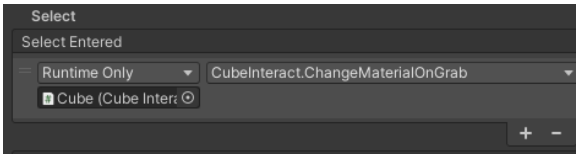
Da bi se omogućila interakcija i postavljanje materijala na kocku prilikom njezina grabljenja i otpuštanja, potrebno je napraviti nekoliko koraka u *Unity* sučelju.

Prvo, potrebno je odabrati objekt kocke unutar *Unity* hijerarhije. To se može učiniti klikom na objekt unutar prozora ***Hierarchy***. U prozoru ***Inspector*** na desnoj strani ekrana trebala bi se vidjeti komponenta *XR Grab Interactable* (ako je prethodno dodana na kocku).



Slika 65. Polja događaja (slika zaslona, Unity)

Unutar komponente ***XR Grab Interactable*** treba pronaći dijelove nazvane ***On Select Entered*** i ***On Select Exited***. Ti događaji odgovaraju trenutcima kada je objekt zgrabljen i kada je otpušten. Zatim treba kliknuti na simbol plus (+). To će dodati novi redak u koji se može dodijeliti funkciju koja će biti pozvana kada se događaj dogodi.



Slika 66. Postavke događaja (slika zaslona, Unity)

U novododanom retku može se vidjeti polje koje omogućava dodjelu objekta koji će reagirati na događaj. Tamo treba povući kocku ili objekt na kojem se nalazi skripta ***CubeInteract***. Kada se kocka doda, pojavit će se drugi padajući izbornik, gdje je potrebno odabrati koja će se funkcija pokrenuti. Nakon toga treba kliknuti na padajući izbornik, pronaći *CubeInteract* ili drugi naziv ovisno o tome kako je skripta nazvana. Zatim treba odabrati *ChangeMaterialOnGrab* za *On Select Entered*, te *RevertMaterialOnRelease* za *On Select Exited*. Sada, kada se scena pokrene, trebalo bi biti moguće zgrabiti kocku te vidjeti kako se mijenja materijal kada se kocka zgrabite kako se vraća na original kad ju otpustimo.

5.2.8. XR utičnice

Ponašanja, poput omogućavanja objektima da se prikvače za određene točke u sceni, vrlo su popularna u VR aplikacijama i igrama. Takva funkcionalnost može poboljšati korisničko iskustvo i uvesti dodatnu razinu interaktivnosti. Primjerice, omogućavanje korisnicima da oruđe ili objekte umetnu ili zakvače na određeni nosač ili nosač alata, može stvoriti imerzivniji i uređeniji virtualni prostor. U kontekstu *Unityja* i *XR Interaction Toolkita*, komponenta ***XR Socket Interactable*** omogućuje razvojnim programerima implementaciju ponašanja toga kvačenja ili „uštekavanja“.

U nastavku je opisan način korištenja komponentom ***XR Socket Interactable***.

Na početku procesa potrebno je odabrati ili stvoriti objekt koji se želi koristiti kao točka na koju će se objekt kvačiti. Taj objekt može biti, na primjer, nosač alata ili bilo koja vrsta *docking* stanice. U ovom je slučaju to obična kugla za potrebe primjera interakcije. S objektom odabranim u hijerarhiji, potrebno je otići u prozor „Inspektor“ i dodati komponentu *XR Socket Interactable*. To će objektu dodati funkcionalnosti potrebne za interakciju s XR objektima.

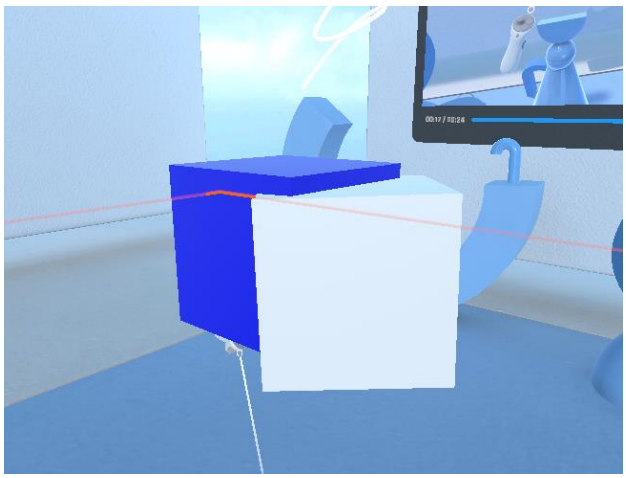
Unutar komponente *XR Socket Interactable* mogu se pronaći razne opcije koje omogućuju prilagođavanje ponašanja kvačenja. To uključuje mogućnost odabira koje se objekte može priključiti, kako se objekt poravnava kad je priključen, te koje događaje pokrenuti kada se objekt priključi ili isključi. Nakon toga treba doći do hijerarhije u *Unity* uređivaču i odabrati objekt s kojim se želi interagirati. U ovom slučaju radi se o kocki. U prozoru „Inspektor“ treba pronaći komponentu *XR Grab Interactable*, koja bi trebala biti dodana objektu. Unutar komponente *XR Grab Interactable* može se pronaći opcija *Movement Type* te, koristeći se padajućim izbornikom, treba odabrati opciju *Kinematic*.

Postavljanjem *Movement Type* na *Kinematic*, *Unityju* se daje informacija da objekt neće reagirati na fizikalne sile poput gravitacije ili sudara dok je zgrabljen. Umjesto toga, objekt će slijediti korisnikove ulazne podatke, odnosno kretanje kontrolora precizno, bez utjecaja fizikalnih simulacija. To može biti posebno korisno za

kontrolirane interakcije ili kada se želi izbjeći neočekivano ponašanje objekta prilikom manipulacije.

Vrsta kretanja *Kinematic* često je odabrana opcija u VR iskustvima, gdje je precizna i predvidljiva kontrola objekata ključna, poput situacija u kojima korisnici moraju pažljivo pozicionirati objekte ili se koristiti alatima na specifičan način.

Kad su *XR Socket Interactable* i *XR Grab Interactable* konfigurirani, i kada korisnici uzmu kocku, moći će vidjeti njezinu interakciju s utičnicom (eng. *socket*) u VR prostoru.



Slika 67. Interakcija XR utičnice (slika zaslona, Unity)

Kada korisnici uzmu kocku i približe je utičnici, primijetit će plavi obrub koji predstavlja predviđeni položaj i orijentaciju kocke ako je otpuste u blizini utičnice. Ta vrsta vizualnoga povratnog signala, često nazvana *ghost* ili *preview* objekt, daje korisnicima jasno razumijevanje kamo i kako će kocka biti pozicionirana unutar utičnice kada je otpuste. Plavi obrub je ugrađeni vizualni pokazatelj koji *Unity* XR pruža kao pomoć korisnicima da shvate gdje će objekt biti smješten kada ga postave u utičnicu.

Navedena značajka osobito je korisna u situacijama gdje preciznost postavljanja objekta ima ključnu ulogu u interakciji, kao što je slaganje predmeta, rješavanje slagalica ili montaža komponenti. Omogućuje korisnicima vizualno predviđanje ishoda njihove akcije prije nego što je dovrše, dodajući tako razinu preciznosti i kontrole koja može biti ključna za određena VR iskustva. Ti su primjeri samo mali uvid u mnogostruke načine na koje možete interagirati s objektima u VR-u pomoću *Unitya*. *XR Interactable toolkit* pruža obilje mogućnosti koje možete istražiti i integrirati u vlastite projekte, primjerice, spojeve, ljuljanje i razne fizikalne interakcije.

Zadatak za samostalni rad

Potrebno je istražiti kako kreirati kompleksne mehanizme poput katapulta ili lifta, različite vrste interaktivnih igračaka, alata ili, čak, interaktivnih umjetničkih instalacija. Svijet virtualne stvarnosti pruža jedinstveno igralište na kojem se može eksperimentirati s fizikom, prostorom i interakcijom na načine koji u stvarnom svijetu nisu mogući.

5.2.9. Sučelja korisnika (eng. User Interface, UI)

Razumijevanje sučelja korisnika (eng. *User Interface*, **UI**) *Unity* razvojnog okruženja ključno je za kreiranje intuitivnih i ugodnih iskustava igrača ili korisnika. *Unity* nudi moćan i prilagodljiv sustav upravljanja UI elementima, koji omogućuje kreiranje različitih sučelja, od tradicionalnih 2D do kompleksnih 3D sučelja i VR ili AR interaktivnih elemenata. To se odnosi na prostor gdje se odvijaju interakcije između ljudi i strojeva. Cilj te interakcije jest učinkovit rad i kontrola stroja s ljudske strane, dok stroj istodobno pruža povratne informacije koje pomažu procesu donošenja odluka operatora.

Osnovni elementi za izgradnju UI-a u *Unityju*

- **Platno (eng. *Canvas*):** svaki UI element u *Unityju* mora biti dio platna (eng. *Canvas*). Platno je kontejner koji drži sve UI elemente i kontrolira kako će se renderirati na ekranu ili u svijetu igre.
- ***Rect Transform*:** za razliku od obične komponente *Transform*, UI elementi koriste *Rect Transform,* koji omogućuje postavljanje pozicije i veličine unutar platna koristeći sidrišta i okretne točke.
- **UI komponente:** *Unity* pruža razne UI komponente poput *Button*, *Text*, *Image*, *Slider* i mnoge druge kojima se može koristiti i kombinirati ih za kompleksna sučelja.
- **Event systems:** *Unityjev* UI sustav koristi sustav događaja (*event system*) za praćenje interakcija poput klika, držanja, povlačenja i sličnih, omogućujući reagiranje na korisničke radnje.

Ako se u hijerarhiji odabere*Spatial Panel Scroll*, može se primjetiti konfiguracija platna (eng. *Canvas*), što je osnovni element svih UI elemenata u *Unityju*. Platno u *Unityju* služi kao površina na kojoj se renderiraju UI elementi i vrlo je važno razumjeti kako se različiti načini renderiranja (eng. *Render Modes*) platna koriste za postizanje željenoga ponašanja UI-a, posebno u kontekstu VR/AR ili 3D svjetova. *Unity* omogućava nekoliko načina renderiranja platna, svaki sa svojom svrhom i uporabom u određenim scenarijima, osiguravajući mogućnost optimizacije i prilagodbe iskustva korisnika vaših aplikacija i igara.

Načini renderiranja platna (eng. *Canvas Render Modes*):

- ***World Space*:** kada je platno postavljeno u način *World Space*, postaje dio 3D svijeta i interagira s ostalim 3D objektima. To platno može biti pogođeno svjetlom i sjenama te se može prikazivati i mijenjati u odnosu na poziciju kamere ili korisnika. Ta se opcija često koristi u VR/AR aplikacijama i 3D igrama kada se želi postići da se UI prikazuje u 3D svijetu, npr. kao holografski zaslon ili interaktivni panel.
- ***Screen Space – Overlay*:** u tom načinu UI elementi renderiraju se na vrhu svih 3D objekata u sceni, neposredno na ekranu, neovisno o poziciji kamere ili svjetlu. Ta je opcija korisna za kreiranje tradicionalnih 2D sučelja, kao što su izbornici, HUD (*Head-Up Display*) elementi ili bilo koji drugi elementi koji uvijek trebaju biti vidljivi.
- ***Screen Space – Camera*:** taj način renderiranja također prikazuje UI elemente na ekranu, ali uzima u obzir perspektivu kamere. To znači da UI elementi mogu interagirati s 3D objektima i mogu se prikazivati ili skrivati ovisno o poziciji i rotaciji kamere, što ih čini korisnim za integraciju 3D efekata ili prikazivanje UI-a u specifičnim dijelovima ekrana.

Svaki od tih načina renderiranja platna ima svoje uporabe i prednosti ovisno o kontekstu i potrebama projekta. Na primjer, u scenarijima proširene stvarnosti (AR) i virtualne stvarnosti (VR) platno *World Space* često je poželjan izbor zbog njegove sposobnosti integriranja s trodimenzionalnim prostorom i interakcijama. Suprotno tome, 2D igre i aplikacije koje ne zahtijevaju 3D interakcije, obično koriste *Screen Space* načine renderiranja za stvaranje UI elemenata. Razmatranje o tome kako i gdje

želite da korisnici vide i interagiraju s UI elementima, ključno je za odabir odgovarajućeg načina renderiranja platna.

5.2.10. Slika (eng. Image)

Komponenta ***Image*** jedan je od osnovnih UI elemenata u *Unityju* i koristi se za prikaz slika na platnu (eng. *Canvas*). Njezina funkcionalnost, posebno u kontekstu platna, ključna je za razvoj korisničkog sučelja u projektnim scenama. Ako se odabere objekt ***Background*** u hijerarhiji, koji se nalazi ispod platna (eng. *Canvas*), može se vidjeti komponentu *Image* u prozoru inspektora. Komponenta *Image* omogućuje prikaz 2D slike unutar UI-a, bilo da se radi o pozadini, ikonama, dugmadi, ili bilo kojim drugim grafikama koje se žele implementirati u korisničkom sučelju.

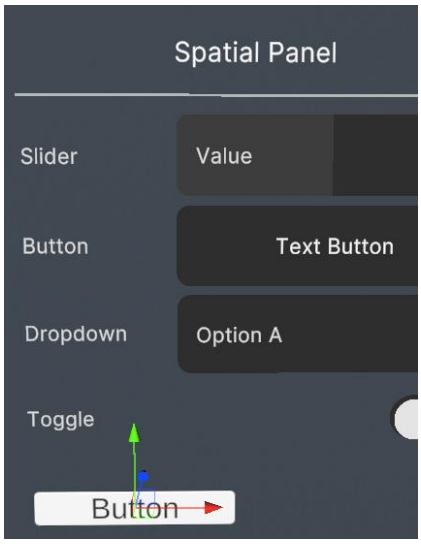
Kako komponenta *Image* funkcionira u odnosu na platno (eng. *canvas*) bit će objašnjeno u nastavku.

- **Povezanost s platnom:** svaka komponenta *Image* mora biti na neki način povezana s platnom da bi se mogla renderirati na ekranu. *Image* će obično biti dijete platna ili nekog drugog UI elementa koji je već dijete platna. Platno upravlja renderiranjem i postavljanjem svih svojih dječjih UI objekata.
- **Svojstva slike:** u prozoru „Inspektora“, dok je odabrana komponenta *Image*, mogu se vidjeti različite opcije koje se mogu prilagoditi. One mogu uključivati izvor slike (*Sprite*), boju, način na koji se slika skalira ili oblikuje unutar definiranih granica, i mnoge druge opcije koje omogućuju širok spektar vizualnih prilagodbi.
- ***Raycast Target*** je svojstvo koje se nalazi unutar komponente *Image* i nekih drugih UI komponenti (primjerice, gumba) u *Unity* sučelju, a odnosi se na sposobnost objekta da primi *raycast* događaje.

Kada je opcija *Raycast Target* označena, a što je predefinirano, objekt će s komponentom *Image* reagirati na događaje poput klikova mišem, dodira na ekranu i slično. Taj objekt postaje dio provjere sudara (eng. *collision check*) kada korisnik interagira s UI-jem. Ukoliko se želi postići da slika ili neki drugi UI element reagira na korisničke interakcije, poput klikova ili dodira, *Raycast Target* treba biti omogućen.

5.2.11. Stvaranje gumba

U nastavku poglavlja bit će detaljno objašnjeno kako se može stvoriti UI gumb u *Unityju* i iskoristiti prethodna znanja o *Unity* događajima (eng. *Unity Events*) da bi mu dodali funkcionalnost.



Slika 68. UI Gumb (slika zaslona, Unity)

Proces počinje stvaranjem gumba. Desnim klikom na ***Spatial Panel Scroll*** u hijerarhiji otvorit će se kontekstualni izbornik. Unutar tog izbornika potrebno je odabrati opciju za stvaranje ***Text Mesh Pro*** gumba. *Text Mesh Pro* je napredni sustav za rad s tekstom, koji omogućuje upotrebu visokokvalitetnih fontova i različitih stilova i postavki za prikaz teksta. Kada se koristi *Text Mesh Pro* za stvaranje UI gumba, omogućuje veću kontrolu i prilagodljivost u usporedbi sa standardnim UI tekstom i gumbima.

Navigacijom do objekata *Left Controller* ili *Right Controller* i pronalaženjem *Ray Interactor* u njihovoj hijerarhiji, moguće je primjetiti komponentu ***XR Ray Interactor*** na tim objektima. Komponenta *XR Ray Interactor* u *Unity* XR *Toolkitu* ima ključnu ulogu u omogućavanju korisnicima da interagiraju s virtualnim objektima i sučeljem koristeći pokazivače, poput laserskih pokazivača, u svojim VR iskustvima.

Komponenta ***XR Ray Interactor*** omogućava objektu na koji je dodana mogućnost „bacanja“ zrake (eng. *raycasting*) u prostoru scene detekciju objekata s kojima korisnik može interagirati. Ta zraka može „udariti“ ili detektirati druge objekte u sceni koji imaju komponentu *Collider* i, ako je potrebno, postavljeni su kao interaktivni.

U kontekstu UI gumba i generalno UI elemenata, *XR Ray Interactor* ima ključnu ulogu u detekciji interakcija s takvim elementima. Na primjer, kada korisnik uperi svoj VR kontrolor prema UI gumbu, zraka emitirana iz kontrolora (zahvaljujući komponenti *XR Ray Interactor*) „pogađa“ gumb, a sustav prepoznaje tu interakciju. Unutar komponente *XR Ray Interactor* različite postavke omogućuju prilagodbu načina i onoga što ta zraka može detektirati te način korisnikova interagiranja s objektima. *Hit Detection Type*

omogućava odabir algoritma detekcije, *End Stop* kontrolira hoće li se zraka zaustaviti kada detektira pogodni objekt, *Max Raycast Distance* određuje maksimalnu duljinu zrake, a *Reference Space* omogućava postavljanje prostora u kojem se zraka provjerava, među mnogim drugim postavkama. Uz to, sekcija *Events* unutar komponente *XR Ray Interactor* omogućava postavljanje različitih odgovora na događaje poput *On Hover Enter*, *On Hover Exit*, *On Select Enter* i *On Select Exit*, što omogućuje dodavanje interakcija ili povratne informacije korisniku prilikom interakcija s UI elementima i objektima u VR okolini.

Kombiniranjem svega toga, moguće je stvoriti intuitivna i responzivna korisnička sučelja i interakcije unutar XR aplikacija, nudeći korisnicima jasne i učinkovite metode komunikacije i navigacije kroz virtualni prostor i funkcionalnosti aplikacije.

5.2.12. Podešavanje postavki izgradnje

Kada je riječ o izgradnji VR aplikacija za stolna računala, postupak može biti jednostavniji u usporedbi s AR razvojem, posebice zbog načina distribucije i implementacije aplikacija. Naime, za stolne VR aplikacije moguće je izgraditi izvršnu datoteku (.exe) koja služi kao samostalna aplikacija, omogućujući korisnicima laganu instalaciju i pokretanje aplikacije direktno s njihovih računala. To uklanja potrebu za prolazak trgovinama aplikacija ili platformama distribucije, pružajući razvojnim timovima i pojedincima veću slobodu u distribuciji njihovih aplikacija krajnjim korisnicima. Za izgradnju VR i stolnih računalnih aplikacija u *Unityju* potrebno je pratiti sljedeće korake:

- otvoriti **File" > "Build Settings**
- dodati sve scene koje se žele uključiti u finalnu igru
- odabrati ciljanu platformu (npr. *Windows*). Provjeriti je li odabrana opcija „**64-bit**“
- kliknuti *Build And Run* unutar *Build Settings*
- odabrati mjesto na računalu za pohranu izgrađenog projekta.

Nakon što je izgradnja završena, igra će se automatski pokrenuti u VR načinu rada. Nakon tih koraka, radno područje VR igre trebalo bi biti spremno za igranje. VR razvoj

može biti složen i postoji mnogo dodatnih specifičnosti ovisno o vrsti igre koju se razvija i platformi na koju se cilja. Preporučuje se upoznavanje s dodatnim resursima i dokumentacijom za svaku specifično korištenu VR platformu. Kada se izgradi projekt u *Unityju*, on stvara izvršnu datoteku često nazivanu „izgradnjom“ ili *buildom*, a prilagođena je platformi za koju se izvozi. Izvršna je datoteka kompajlirana verzija projekta koja se može pokretati kao samostalan program i nije ovisna o *Unity* uređivaču. Za kraj je potrebno pritisnuti dvaput na *.exe file* da bi se pokrenula VR aplikacija.

5.3. Razvoj u mješovitoj stvarnosti (MR) i njegova primjena u Unityju

Mješovita stvarnost, kao spoj proširenje stvarnosti (AR) i virtualne stvarnosti (VR), predstavlja jedan od najdinamičnijih i najinovativnijih pravaca u sektoru alternativne stvarnosti (XR). Uspješan razvoj MR-a zahtijeva kombinaciju tehnologija koje omogućuju precizno praćenje i interakciju stvarnoga i digitalnog svijeta. MR tehnologija obično zahtijeva sofisticiraniju i skuplju opremu u odnosu na AR i VR. Dok AR može funkcionirati na širokom spektru uređaja, uključujući pametne telefone, a VR na raznim VR naočalama, MR obično traži složenije sustave koji mogu obraditi i integrirati podatke iz fizičkoga i virtualnog svijeta u realnom vremenu. Dok elementi u AR aplikacijama ostaju na jednom mjestu, vezani uz stvarni svijet, MR aplikacije mijenjaju sadržaj ovisno o tome kako se okolina stvarnoga svijeta mijenja.

Razvoj mješovite stvarnosti u *Unityju*

Unity je postao jedan od ključnih alata u sektoru proširene stvarnosti, a pruža podršku za razvoj aplikacija mješovite realnosti (MR). Hardver za MR uključuje niz uređaja, kao što su *Microsoftov* ***HoloLens*** , ***Magic Leap*** i druge MR naočale, koje kombiniraju senzore, kamere i holografske tehnologije da bi omogućile visokokvalitetno iskustvo. Oni se često koriste tehnologijama kao što su *spatial computing* i AI za interpretaciju i interakciju s fizičkim i digitalnim svijetom. Razvoj MR hardvera u stalnom je porastu, nudeći sve bolje performanse, lakšu uporabu i realističnije iskustvo korisnicima. Zahvaljujući intuitivnom grafičkom sučelju, bogatoj biblioteci resursa i skriptnih alata, *Unity* omogućava programerima kreiranje interaktivnih i dubokih MR iskustava.

Unityjeva trgovina imovinom također pruža gotove alate i dodatke koji olakšavaju razvoj MR aplikacija.

MR kao rijedak oblik XR-a

Iako se tehnologija brzo razvija, MR je i dalje smatran rijetkim i ne tako raširenim oblikom XR-a. Postoji nekoliko razloga za to:

- **složenost tehnologije**: kombiniranje AR-a i VR-a u koherentno iskustvo MR-a zahtijeva sofisticirane tehnologije praćenja, obrade slike i grafičkoga renderiranja. To stvara tehničke izazove koji nisu prisutni u čistim AR ili VR okruženjima
- **visoka cijena hardvera**: trenutno vodeći uređaji za MR, poput *Microsoftova HoloLensa* ili *Magic Leap Onea*, imaju visoku cijenu. Visoki troškovi proizvodnje, složeni senzori i potrebna računalna snaga čine tu tehnologiju financijski nedostupnom većini potrošača
- **malo tržište**: MR je još uvijek u usponu, s ograničenom količinom sadržaja i aplikacija dostupnih korisnicima. Iako potencijal MR-a postaje sve očitiji, potrebno je više vremena da tehnologija postane dostupna.

5.3.1. Potencijal i budućnost MR-a

Dok se MR trenutno suočava s brojnim izazovima, njegov je potencijal u industriji XR-a nemjerljiv. Očekuje se da će, s padom troškova hardvera i napretkom tehnologije, MR postati sastavni dio svakodnevnog života, baš kao što su to danas pametni telefoni. Njegovi potencijali očituju se, između ostalog, u:

1. **primjeni u različitim sektorima:** osim zabavnog sadržaja, MR nalazi primjenu u različitim industrijskim sektorima. U medicini MR može omogućiti kirurzima proširene vizualne podatke tijekom operacija. U arhitekturi i građevinarstvu MR može olakšati vizualizaciju i planiranje građevinskih projekata u stvarnom vremenu

2. **edukaciji i obuci:** MR ima ogroman potencijal u području obrazovanja. Učenicima i studentima može pružiti uživljena iskustva, olakšavajući razumijevanje kompleksnih tema kroz interaktivnu vizualizaciju
3. **interakciji i socijalnoj sferi:** razvojem MR tehnologije očekuju se i nove vrste društvenih iskustava, gdje će ljudi moći dijeliti digitalne informacije i objekte u stvarnom svijetu, dodajući novu dimenziju komunikaciji.

Jasno je da će razvojni alati i SDK-ovi igrati ključnu ulogu u oblikovanju MR-ove buduće putanje. Ti alati ne samo da omogućavaju izradu aplikacija i iskustava koja smo već vidjeli, već i pružaju platformu za istraživanje nepoznatih područja mješovite stvarnosti. U nastavku su istaknuti neki od alata kojima se može koristiti u svrhu razvoja MR aplikacija.

Razvojni Alati i SDK-ovi:

Windows Mixed Reality SDK: omogućuje razvoj za *HoloLens* i *Windows MR headsetove*, pružajući alate za praćenje pogleda, geste, i interakcije s hologramima
***Magic Leap SDK*:** pruža alate i resurse za razvoj aplikacija koje se koriste jedinstvenim značajkama *Magic Leap* uređaja, uključujući *spatial computing* i okulografsko praćenje.
ARKit* i *ARCore: iako su primarno usredotočeni na AR, *ARKit* (*Apple*) i *ARCore* (*Google*), također pružaju osnove koje se mogu koristiti za kreiranje MR iskustava na mobilnim uređajima.

Iako je MR trenutno u svom začetku, njegova sposobnost kombiniranja najboljih aspekata AR-a i VR-a postavlja ga kao budućeg lidera u XR industriji. Kontinuiranim ulaganjima i istraživanjima, zajedno sa širim korisničkim prihvaćanjem, MR može biti most između digitalnoga i stvarnog svijeta, pružajući neusporediva uživljena iskustva. U *Unityju* razvojni inženjeri imaju platformu koja je spremna za budućnost, pružajući alate i resurse koji će omogućiti inovacije u MR-u.

Zaključno o alternativnim stvarnostima

Polazeći na put sferama proširene stvarnosti (AR), virtualne stvarnosti (VR) i mješovite stvarnosti (MR), svjedočimo transformaciji tehnoloških napredaka i stvaranju novoga

okvira koji spaja fizičke i digitalne doživljaje. AR, VR i MR, zajedno ponekad nazivani „**Alternativna**“ ili „**ekstra stvarnost**“ (XR), uspostavili su revolucionarno područje gdje se fizički i digitalni entiteti stapaju, formirajući složene i uživljene okoline. AR postavlja digitalni sadržaj u fizički svijet, VR odnosi u potpuno virtualne domene, dok MR spaja fizičko i virtualno, omogućujući im suživot i interakciju u stvarnom vremenu.

Primjena AR-a, VR-a i MR-a u brojnim sektorima (igre, zdravstvo, obrazovanje, industrija i šire) simbolizira novu eru gdje su sposobnosti vizualizacije, simulacije i inovacije beskrajne. Od simuliranih operacija i uživljenih igraćih iskustava do interaktivnih učioničkih okolina i poboljšanih proizvodnih procesa, te su se tehnologije uvukle u svako zamislivo polje, revolucionirajući tradicionalne prakse i stvarajući nove puteve istraživanja i razvoja.

Unatoč njihovu transformacijskom potencijalu, razvojni inženjeri i tehnolozi u AR, VR i MR sferama bore se s izazovima, kao što su kompatibilnost uređaja, kreiranje sadržaja, korisničko iskustvo i tehnološka ograničenja. Napredovanje od same konceptualizacije do stvaranja uživljenih iskustava vjernih stvarnosti, orijentiranih prema korisnicima, zahtijeva stalnu inovaciju, istraživanje i duboko razumijevanje kako fizičkih tako i digitalnih interakcija. Odgovor na te izazove potiče tehnološki napredak, poboljšava sposobnosti i proširuje horizonte alternativnih stvarnosti. Te su tehnologije sada temeljni stupovi inovacija, pružajući alate da ideje i maštarije prenesemo u trodimenzionalne, interaktivne svjetove i iskustva koja su prije bila nezamisliva.

PITANJA ZA PONAVLJANJE:

Provjerite svoje znanje odgovorima na sljedeća pitanja:

1. Koja je razlika između AR-a, VR-a, i MR-a?
2. Kako se korisnik može interaktivno angažirati s AR elementima unutar *Unityja*?
3. Koje su vrste AR interakcija moguće u *Unityju*?
4. Koji su ključni koncepti praćenja objekata i detekcije površina u AR aplikacijama razvijenim u *Unityju*?
5. Što je AR sidrište (*anchor*) u kontekstu *Unityja* i zašto je važno u AR aplikacijama?
6. Što je APK datoteka i kako se može pokrenuti na mobilnom uređaju?
7. Koji su pristupi i tehnike dostupni za interakciju s objektima unutar VR okruženja u *Unityju*?
8. Kako se prilagođava ili razvija intuitivno i funkcionalno korisničko sučelje (UI) u VR iskustvima?
9. Što su *Unity* događaji i kako se njima koristi u kontekstu VR aplikacija?

LITERATURA:

1. Braun, A. & Rizzo, R. (2023). *XR Development with Unity: A beginner's guide to creating virtual, augmented, and mixed reality experiences using Unity.* Packt Publishing
2. C# Reference (2023). Preuzeto 20.10.2023. s https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/
3. Genever Benning Oculus Documentation (2023). Preuzeto 20.10.2023. s https://developer.oculus.com/documentation/
4. Nystrom, R. (2014) *Game Programming Patterns*
5. Slike zaslona iz Unity razvojnog programa.
6. Slike zaslona iz Unity Hub-a.
7. Slike zaslona iz igre Pac-Man.
8. Slike zaslona iz programa Visual Studio.
9. Unity Documentation (Version: 2023.3) (2023). Preuzeto 08.10.2023. s https://docs.unity3d.com/Manual/index.html

Projekt je sufinancirala Europska unija iz Europskog socijalnog fonda.
Za više informacija o EU fondovima molimo pogledajte web-stranicu
Ministarstva regionalnoga razvoja i fondova Europske unije.

www.strukturnifondovi.hr

Projekt je sufinancirala Europska unija iz Europskog socijalnog fonda.