Univerzitet u Beogradu
Matematički fakultet


Srđan Terzić

# Strukture podataka u razvoju video igara

Master teza

Mentor: **prof. dr Miodrag Živković**

Beograd,
2016.

Mentor:

**prof. dr Miodrag Živković**
*Matematički fakultet*
*Univerzitet u Beogradu*

Članovi komisije:

**prof. dr Vladimir Filipović**
*Matematički fakultet*
*Univerzitet u Beogradu*

**mr Jelena Hadži Purić**
*Matematički fakultet*
*Univerzitet u Beogradu*

Datum odbrane:

---

# Sadržaj

# Glava 1

# Uvod

## 1.1 Osnovni pojmovi

***Strukture podataka*** *predstavljaju osnovnu gradivnu jedinicu softverskog inženjerstva. One definišu način na koji su podaci organizovani u memoriji i zapravo olakšavaju korišćenje tih podataka unutar same aplikacije. Takođe, bitna stvar u vezi sa strukturama podataka je i to koje operacije podržavaju.*

***Video igra*** *predstavlja bilo koji vid softvera koji je namenjen zabavi, simuliranju ili učenju kroz zabavu, a koji zahteva direktnu interakciju sa čovekom(igračem).*

Konkretan cilj ovog rada je predstavljanje upotrebe različitih struktura podataka pri razvoju video igara u modernim alatima, kao i predstavljanje ovih alata i svih prednosti koje sa sobom nose.

Kada govorimo o **strukturama podataka**, bitno je reći da ih možemo podeliti mahom u dve grupe:

- Osnovne i
- Kompleksne strukture podataka

Osnovne strukture su one koje oslikavaju fizičku prezentaciju podataka u memoriji. Tipični primeri ovoga su nizovi i povezane liste. Ipak, ponekad je neophodno organizovati podatke u malo kompleksniju hijerarhiju koja će olakšati rad sa podacima. U tu svrhu

postoje kompleksne strukture podataka oličene u stablima, heš mapama ili grafovima. Za razvijanje video igara potrebno je dobro upoznati strukture podataka i znati u kom trenutku je najbolje iskoristiti koju od njih. U ovom radu obrađene su neke tipične strukture iz obe grupe koje se danas koriste u velikom obimu, a uz detaljna objašnjenja o razlozima korišćenja. Za konkretnu implementaciju i primenu određenih struktura podataka u razvoju najvećim delom su korišćene knjige [1] i [2]

**Video igre** danas predstavljaju ozbiljnu granu industrije zabave koja je u zadnjih nekoliko godina doživela veliku ekspanziju. Pojava jeftinih i moćnih alata omogućila je entuzijastama i malim grupama ljudi da prave naslove koji se mogu takmičiti sa velikim razvojnim studijima. Odjednom, pravljenje video igara je postalo svakodnevnica mladih ljudi. Ipak, pravljenje video igara je i dalje kompleksan posao koji uključuje mnogo raznorodnih oblasti kao što su 3D modelovanje, dizajniranje, slikanje, obrada zvuka, naracija i na kraju programiranje. Ovaj rad ima za cilj da približi moderne tehnike programiranja ukombinovane sa tradicionalnim postulatima u ovoj oblasti i na taj način doprinese globalnom razvoju ove grane industrije.

U radu je obrađen jedan od najpopularnijih besplatnih alata za višeplatformski razvoj igara - Unity3D po dokumentaciji [3]. Ovde je posebno značajno akcenat staviti na "višeplatformski", jer smo u današnje vreme svedoci naglog razvoja mobilnih telefona i drugih platformi na kojima se igre mogu igrati. Samim tim, alati koji omogućavaju princip *"Napiši jednom, izvršavaj svuda"* (eng. *"Write once, run anywhere"* [1]) su jako traženi i popularni, stoga je jedan od takvih alata obrađen u ovom radu.

## 1.2 Pregled po poglavljima

U nastavku je predstavljen kratak pregled rada kroz poglavlja:

- **Istorijat**

  U ovom poglavlju dat je osvrt na razvoj igara i struktura podataka kroz vreme, kao i razlike u korišćenju tih struktura podataka nekad i sad.

- **Strukture podataka**

[1] Originalno, slogan kompanije Sun Microsystems kojim se naglašavaju višeplatformske mogućnosti programskog jezika Java

S obzirom da je rad zasnovan na analizi struktura podataka, u ovom poglavlju su predstavljene obrađene strukture sa svojim specifičnostima i karakteristikama. Strukture su podeljene na osnovne i drvoidne, a za neke od njih su analizirani najčešće korišćeni algoritmi.

- **Tehnologije implementacije**

  U ovom poglavlju predstavljeni su osnovni alati i tehnologije korišćeni u izradi rada. Među njima se nalaze Unity3D, Visual studio, jezik C#, prodavnica komponenti i drugo.

- **Metodologije razvoja**

  Primena dobrih praksi u modernom razvoju softvera predstavlja svakodnevnicu svakog programera. Zbog toga je u ovom poglavlju predstavljena jedna agilna razvojna metodologija koja može biti primenjena na manje timove u razvoje video igre, a pored nje i nekoliko uzoraka za projektovanje i kratak osvrt na komponentno programiranje.

- **Detalji implementacije**

  Ovde je predstavljena implementacija celokupne igre, uključujući dekompoziciju na manje celine koje mogu biti ponovno upotrebljive. Svaka od ovih celina sadrži implementaciju jedne ili više različitih struktura podataka. Pored toga, u poglavlju je predstavljena i konkretna upotreba uzoraka za projektovanje "Posmatrač" i "Formiranje grupa objekata" koji na efikasan način rešavaju neke netrivijalne probleme na koje se nailazi u razvoju.

# Glava 2

# Istorijat

Da bi se lakše analizirale izabrane strukture podataka i bolje razumele različite mehanike izvođenja video igara, potrebno je da se napravi osvrt na istorijski razvoj kroz epohe. Ova oblast je jako mlada, ali svakako ima veoma bogatu istoriju. Glava je podeljena u dve celine koje zasebno obrađuju istorijate struktura podataka i video igara. Za analizu istorijskog razvoja video igara korišćene su [4] i [5]

## 2.1 Strukture podataka

Kao što je ranije istaknuto, u računarstvu i informatici strukture podataka predstavljaju vid organizacije podataka u memoriji radi njihovog lakšeg i efikasnijeg korišćenja. Različite vrste struktura se koriste u različitim sistemima, a neke od njih su čak specijalizovane za izvršavanje različitih zadataka. Tako se, na primer, u bazama podataka koriste B-stabla za efikasnije pretraživanje indeksa nad tabelama, kompajleri koriste heš mape da bi lakše pretraživali identifikatore, a u video igrama imamo Okt-stabla koja se koriste za particionisanje prostora. Strukture podataka, takođe, pružaju mehanizme za organizovanje velikih količina podataka kao što su velike baze podataka. Očekivano, efikasne strukture podataka su prvi korak u implementaciji efikasnih algoritama sa kojima su u tesnoj vezi. Neke metodologije i programski jezici u prvi plan stavljaju baš strukture podataka kao ključni faktor u razvoju softvera. One su uglavnom bazirane na sposobnosti računara da prikuplja podatke iz stalne memorije i smešta ih bilo gde u radnu memoriju koja je obeležena memorijskom adresom (bitovski string koji se takođe čuva u memoriji i referiše na lokaciju u memoriji). Neke strukture poput nizova su bazirane na računanju ovih

adresa običnim aritmetičkim operacijama, dok druge, poput povezanih listi, ove adrese čuvaju unutar same strukture. Takođe, postoje i strukture koje koriste oba ova principa, ponekad kombinovane na čudan način kao što je XOR povezivanje.

Implementacija neke strukture podataka podrazumeva pisanje procedura, metoda i klasa koje će kreirati i upravljati konkretnim instancama te strukture. Ipak, efikasnost strukture ne može se oceniti bez operacija koje struktura podržava. Zbog toga je uveden koncept *apstraktnog tipa podataka* koji je definisan nezavisno od načina realizacije operacija koje nad njim mogu biti izvršene, kao i njihovih matematičkih svojstava - vremenske i prostorne složenosti.

### 2.1.1 Podrška u programskim jezicima

Većina asemblerskih jezika i neki jezici niskog nivoa (PL/S, BLISS) nemaju podršku za strukture podataka. Sa druge strane, većina jezika visokog nivoa i neki viši asemblerski jezici kao što je MASM imaju specijalnu sintaksu ili neki drugi ugrađeni mehanizam za određene strukture podataka kao što su slogovi i nizovi. Jezici poput C-a podržavaju strukture, jednodimenzione i višedimenzione nizove. Ipak, većina programskih jezika poseduje kolekciju biblioteka koje omogućavaju da se implementacija konkretnih struktura podataka koristi u različitim programima. Moderni jezici visokog nivoa dolaze u paketu sa standardnim bibliotekama koje implementiraju najosnovnije strukture podataka (npr. Microsoft .NET Framework, Java Collections Framework ili C++ Standard Template library)

### 2.1.2 Primene u video igrama

Strukture podataka su našle primenu u različitim delovima video igara, bilo da je to način prikazivanja objekata na ekranu, logika za veštačku inteligenciju ili jednostavno upravljanje scenama. U nastavku je predstavljeno nekoliko najčešćih primena struktura u modernom razvoju video igara.

#### 2.1.2.1 Strukture u upravljanju scenama

Kompleksna virtuelna okruženja u modernim video igrama su često mnogo zahtevnija nego što hardver može da obradi odjednom. U svakom trenutku na hiljade poligona i različitih

efekata treba da bude iscrtano i prikazano. Ovo jako brzo može premašiti mogućnosti hardvera na kom se igra izvršava.

Od samog početka razvoja video igara, kompleksne strukture podataka i algoritmi su korišćeni da se ubrza iscrtavanje kompleksnih scena i objekata u njima, koje bi inače dovele igru do neigrivog stadijuma.

Ovde se najčešće koriste algoritmi deljenja prostora koji se zasnivaju na stablima za deljenje prostora (eng. space partitioning trees) koja uključuju **Q-stabla**, **Okt-stabla**, **KD-stabla** i druga.

Tehnika se zasniva na odsecanju dela prostora koji nije u korisnikovom vidnom polju. Na ovaj način se dobija na brzini izvršavanja i obrade objekata u sceni.

#### 2.1.2.2 Strukture za veštačku inteligenciju

Sami počeci veštačke inteligencije u video igrama datiraju još iz sredine šezdesetih godina. Pre toga igre su smišljane ili za dva igrača (bez računarskih protivnika) ili se bilo kakav objekat koji je morao da se ponaša, a nije bio čovek, skriptovao u samom kodu igre.

Primer skriptovanih protivnika predstavlja igra "Space Invaders". U ovoj igri igrač mora da uništi vanzemaljske brodove dok oni ne dođu do dna ekrana. Način na koji se ovi brodovi kreću je eksplicitno kodiran u samoj igri i na njega nikakav uticaj nema nijedan deo ili ponašanje okruženja.

Jedna od prvih pravih primena veštačke inteligencije u video igrama je ponašanje protivničkog igrača u igri "Pong" ili u nekoj od njenih varijacija kojih je bilo u izobilju. Ovde, protivnička pločica čini sve što može da vrati lopticu u igračevo polje. Algoritam proračunavanja odgovarajuće pozicije pločice realizovan je jednostavnim formulama koje računaju gde bi loptica trebalo da pređe gol liniju. U zavisnosti od dobijenih rezultata pločica se pomera na odgovarajudu poziciju. Sve ovo je bilo realizovano prostim **nizovima** koji su čuvali koordinate loptice i pločica. Postojala su i podešavanja težine, koja su uključivala brzinu pomeranja pločice kao i određenu verovatnoću da će se pločica pomeriti na pogrešno mesto.

Dugo nijedna video igra nije mnogo uznapredovala dalje od Ponga po pitanju veštačke inteligencije. Slabašna unapređenja viđena su u borilačkim igrama kao što su Nintendov "Kung Foo" ili Segin "Mortal Kombat", gde potezi koje sprovodi računar direktno zavise

od poteza koje sprovodi igrač ili od pozicije na kojoj se igrač nalazi. Ovo je realizovano jednostavnim **tabelama** iz kojih računar čita najbolju akciju u odnosu na igračev potez. U najkomplikovanijim slučajevima, računar bi primenio kratku minimaks pretragu kako bi odredio najbolju akciju. Ipak, minimaks pretraga je trebala da bude jako plitka jer su rezultati bili potrebni u realnom vremenu ili bar u najkraćem mogućem roku. Igranje u realnom vremenu je uvek predstavljalo veliku prepreku za veštačku inteligenciju. Tu je uvek na raspolaganju jako kratko vreme za računanje akcija ili mogućih budućih stanja.

Posebna vrsta veštačke inteligencije sreće se u takozvanim igrama na tabli. Klasičan primer ovakvih igara su simulacije šaha. Ovakve igre se odnose na algoritme **stabla pretrage** koji nisu naročito primenljivi u realnom vremenu zbog svoje obimnosti i dugog izvršavanja, ali u poteznom izvođenju briljantno odrađuju posao. Na primer, u simulaciji šaha koja se isporučuje uz operativnoi sistem „Windows 7“ protivniku na najvećoj težini treba i do tri sekunde da proračuna svaki potez. Na nižim nivoima, ovo se odvija skoro momentalno. Problemi ovog tipa javljaju se pre svega zbog ograničene procesorske moći i memorijskih resursa.

Zbog svega toga, jako je bitno prilagoditi strukture podataka konkretnom problemu i obezbediti tečno izvođenje igre.

#### 2.1.2.3 Strukture za simulaciju fizike i detekciju kolizije

Simuliranje fizike postaje sastavni deo modernih 3D video igara. Fizika u igrama se stara da se objekti ponašaju realistično u interakciji sa okolinom. Neophodno je voditi računa o silama koje deluju na objekte (npr. gravitacija) kao i o svim silama koje se javljaju usled međusobnog delovanja objekata jednih na druge, najčešće u vidu kolizija. Fizika i sistem kolizija imaju svoje skupove struktura podataka i algoritama koji uključuju **čvrsta tela** (eng. rigid body), **Q-stabla** i razne algoritme **deljenja prostora** (eng. sapce partitioning). Ovde se takođe koriste i strukture koje su tipične za upravljanje scenama, a sve u svrhu eliminisanja objekata koji nisu u polju pogleda. Na taj način smanjuje se prostor na kom je potrebno vršiti proračune za simuliranje fizike, što direktno utiče na performanse.

## 2.2 Video igre

### 2.2.1 Napredak kroz godine

Da bi se video igre uspešno dizajnirale i programirale u današnje vreme, neophodno je upoznati se sa nekim postulatima koji su vladali kroz istoriju razvoja ove oblasti. Generalno, promene koje su se dešavale u poslednjih skoro šezdeset godina mogu se svrstati u nekoliko različitih kategorija koje su međusobno povezane:

- **Promene u hardveru zaslužnom za pokretanje igara**

  Od početnih mašina koje su imale samo po nekoliko megabajta memorije, preko kućnih konzola i malih kućnih računara, pa sve do pojave interneta, hardver je konstantno unapređivan. Proizvođači hardvera su kao osnovna merila performansi svojih uređaja počeli da koriste video igre, koje su sa druge strane uvek pravljene tako da maksimalno uposle dostupan hardver.

- **Promene u uređajima za interakciju sa korisnikom**

  Kroz istoriju, uređaji za upravljanje u video igrama takođe su se menjali. Rane kućne konzole su koristile obične rotirajuće dugmiće i jednostavne džojstike. Trenutno, video igre se oslanjaju na intenzivnu upotrebu miša, tastature, raznih uređaja za simulaciju poput volana, a od skoro čak i na uređaje koji pomoću raznih žiroskopa i senzora mogu pratiti igračevo kretanje. Naravno, sve ovo ima ogroman uticaj na igrivost u globalu.

- **Promene u dostupnim softverskim alatima**

  U početku, programeri video igara su celokupan kod kucali sami, najčešće u nekom asemblerskom jeziku, dok su grafiku formirali piksel po piksel. Danas je dostupan veliki broj različitih alata, biblioteka i razvojnih okruženja koja doprinose brzini i lakoći rada. U ovu grupu takođe mogu da se dodaju i umetnici, animatori i muzičari koji takođe imaju pregršt korisnih alata koji su, pre svega, laki za upotrebu.

- **Promene u poslovanju**

  Kompanije koje se bave razvojem igara su promenile svoju filozofiju u velikoj meri u poslednjih pedeset godina. Dok su pre igre razvijali individualci, sada se bez velikog

tima ne može zamisliti veliki naslov. Ipak, u poslednje vreme smo svedoci sve većeg broja tzv. *indie*[1] kompanija koje unose neophodan balans i neosporan kvalitet.

- **Promene u igračkoj strukturi**.

  Dok su u prošlosti igre igrali samo mladi muškarci, danas je procenat žena-igrača na zavidnom nivou. Takođe, starosna granica skoro i da ne postoji, što je neminovno dovelo do pojave novih žanrova prilagođenih određenim ciljnim grupama.

- **Proširivanje asortimana**

  Nekad su se igre igrale samo na arkadnim aparatima. Vremenom se ovaj skup širio, pa smo tako dobijali kućne konzole, lične računare, prenosne uređaje, mobilne telefone, konzole nove generacije i tako dalje. Takođe su se i tipovi igrača menjali, pa danas razlikujemo povremene, zagrižene, opuštene ili čak one koji igraju samo igre putem interneta.

- **Promene u dizajnu igara**

  Sve ove promene, neminovno su dovele do promene u samom dizajniranju video igara. Dizajneri igara su koristili nove tehnologije i uređaje za interakciju da bi kreirali nove forme igranja, što je za cilj imalo privlačenje određenih ciljnih grupa igrača. Na kraju, svi su se oni trudili da svaka sledeća igra bude zanimljivija od prethodne.

### 2.2.2 Istorijat

Prva video igra napravljena je 1958. godine. Zvala se "Tenis for two", napravio ju je fizičar **Vilijem Higinbotam** [2] i igrala se na osciloskopu. Prva igra koja je zapravo pokretana na računaru bila je "Spacewar", grafički je bila realizovana preko ASCII karaktera, a pokretana je na PDP-1 mainframe računaru. Godine 1970. budući osnivači Atarija **Nolan Bušnel**[3] i **Ted Dabni**[4] izbacili su prvu arkadnu video igru "Computer Space". U narednih 10 godina kompanije kao što su Atari, Coleco i Magnavox su izbacivale svoje konzole

[1] skraćeno od nezavisno (eng. independant). Termin se obično koristi za igre koje su izdate mimo aktuelnih visokobudžetnih tokova.

[2] William "Willy" A. Higinbotham (25. Oktobar 1910 – 10. Novembar 1994) bio je američki fizičar, član tima koji je napravio prvu atomsku bombu. Upisan u istoriju video igara zahvaljujući svojoj "*Tennis for two*" video igri iz 1958. godine koja je jedna od prvih igara sa grafičkim prikazom.

[3] Nolan Kay Bushnell (rođen 5. Februara 1943.)je američki inženjer i preduzetnik, osnivač kompanije Atari. Tvorac aforizma "Sve najbolje igre su lake za učenje, ali teške za ovladavanje" (eng. "All the best games are easy to learn and difficult to master.")

[4] Ted Dabney (rođen 15. Maja 1937.) je američki elektro inženjer, koosnivač kompanije Atari.

za video igre koje su doživele veliku popularnost. Godine 1980. pojavila se prva 3D igra u istoriji, a zvala se "Battlezone". Ova igra je korišćena od strane vlade Sjedinjenih Američkih Država za treniranje vojnih snaga. Četiri godine kasnije pojavio se NES (Nintendo Entertainment System) koji je obeležio početak nove ere u oblasti video igara. U to vreme su i kućni računari počeli da dobijaju na popularnosti, što je dovelo do naglog razvoja velikog broja igara za ovu platformu. Činjenica da kućni računari imaju više memorijskog prostora i jaču procesorsku snagu od tadašnjih igračkih konzola dovela je do razvijanja komplikovanijih i resursno zahtevnijih igara. Ipak, industrija igračkih konzola se vraća na scenu 1995. godine kada se pojavljuje igračka konzola PlayStation firme Sony, koja je načinila veliki korak unapred na polju razvoja video igara. Do danas, ova konzola je doživela još tri inkarnacije, PlayStation 2, 3 i 4, a svaka poseduje zavidan nivo poboljšanja u odnosu na prethodnu. Kvalitetu u velikoj meri doprinosi i konkurencija na tržištu u vidu Microsoftove konzole Xbox One, kao i Nintendove konzole Wii.

# Glava 3

# Strukture podataka

U ovom poglavlju dat je osvrt na matematičku prezentaciju i opis određenih struktura podataka. Akcenat je stavljen na upotrebnu vrednost ovih struktura u današnjem razvoju video igara, a neke od njih su još detaljnije obrađene u poglavlju 6 kroz analizu njihove konkretne implementacije. Radi bolje struktuiranosti rada, sve strukture su podeljene u dve celine, osnovne i drvoidne.

## 3.1 Analiza algoritama

Analiza algoritama je oblast koja ima za cilj da predvidi ponašanje i brzinu izvršavanja algoritma bez konkretne realizacije. Na ovaj način procenjena brzina rada algoritma važi za bilo koji računar. Ipak, tačno ponašanje nekog algoritma nije moguće predvideti, osim u jednostavnim slučajevima, pa se zbog toga analiziraju samo osnovne karakteristike bez nepotrebnih detalja konkretne implementacije.

Na ovaj način dobijamo informacije o algoritmu iz kojih možemo izvući korisne zaključke i čak rasuđivati koji algoritam upotrebiti u kojoj situaciji.

Određivanje vremena izvršavanja algoritma za određeni ulaz je u praksi nemoguće izvesti, jer ne možemo sagledati sve moguće ulaze. Zbog ovoga se prvo za svaki mogući ulaz definiše njegova veličina $n$ i onda u analizi koristimo samo taj podatak. Ipak, nepostojanje opšte definicije veličine ulaza ne predstavlja veliku prepreku zato što se najčešće upoređuju različiti algoritmi, ali za rešavanje istog problema, pa će ulazi koje ispitujemo u svakom slučaju biti isti.

Problem se može javiti prilikom odabira reprezentativnog uzorka među svim ulazima. Dobra je praksa da se tada uzima najgori slučaj i radi takozvana **analiza najgoreg slučaja** gde je moguće zaključiti kako će najlošije da se ponaša algoritam.

### 3.1.1 O notacija

O notacija se koristi za opisivanje teorijskih performansi algoritama i najčešće služi za merenje vremenske ili memorijske potrošnje od strane algoritma. Ona nam takođe pruža dobar sistem za međusobno upoređivanje različitih algoritama koji obavljaju istu stvar za isti ulaz. Na primer, ako se algoritam A1 izvršava brže od algoritma A2 za ulazni skup od 1000 elemenata, ali radi sporije za veći ulazni skup od, recimo, 100 000 elemenata postaje teško uvideti koji je od ta dva algoritma zapravo bolji. Zbog toga nam O notacija daje adekvatnu sliku o performansama algoritma u zavisnosti od rasta broja ulaznih vrednosti.

Analiziranjem primera umetanja elementa u neuređeni niz, dolazi se do zaključka da ta operacija ne zavisi ni od čega drugog osim umetanja samog elementa, stoga je vreme izvršavanja konstantno. To se obeležava oznakom $O(1)$ koja označava konstantno izvršavanje nezavisno od bilo kog parametra.

Sa druge strane, linearna pretraga nesortiranog niza zavisi od broja elemenata u toj listi. Zbog toga, po O notaciji složenost ovog algoritma će biti $O(n)$, gde $n$ predstavlja broj elemenata. Što se tiče binarne pretrage, može se koristiti logaritam sa osnovom 2 za predstavljanje složenosti i to najčešće izgleda ovako: $O(\log_2 n)$.

Treba napomenuti da su performanse algoritma koji ima složenost $O(n^2)$ lošije od algoritama koji imaju složenost $O(1)$, $O(n)$, $O(\log_2 n)$ i $O(n\log_2 n)$, ali bolje od algoritama sa složenošću $O(n^3)$. U ovom radu O notacija se koristi za opisivanje ponašanja struktura podataka, tačnije algoritama koji rade sa podacima unutar njih.

### 3.1.2 Vremenska i prostorna složenost

#### 3.1.2.1 Vremenska složenost

Vremenska složenost nekog algoritma može se formirati brojanjem računskih koraka koji treba da budu izvršeni. Ipak, sam pojam računske operacije odnosi se na različite operacije koje po svojoj prirodi imaju različitu kompleksnost. Takođe, vreme izvršavanja zavisi i od

konkretnog računara, programskog jezika i drugog, pa se onda u oviru algoritma izdvaja neki osnovni korak koje se često ponavlja i on koristi kao referentna vrednost. Taj korak kod sortirajućih algoritama može da bude upoređivanje elemenata. Na primer, ako je broj upoređivanja $O(f(n))$, a broj ostalih operacija je proporcionalan broju upoređivanja, tada je $O(f(n))$ granica vremenske složenosti algoritma.

#### 3.1.2.2 Prostorna složenost

Kada se govori o prostornoj složenosti algoritma misli se na veličinu memorije koja je potrebna za izvršavanje tog algoritma. Pri tome, prostor potreban za smeštanje ulaznih podataka se ne računa jer on ne igra bitnu ulogu. Kao i kod vremenske složenosti, i ovde se analizira njeno asimptotsko ponašanje u najgorem slučaju. Prostorna složenost $O(n)$ znači da je za izvršavanje algoritma neophodna memorija proporcionalna onoj koja se koristi za smeštanje ulaznih podataka. Ako je, na primer, prostorna složenost algoritma $O(1)$, to znači da je memorijski prostor potreban za njegovo izvršavanje ograničen konstantom i ne zavisi od veličine ulaznih podataka.

O osnovnim stvarima u vezi sa strukturama podataka više u [6].

## 3.2 Osnovne strukture podataka

U narednom delu su predstavljene neke od osnovnih struktura podataka jednodimenzione prirode, koje osim pukog čuvanja podataka odražavaju i njihov redosled. One predstavljaju osnovni alat u razvoju bilo kog softvera, ne samo video igara.

### 3.2.1 Bitvektor

Bitvektor u osnovi predstavlja specijalnu vrstu niza, takvog da se na svakoj poziciji u nizu nalazi bit (0 ili 1). Ipak, to nije klasičan niz bulovskih vrednosti već efikasnija struktura. Niz bulovskih vrednosti može biti zahtevan jer kompilatori najčešće koriste neki komplikovaniji tip podatka za predstavljanje jednog elementa kao što je ceo broj.

Samim tim što se sastoji od bitova, neophodno je dobro poznavati bitovske operacije da bi se bitvektor koristio na najbolji mogući način.

Osnovne operacije nad vektorima su & (konjunkcija), | (disjunkcija), ~(negacija), ⊗ (ekskluzivna disjunkcija), « (šiftovanje ulevo) i » (šiftovanje udesno).

Ovi operatori se koriste u kombinaciji sa bitovskim maskama da bi se dobile vrednosti određenih bitova, promenila vrednost bitu ili pak resetovao ceo skup bitova. Takođe, bit na poziciji $n$ možemo dobiti formiranjem maske šiftovanjem jedinice za $n$ mesta ulevo i obavljanjem bitovske konjunkcije ova dva niza. Ako je rezultat urađene konjunkcije 0, to znači da je i vrednost bita na poziciji $n$ 0. Ako je, pak, rezultat neki broj koji nije nula, dolazi se do zaključka da je vrednost bita na poziciji $n$ 1.

#### 3.2.1.1 Primena

Bitvektor je našao primenu u industriji video igara pre svega za čuvanje stanja između različitih scena gde je potrebno čuvati informaciju o prisutnosti za veliki broj objekata. Pored toga, korisno je znati koja vrata su zaključana, a koja otključana ili koji je protivnik mrtav, a koji živ. Kada se sve ovo ukomponuje sa veoma popularnim mehanizmom "*brzog čuvanja*" (eng. Quick save) dolazi se do zaključka da bitvektor može da zadovolji sve zahteve. Mehanizam brzog čuvanja se realizuje tako što se čuvaju dva niza od kojih je jedan bitvektor, a drugi standardni niz objekata za koje se prati hoće li biti prisutni u sceni ili ne. Nizovi se formiraju tako da svakom elementu jednog, odgovara tačno jedan element drugog niza pa će njihovi indeksi u oba niza biti isti. Na ovaj način mi iteriramo samo kroz brzi bitvektor i instanciramo samo objekte koji imaju vrednost 1.

U ovim situacijama bitvektor briljira zbog svoje jednostavnosti i minimalne zahtevnosti, pa zbog toga često predstavlja razuman izbor pri razvoju video igara.

### 3.2.2 Višedimenzioni niz

Klasični nizovi imaju samo jednu dimenziju, tačnije njihovi elementi su poređani jedan za drugim. Nizovi ipak mogu biti organizovani i u više dimenzija, gde je taj broj najčešće 2 ili 3. Zbog jednostavnosti prikaza, ovde se zadržavamo samo na nizovima dimenzije 2.

Ako jednodimenzioni niz izgleda kao linija, onda dvodimenzioni niz možemo predstaviti kao mrežu. Tako formirani niz ima širinu i visinu i određeni broj polja.

Deklarisanje višedimenzionih nizova u jeziku C# izgleda ovako:

```
int array2d[5][5];
int array3d[4][4][4];
```

Skripta 3.1: Deklarisanje nizova u C#-u

Što respektivno predstavlja dvodimenzioni niz od 5x5 elemenata i trodimenzioni niz od 4x4x4 elemenata. Dvodimenzioni niz još možemo shvatiti kao niz kod kog svaki element predstavlja još jedan niz.

Ova struktura podataka je jako efikasna zbog brzine pristupa elementima koja se obavlja u konstantnom vremenu (O(1)), ali sa druge strane njena fiksirana dimenzija može predstavljati ograničavajući faktor. Pored toga, svi elementi ove strukture moraju biti istog tipa, pa se dolazi do zaključka da korišćenje niza zahteva dobru analizu problema i implementaciju na pravi način.

#### 3.2.2.1 Primena

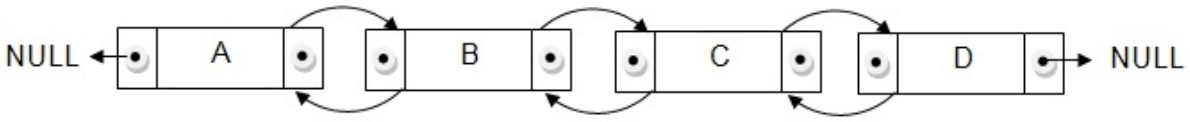
Pored svih nedostataka, višedimenzioni niz je našao primenu u velikom broju modernih igara, mahom logičke prirode, gde određene elemente treba pomerati i premeštati po nekoj mreži. Isto važi i za mape po kojima se kreću likovi, gde korišćenje višedimenzionog niza olakšava implementaciju algoritama za traženje najkraćih puteva. Ne treba zaboraviti i da se najobičnija tekstura predstavlja kao dvodimenzioni niz piksela uz najrazličitije primene.

Višedimenzioni niz je intenzivno korišćen u implementacionom delu ovog rada, pre svega u modulu za igru "Spoji 3", ali i u modulu za generisanje lavirinta gde je formirana tekstura sa pikselima koji predstavljaju zidove i prolaze i kao takva korišćena za generisanje pravih objekata u igri.

### 3.2.3 Dvostruko povezana lista

Vektor se pokazuje kao jako dobar izbor u nekim situacijama, ali često je potrebna struktura čija veličina može dinamički da se menja uz podržavanje umetanja i brisanja elemenata između drugih elemenata. Tu se kao rešenje nameće dvostruko povezana lista, specijalan slučaj jednostruko povezane liste.

Jednostruko povezana lista predstavlja skup parova koji sadrže element i pokazivač na sledeći element, pa su na taj način svi elementi ulančani. Sa druge strane dvostruko povezana lista sadrži dva pokazivača koji pokazuju na prethodni i sledeći element i na taj način ubrzavaju pretragu i lociranje određenih elemenata u celoj strukturi. Grafička prezentacija dvostruko povezane liste je data na slici 3.1 Prilikom unošenja ili brisanja iz liste, dovoljno je prevezati pokazivače tako da se novi element skladno uklopi u celinu. Ovo konkretno znači da se prilikom unošenja elementa pronađu dva elementa između kojih treba ubaciti željeni element i zatim pokazivač na sledeći element prethodnika preusmeriti da pokazuje na naš element, a pokazivač na prethodni element sledbenika takođe preusmeriti da pokazuje na željeni element. Nakon toga i pokazivače elementa koji se unose treba usmeriti da pokazuju na prethodnika i sledbenika i na taj način lista je proširena za jedan element bez narušavanja redosleda i nepotrebnog pomeranja velikog broja elemenata kao što je slučaj sa klasičnim nizom.




Slika 3.1: Grafička prezentacija dvostruko povezane liste

Sličan postupak se sprovodi i prilikom izbacivanja elementa gde se pokazivač na prethodni element sledbenika preusmerava na prethodnika, a pokazivač na sledeći element prethodnika preusmerava na sledbenika.

Problem sa ovom strukturom predstavlja veća zahtevnost u memorijskim resursima, pošto uz svaki podatak moramo da čuvamo po jedan pokazivač na prethodni i sledeći element. Takođe, pristupanje elementima moguće je samo kroz iteriranje kroz elemente (tj. njihove pokazivače), što značajno smanjuje brzinu pristupa elementima.

Još jedan problem javlja se pri određivanju kraja i početka liste, odnosno formiranju pokazivača na prethodnika prvog elementa i pokazivača na sledbenika poslednjeg elementa. U svrhu rešavanja tog problema vrednost pokazivača se postavlja na **nil**, što predstavlja pokazivač ni na šta.

#### 3.2.3.1 Primena

U razvoju video igara, lista se može smatrati osnovnim oružjem sa skladištenje podataka. Zbog svoje fleksibilnosti često predstavlja prvi izbor, pogotovo što softver kao što je video

igra zahteva intenzivno ubacivanje i izbacivanje elemenata iz kolekcija gde nizovi ne predstavljaju pametno rešenje. Pored toga, memorijski prostor danas uglavnom ne predstavlja veliki problem (osim u razvoju igara za mobilne platforme, gde je neophodno voditi računa o svakom resursu) pa zbog toga lista dobija prednost u odnosu na klasične nizove.

Klasična primena predstavlja simulacija inventara, gde se elementi često premeštaju, uklanjaju i unose. Takva implementacija se nalazi u ovom radu, čiji detalji su opisani u poglavlju 6.

Veoma je bitno napomenuti da većina modernih okvira za implementaciju (*eng. Framework*) u sebi sadrži implementaciju dvostruko povezane liste, te su programeri pošteđeni vođenja računa o pokazivačima i standardnim operacijama. Ipak, radi efikasnijeg korišćenja ovih struktura neophodno je dobro poznavati mehanizme funkcionisanja svake od njih.

## 3.3 Drvoidne strukture podataka

Vektori i povezane liste su jednodimenzione strukture podataka koje odražavaju samo redosled elemenata. Kada je potrebno predstaviti složenije odnose između elemenata korisno je koristiti drvoidne strukture podataka ili skraćeno stabla.

Stablo je hijerarhijska struktura podataka koja predstavlja skup čvorova i grana koje povezuju čvorove na određeni način. Jedan od čvorova je izdvojen i on predstavlja koren stabla, što znači da nema direktne pretke. Ostali čvorovi grade nivoe stabla, i to tako što svi čvorovi vezani sa korenom čine nivo 1 hijerarhije, svi čvorovi vezani na čvorove nivoa 1 (osim korena) čine nivo 2, itd. Svaka grana u stablu povezuje čvor sa njegovim prethodnikom (ocem), osim u slučaju korena stabla, koji je jedini čvor bez oca. Osnovna karakteristika stabala jeste da u njemu ne postoje ciklusi (zatvoreni putevi), već između svaka dva čvora postoji jedinstven put. Analogno, čvorovi mogu imati sinove (mada ne moraju), a maksimalni broj sinova čvora u grafu naziva se stepen stabla. Na primer, stablo stepena dva naziva se binarno stablo, a svaki čvor u njemu može imati maksimalno dva sina, levog i desnog. Čvor koji nema decu zove se list, a čvor koji nije list ili koren naziva se unutrašnji čvor.

Visina stabla predstavlja najveći nivo hijerarhije u njemu, a to još predstavlja i maksimalno rastojanje od korena do nekog čvora.

Što se tiče predstavljanja i implementacije stabala, razlikuju se dva pristupa, eksplicitno i implicitno predstavljanje.

Pri **eksplicitnom** predstavljanju se čvor sa k sinova predstavlja slogom čiji je deo vektor sa k pokazivača ka sinovima, a ponekad je deo tog sloga i pokazivač ka ocu. Poželjno je da svi čvorovi budu istog tipa, tj. sa m pokazivača gde m predstavlja stepen stabla.

Za **implicitno** predstavljanje stabla ne koriste se pokazivači već se svi čvorovi smeštaju u vektor, a veze između čvorova određene su njihovom pozicijom u vektoru. Ako je sa A označen vektor u koji se smeštaju čvorovi binarnog stabla, onda se koren smešta u A[1], a njegov levi, odnosno desni sin u A[2] i A[3], itd. Ovo znači da se čvorovi u vektoru zapisuju onim redom kojim se prolaze s leva udesno po nivoima. Nedostatak ove metode predstavlja to što u ovako formiranom vektoru moramo rezervisati mesta čak i za nedostajuće čvorove. Ovakav način predstavljanja stabla je pogodan zbog svoje kompaktnosti, ali može se pokazati kao loš u slučaju da je stablo neuravnoteženo (što znači da su neki listovi mnogo više udaljeni od korena od nekih drugih) pa se mora rezervisati prostor za veliki broj nepostojećih čvorova, što dovodi do jako neefikasnog korišćenja memorijskog prostora.

U nastavku su prikazana neka najčešće korišćena stabla u razvoju video igara.

### 3.3.1 Binarno stablo pretrage

Kao što je ranije rečeno, binarno stablo je svako stablo stepena 2, što znači da svaki njegov čvor ima maksimalno dva sina. Zbog toga se može fiksirati neko preslikavanje skupa njegovih sinova u skup levi, desni. Ovo preslikavanje može se iskoristiti i za iscrtavanje stabla u ravni tako što leve sinove iscrtavamo levo, a desne desno.

U binarnom stablu pretrage (BSP) ključ svakog čvora veći je od svih ključeva levog podstabla, a manji od ključeva desnog podstabla. BSP omogućava efikasno izvršavanje sledeće tri operacije:

- **Nalaženje ključa.** Ovo je operacija po kojoj je struktura podataka BSP dobila ime. Potrebno je u stablu pronaći element sa zadatim ključem x. Broj x upoređuje se sa ključem korena stabla i ako je vrednost ista, element je pronađen. Ako je, pak $x < r$ (ili $x > r$), onda se traženje rekurzivno nastavlja u levom, odnosno desnom podstablu.

- **Umetanje.** Kao i pronalaženje ključa, umetanje je takođe jednostavna operacija. Ključ x koji treba umetnuti najpre se potraži u stablu. Ako je pronađen umetanje je gotovo, jer stoji pretpostavka da se u stablu ne čuvaju duple vrednosti. Ako se x ne nalazi u stablu, onda se prilikom pretrage došlo do lista ili do čvora bez jednog sina, upravo sa one strane gde treba umetnuti novi čvor. Tada se x umeće kao levi, odnosno desni sin tog čvora u zavisnosti od toga da li je vrednost ključa manja ili veća od vrednosti ključa tog čvora.

- **Brisanje.** Brisanje čvora iz stabla nešto je komplikovanija operacija. Najlakši slučaj je brisanje lista iz stabla. Slično se može uraditi i sa čvorom koji ima samo jednog sina, gde se čvor uklanja, a celo podstablo sa korenom u njegovom sinu se podiže za jedan nivo tako da mu koren bude na mestu obrisanog čvora. Ako je potrebno ukloniti unutrašnji čvor B stabla koji ima oba sina, može se najpre u levom podstablu čvora B pronaći najdesniji čvor čvor x, odnosno čvor sa najvećim ključem. Ovo se može realizovati tako što polazeći od korena tog stabla prelazimo iz svakog čvora u njegovog desnog sina. Na kraju tog puta nalazi se čvor x. Nakon toga ključ čvora x kopira se u ključ čvora B, a čvor x se ukljanja na već opisan način pošto on predstavlja ili list ili čvor sa jednim sinom. Bitno je napomenuti da stablo na ovaj način ostaje konzistentno jer je ključ x veći od svih ključeva u levom podstablu svoje nove lokacije. Čvor x se zove prethodnik čvora B u stablu. Isti rezultat može se postići pomoću najlevljeg čvora u desnom podstablu čvora B, njegovog sledbenika.

#### 3.3.1.1 Složenost

Za sve tri razmatrane operacije vreme izvršenja zavisi od oblika stabla i položaja čvora na kome se vrši intervencija. U najgorem slučaju potraga se završava u listu stabla, dok sve ostale operacije zahtevaju samo konstantan broj elementarnih operacija. Dakle, u najgorem slučaju je složenost proporcionalna visini stabla. Ako je stablo sa n čvorova u razumnoj meri uravnoteženo onda je njegova visina proporcionalna sa $\log_2 n$ pa se sve tri operacije efikasno izvršavaju. Problem nastaje kada stablo nije uravnoteženo. Ovakva stabla mogu da nastanu kao rezultat umetanja elemenata u uređenom ili skoro uređenom redosledu. Brisanja mogu da izazovu probleme čak i ako se izvode u slučajnom redosledu. Razlog tome predstavlja asimetrija do koje dolazi ako se uvek bira prethodnik za zamenu obrisanog čvora. Asimetrija se može smanjiti ako se za zamenu obrisanog čvora naizmenično koriste prethodnik i sledbenik.

#### 3.3.1.2 Primena

Glavna prednost binarnog stabla pretrage predstavlja njegov koncept rekurzivnog deljenja elemenata na celine. Ovo stablo je iskorišćeno kao polazna tačka u formiranju stabala za binarno particionisanje prostora koje ima ogromnu primenu u razvoju video igara kao što je deljenje poligona u 3D sceni radi određivanja koji od njih će biti vidljivi a koji ne. Na kraju, binarno stablo pretrage predstavlja jednu od osnovnih drvoidnih struktura i kao takvo je neophodno za bolje razumevanje kompleksnijih stabala opisanih u nastavku ovog poglavlja.

### 3.3.2 Minimaks stablo

Minimaks stablo u osnovi predstavlja obično stablo sa proizvoljnim brojem sinova svakog čvora. Ono predstavlja jednu vrstu stabla igre, kod kog se ishodi determinističke igre po potezima predstavljaju u vidu stabla.

Čvorovi u minimaks stablu uzimaju celobrojne vrednosti koje predstavljaju bodovane ishode konkretne igre. Ovako napravljeno stablo predstavlja osnovu za minimaks algoritam traženja optimalnog poteza u datom trenutku.

Minimaks algoritam pretraživanjem stabla igre za igrača koji je na potezu odreduje najbolji mogući potez u datoj situaciji, pri čemu se pod "najboljim" podrazumeva najbolji za zadati čvor i zadatu dubinu pretraživanja. Pretpostavimo da je bolji potez onaj koji obezbeđuje veću vrednost na kraju izvršavanja algoritma i, jednostavnosti radi, da se pretraživanje vrši do fiksne dubine stabla.

Nekom heuristikom se zatim dodeljuju celobrojne vrednosti u listove stabla koji predstavljaju konačne, tj. završne stadijume igre, u zavisnosti od povoljnosti ishoda za igrača. Celobrojne vrednosti se dodeljuju tako što veći brojevi predstavljaju pozitivan ishod za igrača, a manji negativan. Nastavak postupka podrazumeva rekurzivnu dodelu ocena čvorovima gde se čvoru dodeljuje minimum ocena čvorova potomaka ako je u tom čvoru na potezu protivnik, a maksimum ocena čvorova potomaka ako je na potezu igrač.

Ocena početnog čvora predstavlja maksimum ocena čvorova neposrednih potomaka i rezultat je potez kom taj maksimum odgovara. Dakle, algoritam karakteriše maksimizovanje ocene kada je na potezu igrač, a minimizovanje ocene kada je na potezu protivnik.

Algoritam minimaks (kao i ostali algoritmi zasnovani na minimaksingu) vrši izbor poteza samo na osnovu vrednosti koje su pridružene čvorovima na maksimalnoj dubini pretraživanja. To znači da se ne ispituju potezi koji dalje slede i da se ne koriste informacije o njima (a te informacije mogu biti veoma važne i često bi promenile odluku o izabranom potezu). Kada je neki potez odabran na osnovu čvorova na nekoj dubini, a zatim i odigran, informacija o tome se ne koristi u procesu izbora narednog poteza (npr. ako je u šahu izabran neki potez kojem se protivniku daje šah, u sledećem potezu pretraživanje kreće se iznova i često neće biti izabran potez koji vodi do šaha protivniku, sada u dva poteza). Dakle, pri pretraživanju stabla igre "vide se" samo čvorovi na nekoj fiksnoj dubini – ni oni posle, ni oni pre njih. Ovaj fenomen često se naziva efekat horizonta (*eng. horizon effect*). Za bolje razumevanje minimaks algoritma pogledati [7].

#### 3.3.2.1 Složenost

Ako se maksimalna dubina stabla označava sa $m$, a u svakom trenutku postoji maksimalno $b$ legalnih poteza, tada se vremenska složenost algoritma ocenjuje sa $O(b^m)$, a prostorna sa $O(bm)$, ako se svi čvorovi otvaraju odjednom ili $O(b)$ ako se čvorovi otvaraju pojedinačno.

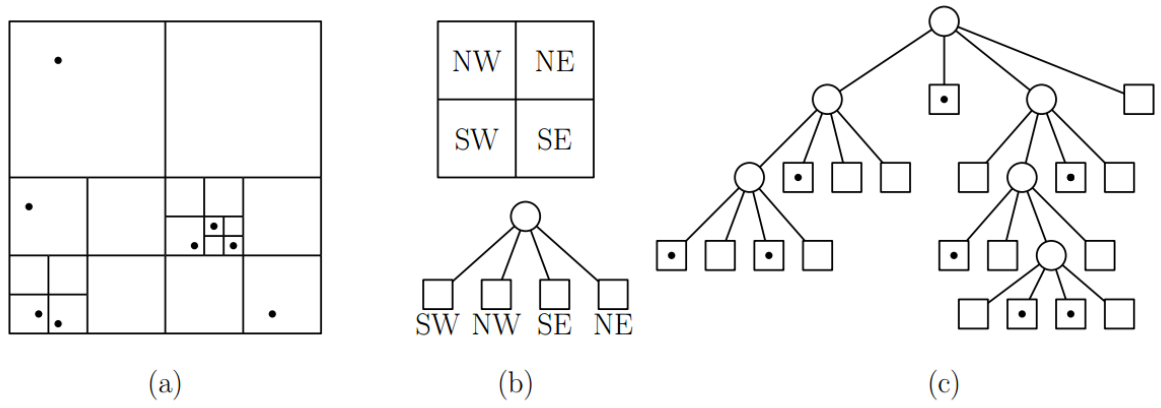
Problem sa minimaks algoritmom definitivno predstavlja njegova velika složenost. Ipak, ovaj problem se lako prevazilazi tehnikom Alfa-Beta odsecanja koja zanemaruje delove stabla koji nisu od interesa, a donosi potpuno isti rezultat kao i minimaks algoritam. Više o Alfa-Beta odsecanju možete naći u [8].

#### 3.3.2.2 Primena

Minimaks stablo, kao i minimaks algoritam ima veliku primenu u logičkim igrama u kojima se koristi simulacija igranja protivnika. Igre kao što su šah, iks-oks, bekgemon (*eng. backgammon*), čekers (*eng. checkers*) i slično, intenzivno koriste minimaks stabla. U ovakvim igrama težina protivnika se može nameštati jednostavnim menjanjem dubine do koje pretraga stabla može da ide, gde dopuštanje silaska do kraja stabla rezultuje u nemogućem scenariju za igračevu pobedu. U ovom radu je obrađena igra iks-oks sa implementacijom minimaks stabla za odabir poteza računarskog protivnika.

### 3.3.3 Q-stablo

Q-stablo (eng. Quad-tree) predstavlja drvoidnu strukturu podataka u kojoj svaki čvor ima tačno četiri sina, ili je bez sinova ukoliko se radi o listu. Ovo stablo se najčešće koristi za deljenje dvodimenzionog prostora rekurzivnom podelom na četiri kvadranta ili regije. Regije su najčešće kvadratnog oblika, ali mogu biti i pravougaone ili čak proizvoljnih oblika. Ovakva podela je prikazana na slici 3.2 gde se može uočiti podela na kvadrante (NW, NE, SW, SE), kao i podela dvodimenzionog prostora na osnovu kreiranog stabla na slikama (a) i (c).




Slika 3.2: Grafička prezentacija Q-stabla

**Rafael Finkel** [1] i **Džon Bentli** [2] dali su naziv ovoj strukturi podataka Q-stablo 1974. godine.

Sve forme Q-stabla dele neke zajedničke karakteristike koje su između ostalog:

- Razlažu prostor na ćelije
- Svaka ćelija ima maksimalni kapacitet. Kada je taj kapacitet dostignut ćelija se deli.

#### 3.3.3.1 Vrste Q-stabala

Q-stabla mogu se klasifikovati prema vrsti podataka koje oni predstavljaju, što uključuje regije, tačke, prave i krive. Q-stabla takođe mogu biti klasifikovana po tome da li je oblik stabla nezavisan od redosleda podataka koji obrađuje. Neki uobičajeni tipovi su:

[1] Raphael Finkel (rođen 1951.) je američki informatičar i profesor na univerzitetu Kentaki (*eng. Kentucky*). Autor knjige *"An Operating Systems Vade Mecum"* i kreator Q-stabla kao strukture podataka.

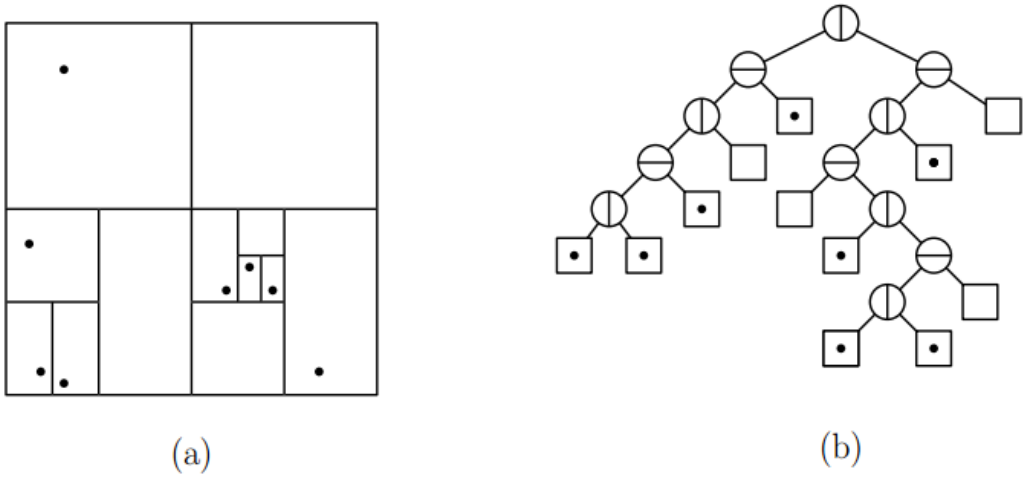
[2] Jon Louis Bentley (rođen 20. Februara 1953.) je američki informatičar zaslužan za algoritam particionisanja baziran na heuristikama uz pomoć KD-stabla.

**Regionsko Q-stablo** Regionsko Q-stablo predstavlja podelu prostora u dve dimenzije tako što razlaže regiju na četiri kvadranta, a svaki od njih na podkvadrante i tako dalje. Svaki list sadrži podatke koji odgovaraju određenom podregionu. Svaki čvor u stablu ili ima tačno četvoro dece, ili nema decu (list).

**Tačkasto Q-stablo** Tačkasto Q-stablo kombinuje pristup mrežne strukture podataka (*eng. Grid*) sa višedimenzionom generalizacijom stabla binarnog pretraživanja. Svaki unutrašnji čvor je povezan sa podatkovnim zapisom lokacije tačke i ima četiri sledbenika (NW, NE, SW i SE). Svaka kvadrantska podela je centrirana nad podatkovnom tačkom. Vreme potrebno za formiranje Q-stabla je $O(n\log\, n)$, a vreme traženja je $O(\log\, n)$.

Čvor tačkastog kvadratnog stabla je sličan čvoru binarnog stabla, sa glavnom razlikom u tome što ima četiri pokazivača (jedan za svaki kvadrant), umesto dva ("levi" i "desni") kao u uobičajenom binarnom stablu. Takođe, ključ se obično dekomponuje na dva dela, koji se odnose na x i y koordinate.

**Binarno Q-stablo** Ova specijalna konstrukcija može se primeniti zbog boljeg organizovanja prostora, gde se primenjuje binarno deljenje. To realizujemo tako što na svakom parnom nivou vršimo podelu po x-osi, a na svakom neparnom po y-osi i na taj način dobijamo po dva sina svakog čvora. Naravno, na ovaj način smo ograničeni na podelu na dva dela, ali u nekim situacijama nam ovo može biti od izuzetne koristi. Organzacija ovakvog stabla predstavljena je na slici 3.3, gde je na slici (a) predstavljeno rasparčavanje ravni, a na slici (b) organizacija samog stabla.




Slika 3.3: Grafička prezentacija binarnog Q-stabla

Više o ovoj podeli u [9][10]

#### 3.3.3.2 Primena

Q-stablo se koristi u modernom razvoju video igara, najviše u upravljanju vidljivim objektima na scenama gde se po regionima određuje koji će objekat biti vidljiv a koji ne. Takođe, ima veliku primenu i u proceduralnom generisanju nivoa, prepreka i sistema soba i hodnika što je prikazano i u ovom radu. Jedan od najpoznatijih primera primene Q-stabla u proceduralnom generisanju u video igri je **Konvejeva** [3] igra života (*eng. Game of life*). Pored toga, Q-stablo igra bitnu ulogu i u preciznoj detekciji kolizije i kao takvo je sastavni deo većine modula za simulaciju fizike, kao što je Nvidia PhysX. (4.1.5)

### 3.3.4 Okt-stablo

Nakon upoznavanja sa Q-stablom, upoznavanje sa Okt-stablom je mnogo lakše. Okt-stablo predstavlja komplikovaniju verziju Q-stabla kod kog se svaki čvor deli na 8 podčvorova. Ono što je kod Q-stabla bio kvadrat, tj. dvodimenzioni prostor, kod Okt-stabla predstavlja kocka, tj. trodimenzioni objekat. Na ovaj način se, sada već trodimenzioni prostor, deli na manje delove rekurzivno. Slično kao Q-stablo, i Okt-stablo se može koristiti za efikasno nalaženje tačaka u prostoru (samo sada u trodimenzionom prostoru).

Okt-stablo može jako brzo da se razgrana i, generalno, nije potrebno mnogo nivoa da bi se generisao ogroman broj čvorova. Ono što često može predstavljati problem jeste to da se Okt-stabla često generišu dinamički u zavisnosti od dostupnosti određenih informacija, što neminovno može dovesti do neizbalansiranog stabla koje nije zahvalno za obradu i kod kog su neki delovi prostora pokriveni mnogo detaljnije od ostalih.

#### 3.3.4.1 Primena

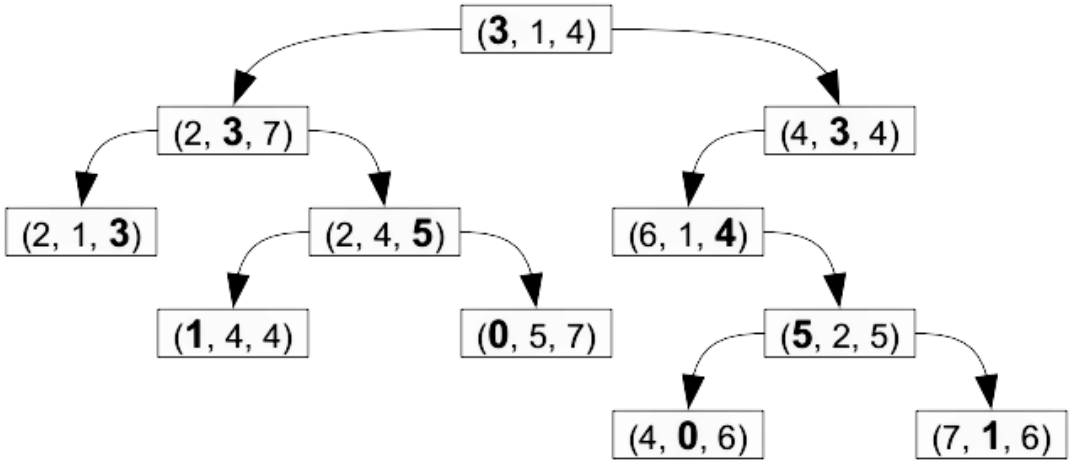
Kao što je već rečeno, Okt-stabla su zgodna za pretraživanje 3D prostora. Pored toga koriste se još i u eifikasnoj detekciji kolizije (slično kao i Q-stabla, samo za trodimenzioni prostor). Pored toga, Okt-stabla imaju primenu i u apstraktnim 3D prostorima kao što je trodimenzioni prostor boja čije su dimenzije Crvena, Zelena i Plava (*eng. RGB*). Ovde se Okt-stablo može koristiti za razne izmene na slikama, redukciju boja itd.

[3] John Horton Conway (rođen 26. Decembra 1937.) je britanski matematičar i profesor na univerzitetu Prinston (*eng. Princeton*). Značajno je doprineo razvoju kombinovane teorije igara kroz svoju Igru života (*eng. Game of life*).

### 3.3.5 KD-stablo

KD-stablo predstavlja generalizaciju binarnog stabla kod kog čvorovi mogu sadržati neki višedimenzioni podatak. Najčešće su to tačke u ravni ili u prostoru, što respektivno predstavlja dvodimenziono i trodimenziono KD-stablo. KD-stablo sa čvorovima dimenzije 1 je standardno binarno stablo.

Razvrstavanje čvorova unutar stabla se obavlja tako što se na svakom nivou uzima ona dimenzija elementa koja je jednaka ostatku pri deljenju nivoa na kom se čvor nalazi sa maksimalnom dimenzijom svakog čvora. Primer ovakvog razvrstavanja dat je u nastavku.




Slika 3.4: KD-stablo dimenzije 3

Na slici 3.4 je predstavljeno KD-stablo dimenzije 3. U svakom čvoru podebljan je element koji se koristi kao referentni i u odnosu na kog se posmatraju podstabla. Ovo znači da se u levom podstablu korena nalaze elementi koji na nultoj poziciji u vektoru (zbog toga što je referentni element na nultoj poziciji) imaju manju vrednost od referentne vrednosti korena, koja je u ovom slučaju 3. Analogno, u desnom podstablu se nalaze čvorovi koji na nultoj poziciji u vektoru imaju veće vrednosti od referentne. Kako se dubina stabla povećava, tako se pomeraju i pozicije referentnih elemenata. Ako se nalazimo na nivou $n$, referentna pozicija elementa biće jednaka $[n\%3]$, gde 3 predstavlja dimenziju svakog čvora.

#### 3.3.5.1 Osnovne operacije

Što se tiče pretrage KD-stabla na neki čvor P, polazeći od korena ispituje se nulta pozicija korena. Ako je vrednost tačke P na toj poziciji manja od vrednosti u korenu silazi se u levo podstablo, a u suprotnom u desno. Na narednom nivou ponavlja se postupak sa razlikom u izboru pozicije u vektoru koja se određuje po ranije objašnjenom principu. Pretraga se završava ili kada je pronađen željeni element P ili kada se otpadne sa stabla što znači da element P nije pronađen. Složenost ove operacije je u najgorem slučaju $O(n)$, ali se mnogo bolje ponaša u prosečnom slučaju kada složenost iznosi $O(\log\, n)$.

Operacija umetanja se obavlja slično kao u standardnom binarnom stablu pretrage, osim što se opet mora voditi računa o referentnim pozicijama.

Kad se govori o izbacivanju elemenata iz stabla, postoje dva pristupa. Prvi predstavlja formiranje liste svih čvorova i listova iz podstabla čvora koji izbacujemo i nakon njegovog izbacivanja ponovo se kreira taj deo stabla. Drugi pristup se svodi na traženje zamenskog čvora i njegovo ubacivanje. Ako je čvor koji se izbacuje list, zamenski čvor nije potreban, a ako je taj čvor unutrašnji čvor, nalazi se zamenski čvor $p$, upisuje se vrednost čvora $p$ u čvor koji se izbacuje i zatim rekurzivno čvor p. Složenost je ista kao i u slučaju pretrage.

Zamenska tačka se pronalazi u zavisnosti od referentne pozicije čvora koji se izbacuje. Pretražuje se desno podstablo čvora (ako postoji) na najmanju vrednost na referentnoj poziciji. U suprotnom se pretražuje levo podstablo na najveću vrednost na referentnoj poziciji.

#### 3.3.5.2 Pronalaženje najbližih suseda

Pronalaženje najbližeg suseda se zasniva na efikasnom odsecanju delova stabla. Ovaj algoritam se odvija u sledećim koracima:

- Počinje se od korena stabla i stablo se obilazi kao u pretrazi opisanoj ranije dok se ne dođe do nekog lista.
- Taj list se uzima za trenutni optimum.
  - Ako je tekući čvor bliži od trenutnog optimuma, on postaje trenutni optimum.

  - Algoritam zatim proverava da li postoje tačke sa druge strane deobne ravni koje su bliže traženoj tački od trenutnog optimuma. Konceptualno, ovo se obavlja posmatranjem preseka deobne ravni i hipersfere koja okružuje tačku pretrage, a čiji je poluprečnik jednak udaljenosti od ternutno optimalne tačke.
    * Ako hipersfera prelazi deobnu ravan, moguće je da postoje bliže tačke sa druge strane, pa se algoritam mora spustiti na sledeći nivo ispod u potrazi za bližom tačkom, a sve primenjujući isti rekurzivni proces.
    * Ako hipersfera nema preseka sa deobnom ravni, algoritam nastavlja sa kretanjem uz stablo, a cela grana sa druge strane je eliminisana.

- Kada se ovaj proces završi u korenu stabla, pretraga je završena.

I ovaj algoritam takođe ima složenost prosečnog slučaja od $O(\log n)$.

#### 3.3.5.3 Primena

Zbog svoje efikasnosti u pretrazi najbližih suseda (kako jednog, tako i više njih) KD-stablo je našlo svoje mesto u industriji video igara. Sa pojavom igara tipa najezde (*eng. swarm*) protivnika ili odbrambenih simulacija (*eng. tower defence*) gde je potrebno jako brzo izvršavati analize nabližih protivnika KD-stablo je pokazalo svoj puni potencijal. Takođe, jako efikasan algoritam formiranja novog stabla odgovara konstantnoj potrebi za ponovnim kreiranjem pozicija objekata u realnom vremenu. Zbog svoje velike prednosti, u ovom radu je predstavljena implementacija jednog ovakvog stabla u klasičnom primeru odbrambene simulacije.

O svim obrađenim strukturama podataka, kao i o velikom broju neobrađenih više se može porčitati u [1] [11] [2].

# Glava 4

# Tehnologije implementacije

U ovom odeljku dat je osvrt na konkretan alat korišćen pri izradi praktičnog dela rada, Unity 3D, i sve njegove delove koji ga čine kompletnim rešenjem za razvoj video igara. Takođe, pošto za izradu rada nije korišćen standardni MonoDevelop već Microsoft Visual Studio 2012, dat je kratak osvrt i na njegove mogućnosti i prednosti.

## 4.1 O Unity 3D pokretaču igara

Unity je pre svega **pokretač igara** (*eng. Game Engine*), što predstavlja kompletno razvojno okruženje koje uključuje simulaciju fizike, upravljenje scenama i drugo. U njemu su sadržani sistemi za upravljanje scenama, moduli za simulaciju fizike kao i celokupna podrška za prikazivanje bilo kakvog vida grafike u realnom vremenu. Jedna od glavnih karakteristika ovog alata je to što je on višeplatformski, tj. softveri napisani na njemu mogu biti izvršavani na različitim platformama koje se mogu svrstati u četiri kategorije:

- **Desktop** - Izvršne verzije se mogu praviti za Windows, Windows prodavnicu, Mac OS X i Linux (oficijelno je podržana distribucija Ubuntu, ali je funkcionalno i za većinu ostalih distribucija). Programeri ovde mogu birati između pravljenja 32-bitnih i 64-bitnih instalacionih fajlova, dok je za Mac OS X dostupna i opcija Univerzalnog građenja (eng. *Universal build*).
- **Mobilne** - Ovde su uključeni Android, iOS, Windows phone 8 i BlackBerry.

- **Web** - U slučaju izvršavanja u Web okruženju, postoje tri opcije: *Unity web player*, *Google Native Client* i nekad popularni Flash, ali je on izuzet iz podrške počev od Unity verzije 4.
- **Konzole za igru** - Ovde su uključene sve najnovije popularne konzole kao što su PlayStation 4, Xbox 360, Wii U i druge.

Takođe, od skoro Unity pruža podršku i za Virtuelnu realnost (skr. **VR**), a platforme koje su za sad podržane su *Native Oculus Rift*, *Gear VR* i *Playstation VR*.

Sve platforme koje Unity podržava date su na slici 4.1.


Slika 4.1: Platforme koje podržava Unity

### 4.1.1 Jezici za kodiranje

Unity je omogućio programerima da koriste tri različita jezika pri razvoju svojih video igara. Tu se nalaze C#, Unity JavaScript (poznatiji kao UnityScript) i Boo koji je zasnovan na Pythonu. Svi ovi jezici imaju podršku za Mono, koji predstavlja Open sors

verziju Majkrosoftovog .NET skupa biblioteka što omogućava unakrsnu kompatibilnost za različite platforme.

#### 4.1.1.1 C#

C# predstavlja jedan od prvih izbora pri kodiranju igara u Unity razvojnom okruženju. Jezik je jako izražajan i dozvoljava programeru da ima potpunu i preciznu kontrolu nad svojim kodom. On podržava više paradigmi, kao što su na primer funkcionalna, objektno-orijentisana ili komponentna. C# koristi standardnu C sintaksu koja se još može naći u sličnom obliku kod C++ ili Jave. Sam naziv C#, dobio je po muzičkoj povisilici (#), ali u nekim tumačenjima znak # predstavlja i 4 plusa (što može da referiše na inkrement jezika C++).

Za njega se obično opredeljuju ljudi koji već imaju neko programersko iskustvo, dok se UnityScript uglavnom preporučuje početnicima.

#### 4.1.1.2 UnityScript

UnityScript je jezik nalik na JavaScript i najbolji je izbor za neiskusne početnike u oblasti. On je lak za učenje kao i za kucanje, jer veliki deo kastovanja tipova obavlja sam iza scene i dozvoljava korisniku da se prebacuje između dinamičkog i striktnog kucanja koda. UnityScript takođe podržava rad sa klasama, nasleđivanje kao i modifikatore pristupa iste kao u C#-u. Iako je generalno smatran za početnički jezik, postoji značajan broj poznatih programera koji ga koriste kao prvu opciju pri razvoju.

#### 4.1.1.3 Boo

Boo poseduje sintaksu koja najviše podseća na Python, ali je struktuiran slično kao UnityScript. Mali broj ljudi zapravo koristi Boo, pa pronalaženje pomoći i podrške unutar zajednice može biti problematično.

U nastavku sledi isti deo koda napisan u sva tri različita jezika. Primetićemo da C# zahteva najviše pisanja koda, UnityScript nema uključivanje biblioteka koje se koriste, a Boo se služi uvlačenjem koda za određivanje blokova (ne koriste se vitičaste zagrade, kao ni ";"). U ovom radu je kao jezik za kodiranje korišćen isključivo C#.

C#:

```
using UnityEngine;
using System.Collections;
public class Example : MonoBehaviour {
    void Update() {
        //Transliramo objekat za koji je zakacen skript
        transform.Translate(Vector3.forward * Time.deltaTime);
        transform.Translate(Vector3.up * Time.deltaTime, Space.World);
    }
}
```

Skripta 4.1: Primer koda u C#-u

UnityScript:

```
function Update() {
    //Transliramo objekat za koji je zakacen skript
    transform.Translate(Vector3.forward * Time.deltaTime);
    transform.Translate(Vector3.up * Time.deltaTime, Space.World);
}
```

Skripta 4.2: Primer koda u UnityScript-u

Boo:

```
import UnityEngine
import System.Collections

public class Example(MonoBehaviour):
        def Update() as void:
        //Transliramo objekat za koji je zakacen skript
        transform.Translate((Vector3.forward * Time.deltaTime))
        transform.Translate((Vector3.up * Time.deltaTime), Space.World)
```

Skripta 4.3: Primer koda u Boo-u

### 4.1.2 Prodavnica komponenti (eng. *Asset store*)

Desetog novembra 2010. godine Unity Technologies je lansirao platformu za razmenu sadržaja za Unity korisnike i nazvao je "Asset store". Cela platforma je pored Web interfejsa takođe integrisana i u razvojno okruženje što dodatno olakšava nabavljanje nekih

standardizovanih komponenti koje se u igrama koriste u manje-više istoj formi. Jedan od ciljeva ovog rada je pravljenje nekoliko komponenti koje se mogu koristiti u velikom broju projekata u istoj ili malo izmenjenoj formi, kao i objavljivanje tih komponenti u svrhu olakšavanja rada programerima koji ih smatraju upotrebljivim. Na ovaj način autor pokušava da doprinese globalnom širenju i popularizaciji opisanog alata.

Sve komponente koje se nalaze u prodavnici su spremne za uključenje u projekat i momentalno korišćenje bez nekih posebnih podešavanja. Veliki broj komponenti u prodavnici su zapravo besplatne, ali Unity Technologies ohrabruje programere da prave kompleksne komponente koje se mogu prodavati za realan novac (Unity zadržava 30% od cene, a ostatak ide autoru komponente).

Kad se govori o komponentama, one mogu biti različiti delovi bilo koje igre, uključujući 3D modele, dvodimenzionu grafiku, teksture, muziku, delove izvornog koda pa čak i kompletne demo igre.

Ovakav vid razmene i kolaboracije među projektantima igara doprinosi sve većem jačanju zajednice i ekspanziji ove grane industrije.

### 4.1.3 Zapažena izdanja

Unity se osim u manjim nezavisnim timovima koristi i u velikim korporacijama koje izbacuju ozbiljne naslove. Neke od igara koje su upotrebljivost Unitija prikazale u najboljem svetlu su navedene u nastavku:

- **Temple Run** (Imangi Studios, 2011) - Igra napravljena za mobilne platforme u formi beskonačnog trčanja i savladavanja prepreka. Inovativna mehanika prilagođena mobilnim uređajima naglo je postala hit i uz prisustvo još nekoliko igara sličnog tipa dovela do nastajanja novog pod-žanra mobilnih igara "Beskonačna jurnjava" (eng. *Endless runner*).
- **Hearthstone: Heroes of Warcraft** (Blizzard Entertainment, 2014) - Kartična igra na tabli izašla iz Blizzardove kuhinje osvojila je srca više miliona igrača širom sveta. Uz konstantu nadogradnju njihove brojke se samo povećavaju, a Unity se pokazao kao alat koji je sposoban da donese odličan kvalitet uz optimalne performanse i minimalnu potrošnju hardverskih resursa.


Slika 4.2: Hearthstone: Heroes of Warcraft

- **Tiger Woods PGA Tour Online** (EA Sports, 2011) - U ovoj igri smo mogli da vidimo svu raskoš novog Unity-a ogledanu kroz odlično modelovane golf terene sa realnim ponašanjem vode i vrhunskom fizikom ponašanja golf loptice.
- **Wasteland 2** (Obsidian Entertainment, 2014) - Postapokaliptični RPG[1] sa bogatom izometrijskom grafikom i mnoštvom sadržaja, koji je direktni nastavak igre **Wasteland** iz 1988. godine. U potpunosti je finansiran preko Kickstartera[2].

### 4.1.4 Radno okruženje

Razvojno okruženje Unity 3D alata se sastoji iz dva ključna dela: Unity editora i MonoDevelop IDE-a[3]. U nastavku je dat osvrt na obe komponente.

#### 4.1.4.1 Unity editor

Unity editor predstavlja softverski paket koji pruža mogućnosti za organizovanje i upravljanje svim resursima neophodnim za razvoj jedne video igre. U njemu se nalaze alati za pravljenje scena, inspektori objekata, alati za animaciju (2D i 3D), podrška za rad sa

[1] RPG (eng. Role Playing Game) - žanr igara gde se akcenat stavlja na detaljan razvoj likova unutar igre prema raznim parametrima.

[2] Kickstarter - globalna platforma koja koristi finansiranje iz gomile za finansiranje kreativnih projekata svih vrsta.

[3] IDE (eng. Integrated Development Environment) predstavlja integrisano razvojno okruženje koje je realizovano kao softverska aplikacija koja pruža sveobuhvatne pogodnosti za programere.

mašinama stanja, alati za rad sa zvukom, simulator fizike, podrška za rad sa kamerama i još mnogo toga.

U njega je takođe integrisan i omotač za pokretanje napravljene igre, koji je prilagodljiv u smislu različitih platformi za koje se igra pravi. On omogućava i izvršavanje igre sličicu po sličicu što dizajnerima i programerima daje mogućnost za nadgledanje najsitnijih detalja svoje igre.

Takođe, i prodavnica komponenti je uključena u celokupno okruženje, pa je biranje i preuzimanje komponenti (engl. Assets) olakšano. Na ovaj način probrana komponenta je odmah uključena u tekući projekat i na taj način spremna za upotrebu.

Pored svega navedenog, ovaj editor je jako udoban za rad. Sve sekcije i prozore (kojih ima mnogo) moguće je organizovati na potpuno proizvoljan način koji korisniku najviše odgovara. Unity editor je napravljen sa ciljem da u njemu rade svi ljudi odgovorni za nastanak jedne igre, a ne samo programeri. To naravno uključuje muzičare, dizajnere, 3D modelare, scenariste i druge.

#### 4.1.4.2 MonoDevelop

MonoDevelop je integrisano razvojno okruženje otvorenog koda koje se može pokretati na Linuksu, OS X-u i Windowsu. Ovo okruženje se najviše koristi pri razvoju projekata koji koriste Mono i .NET okvire za razvoj. On poseduje sve prednosti modernih razvojnih okruženja koje je mahom pozajmio od NetBeans-a i Microsoft Visual Studija. Tu se najviše izdvajaju automatsko kompletiranje koda, sistemi za upravljanje izvornim kodom (eng. source control), grafički korisnički interfejs, Web dizajner, dibager[4], praćenje promenljivih, itd.

On podržava veliki broj programskih jezika, među kojima se nalaze:

- C#
- C++
- C
- JavaScript

[4]eng. Debugger, softver koji pomaže programeru prilikom pronalaženja i otklanjanja grešaka.

- HTML
- CSS
- CLI
- F#
- Java
- Boo
- Visual Basic

Monodevelop je svoj razvoj počeo krajem 2003. godine, kada je nekoliko programera iz Mono zajednice počelo da prepravlja SharpDevelop, koji je u tom periodu bio jako dobar open-sors editor. Vremenom je MonoDevelop integrisan u ostatak Mono projekta i aktivno je održavan od strane Xamarin i Mono zajednice. Kasnije se pojavljuju i specijalizovane verzije MonoDevelopa, kao što je verzija za Unity ili Xamarin studio koji se koristi za razvoj mobilnih aplikacija.

### 4.1.5 Nvidia PhysX

Nvidia PhysX je modul za simuliranje fizike koji programerima olakšava posao oko pisanja sopstvenog koda koji definiše ponašanje objekata u smislu klasične mehanike. On radi u višenitnom modu izvršavanja i podržava mogućnosti kao što su: dinamika tvrdih i mekih tela, simulaciju beživotnih ljudskih tela [5], kontroleri likova, dinamika kretanja vozila, ponašanje animacijskih čestica (eng. particles), simulacija vodenih površina, garderobe, itd. PhysX koristi veliki broj pokretača igara među kojima se izdvajaju Unity, Unreal Engine, Torque, Vision i drugi. Takođe, neki od najpoznatijih naslova današnjice koriste ovaj vid simuliranja fizike, a među njima su: Batman: Arkham City, Borderlands 2, Need For Speed: Shift, Mafia II i ostali.

Trenutno ga održava kompanija Nvidia.

[5] engl. Rigidbody, koristi se kao zamena za nekadašnje animacije smrti ljudskih likova. Na modelovani ljudski skelet deluju samo sile gravitacije.

### 4.1.6 Rezime

Unity 3D predstavlja odličan alat za prve korake u svetu razvoja video igara, pre svega zbog svoje pristupačnosti i jako blage krive učenja. Najosnovniju igru je moguće napraviti i bez ikakvog stručnog programerskog znanja, a uz dovoljnu količinu entuzijazma. Ipak, kada se jednom uđe u ovaj svet, Unity je spreman da ponudi i veliki broj naprednih koncepata kojih se ne bi postideli ni komercijalni alati. Uzimajući u obzir sve navedeno, Unity predstavlja odličan izbor za pravljenje igara, što je pokazano i u ovom radu.

## 4.2 Visual studio

Za izradu praktičnog dela ovog rada korišćen je Visual Studio 2012, umesto standardnog MonoDevelopa koji je opisan ranije, pre svega zbog većih mogućnosti i udobnijeg rada. Kao i MonoDevelop, i Visual studio je integrisano razvojno okruženje koje uključuje editor teksta, dibager (eng. *debugger*), automatsko kompletiranje koda, ali poseduje i neke naprednije koncepte kao što je podrška za refaktorisanje, veliki broj vizuelnih dizajnera i podrška za integraciju sa velikim brojem alata za kontrolu verzija. On generalno može biti prilagođen radu sa bilo kojim programskim jezikom, ali se spisak ugrađenih jezika svodi na:

- C#
- C++
- C
- JavaScript
- HTML
- CSS
- CLI
- F#
- Visual Basic

Pre verzije 2015, komercijalne verzije ovog alata su bile dostupne besplatno za studente u okviru programa DreamSpark, ali su u tom periodu samo komercijalne verzije imale mogućnost instalacije dodataka. Počevši od verzije 2015, Microsoft je počeo da distribuira "Comunity" verziju sasvim besplatno za sve, a na nju se sada mogu instalirati dodaci.

Visual Studio trenutno predstavlja jedan on najboljih alata na tržištu i zbog toga je korišćen u izradi ovog rada.

# Glava 5

# Metodologija razvoja

U ovom poglavlju predstavljene su tehnike korišćene pri razvijanju igre, kao i samo komponentno programiranje i organizovanje koda u ponovno iskoristive module.

## 5.1 Pristup razvoju

### 5.1.1 Sistem komponenti u razvojnom okruženju Unity

Kao što je ranije navedeno, Unity je svojim korisnicima obezbedio jedinstvenu prodavnicu komponenti u kojoj se može naći veliki broj korisnih stvari za razvoj jedne igre. To uključuje grafiku, 3D modele, animacije, delove koda, muziku, zvučne efekte pa čak i kompletne igre koje se mogu koristiti za učenje. Poseban deo tu čine i komponente koje obavljaju jednu celinu posla kao što je simulacija kretanja, razni meniji, modeli ponašanja objekata ili kompleksne komponente koje uključuju integrisane skriptove, objekte i grafiku.

Ovakve komponente mogu se upotrebiti odmah nakon preuzimanja sa prodavnice i uz vrlo male adaptacije mogu programeru uštedeti dosta vremena.

Pošto je ovaj rad koncipiran tako da prikaže upotrebu različitih struktura podataka, nametnulo se rešenje da se njihova upotreba grupiše u zasebne komponente koje same za sebe obavljaju jedan celovit posao. Na ovaj način su dobijene sledeće komponente:

- **Komponenta za simulaciju inventara**

- **Komponenta sa logikom za klasičnu igru XO**
- **Komponenta za generisanje lavirinta**
- **Komponenta sa KD-stablom za upravljanje tačkama u 2D prostoru**
- **Komponenta za igru "Spoji 3" sa logikom i grafikom**

Sve napravljene komponente imaju prostu grafiku i celokupnu funkcionalnost. Na korisniku je samo da prilagodi grafička rešenja i komponenta se može koristiti. Sve ovo je urađeno u cilju popularizacije i doprinosa zajednici u ekspanziji, kao i podrške mladim kolegama u njihovim prvim koracima sa opisanim alatom.

### 5.1.2 Razvoj vođen funkcionalnostima

Razvoj vođen funkcionalnostima[1] predstavlja metodologiju razvoja softvera u kojoj se akcenat stavlja na funkcionalnosti koje aplikacija treba da obavlja. Ova metodologija se fokusira na iterativni i inkrementalni proces razvoja, a predstavlja jednu od mnogih agilnih metodologija za razvoj softvera.

Razvoj vođen funkcionalnostima prvi put je predstavljen 1999. godine u knjizi "Java Modelling in Color with UML". Prva primena je izvedena na velikom petnaestomesečnom projektu koji je uključivao 50 ljudi, a rađen je za jednu veliku singapursku banku. Zbog velike efikasnosti koju je pokazao, odmah nakon toga je primenjen na većem projektu koji je uključivao čak 250 ljudi.

Ova metodologija je zasnovana na procesu kratkih iteracija koji se sastoji od 5 osnovnih aktivnosti. Prve dve aktivnosti se obavljaju na početku i u njima se formira šira slika i globalna vizija o celom sistemu, dok se poslednje tri aktivnosti ponavljaju za svaku funkcionalnost posebno, što je predstavljeno na slici 5.1. Glavne karakteristike ovih aktivnosti su sledeće:

- **Planiranje modela**

  Projekat počinje detaljnom analizom obima sistema i njegovog konteksta. Nakon toga se kreiraju detaljni modeli domena za svaku oblast u malim grupama. Ovakvi modeli se progresivno stapaju u globalni model kroz proces razvoja.

[1] skraćeno *FDD* (eng. *Feature driven development*).

- **Pravljenje liste funkcionalnosti**

  Sakupljeno znanje iz prve faze ovde se koristi za formiranje liste funkcionalnosti, najčešće funkcionalnom dekompozicijom domena u interesne oblasti. Svaka od ovih oblasti sadrži aktivnosti, a zasebni koraci unutar svake aktivnosti predstavljaju bazu za formiranje funkcionalnosti. Funkcionalnost u ovom kontekstu predstavlja malu funkcionalnu celinu iz ugla korisnika i može se opisati u formi <akcija><rezultat><objekat> (npr. "Izračunaj ukupnu vrednost poena.") Jedna funkcionalnost ne bi trebalo da zahteva više od 2 nedelje rada i u tom slučaju je poželjno razbiti je na sitnije delove.

- **Planiranje po funkcionalnosti**

  Nakon što je lista funkcionalnosti spremna, sledeći korak je pravljenje razvojnog plana koji uključuje dodelu pojedinačnih funkcionalnosti ili seta funkcionalnosti programerima, najčešće u vidu klasa.

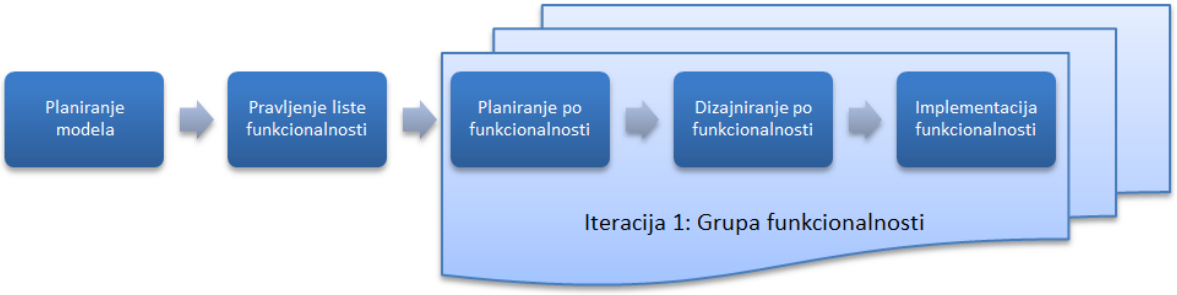
- **Dizajniranje po funkcionalnosti**

  Za svaku funkcionalnost pravi se zaseban dizajn. Glavni programer bira malu grupu funkcionalnosti koje će biti implementirane u vremenskom okviru od 2 nedelje. Zajedno sa ostalim programerima (vlasnicima funkcionalnosti) on pravi detaljne dijagrame sekvence za svaku od funkcionalnosti i upotpunjuje celokupni model. Nakon ovoga pravi se pregled klasa i metoda, a kada je sve završeno obavlja se obavezna kontrola celokupnog dizajna.

- **Implementacija funkcionalnosti**

  U ovom delu programeri pišu kod za svoje klase. Poželjna je upotreba testova jedinica koda koja treba da prethodi globalnoj inspekciji celog koda. Nakon što funkcionalnost prođe sve potrebne provere, spremna je da bude uključena u glavnu verziju aplikacije.

Ako se govori o ulogama učesnika u ovoj metodologiji, treba ih pomenuti nekoliko. Pre svega, tu je 6 primarnih uloga: Menadžer projekta, Glavni programer, Glavni arhitekta sistema, Menadžer razvoja, Vlasnik klase i Ekspert za domen. Jedna osoba može obavljati više različitih uloga. Programeri imaju vlasništvo nad određenim klasama, što dovodi do neophodne saradnje među njima kada je potrebno omogućiti funkcionalnost koja zahteva menjanje različitih klasa. Na ovaj način se podstiče saradnja unutar tima i veći stepen




Slika 5.1: Faze ciklusa razvoja vođenog funkcionalnostima

kontrole (nešto nalik programiranju u paru). Ovo je jedna od glavnih razlika u odnosu na ostale agilne metodologije gde nalazimo princip univerzalnog vlasništva nad klasama.

Ovoj listi uloga mogu se pridodati još neke koje upotpunjuju raspodelu poslova: Menadžer domena, Menadžer puštanja u promet (eng. *Release manager*), Specijalista za jezike, Inženjer za integraciju, Administrator sistema, Tester, Pisac tehničke dokumentacije itd.

Pošto je autor bio sam svoj tim, ova metodologija nije mogla biti primenjena u punoj meri, ali su osnovni koncepti svakako iskorišćeni. Metodologija je predstavljena prvenstveno zbog svoje primenljivosti na razvoj igara. Posebno je bilo korisno koristiti korake definisane ovom metodologijom zbog organizacije rada, tj. njegove komponentne strukture koja se lako mogla dekomponovati na funkcionalnosti i zasebne celine. Takođe, metodologija je pogodna i za korišćenje u manjim timovima od nekoliko ljudi gde bi, na primer, grafički dizajner delio vlasništvo nad klasama sa programerima. Opet, na ovaj način svi učesnici u razvoju će imati uvid u sve delove sistema.

Tehnike za unapređenje ove razvojne metodologije, kao i više detalja, mogu se naći u [12].

## 5.2 Komponentno programiranje

Komponentno programiranje predstavlja vid organizovanja koda u slabo spregnute celine koje predstavljaju zasebene entitete i obavljaju određene celine posla. Posebno je pogodno za ponovno iskorišćavanje napravljenih komponenti, agregiranje komponenti u veće celine ili kao obične kontejnere. Kada se kreira novi entitet (često ga možemo naći i pod nazivom *GameObject*), funkcionalnosti se entitetu dodaju preko komponenti koje su često napravljene univerzalno. Te komponente mogu biti elementi grafike, skriptovi, simulacija fizike, zvukovi, 3D modeli i još mnogo toga. Entiteti sami za sebe nisu ništa drugo

do prosti kontejneri u kojima su sadržane sve neophodne komponente za funkcionisanje određenog entiteta. Više detalja o komponentnom programiranju i dizajniranju igara kroz komponentni način razmišljanja može se naći u [13] [14].

Ako se entitetu doda komponenta za prikazivanje (eng. *Rendering*) 3D modela, to će omogućiti da se vidi konkretan entitet na sceni. Naravno, neke komponente zahtevaju neke druge da bi mogle da funkcionišu, ali se uglavnom ide na to da komponente među sobom budu slabo spregnute.

Što se tiče Unity razvojnog okruženja, prilikom pravljenja novog entiteta automatski se pravi *Transform* komponenta koja poseduje koordinate pozicije entiteta u prostoru scene.

### 5.2.1 Razlike u odnosu na OOP

Objektno orijentisana paradigma zasnovana je na modelovanju objekata nalik na ono što predstavljaju u realnom svetu, tj. da se učini lakšim za čitanje ljudima (što programerima to i krajnjim korisnicima).

Sa druge strane, komponentno orijentisani razvoj softvera ne predlaže ništa slično i umesto toga savetuje da bi programeri trebalo da konstruišu softver spajanjem ranije napravljenih komponenti - slično kao u elektrotehnici ili mehanici. Ovo je takođe poželjno i zbog lakog održavanja i prepravljanja komponenti, a ponekad i kompletnom zamenom drugom komponentom.

Ipak, komponentno orijentisani razvoj ne isključuje korišćenje objektno orijentisanih principa. Zasebni skriptovi mogu u okviru svoje enkapsulacije imati hijerarhiju klasa, interfejse i sve druge prednosti OO jezika. Najbolje stvari ipak mogu se dobiti kombinacijom ova dva principa, što se najčešće i radi.

## 5.3 Uzorci za projektovanje

U izradi praktičnog dela rada korišćene su i tehnike programiranja zasnovane na uzorcima za projektovanje (eng. *Design patterns*) koji su jako zastupljeni u modernom razvoju softvera. Implementirani su uzorci "**Posmatrač**" i "**Grupisanje objekata**" koji su doprineli efikasnijem korišćenju resursa platforme i boljoj organizaciji koda. Detaljan prikaz njihove implementacije i problema koje rešavaju predstavljen je u poglavlju 6.3

# Glava 6

# Detalji implementacije

Imlementacioni proces je radi lakše organizacije podeljen u nekoliko etapa. Prvo je određen podskup struktura podataka koje će biti implementirane, a zatim osmišljene i funkcionalnosti u kojima će se te strukture koristiti. Nakon toga je pristupljeno razvoju svake od 5 različitih komponenti gde je svaka od njih imala određene specifičnosti i koristila različite mogućnosti Unity razvojnog okruženja. Kada su sve komponente završene i dovedene na zadovoljavajući nivo funkcionalnosti usledila je faza integracije. U ovoj fazi prvo je trebalo osmisliti kompletan koncept konačne igre i mehanike igranja (*eng. Gameplay*), što se još naziva i dizajniranje igre (eng. Game design). Kada je kreirana vizija o celokupnom izgledu prvo je prikupljen sav grafički materijal neophodan za formiranje vizuelnog stila igre, a zatim pristupljeno samom procesu integracije koda. Detaljan prikaz svih ovih koraka dat je u nastavku.

## 6.1 Komponente za izdvojene funkcionalnosti

Sistem paketa koji koristi Unity omogućava da se sklopljena celina izvozi kao zapakovani fajl i zatim uvozi u bilo koji projekat gde nam je potrebna. Pošto se paketi uvoze sa istom hijerarhijom direktorijuma iz koje su izvezeni potrebno je ispoštovati određeni način pravljenja direktorijuma unutar svakog zasebnog paketa, da bi se struktura očuvala i nakon uvoženja svakog od paketa. Poznato je da u svakom Unity projektu koren svih direktorijuma predstavlja direktorijum **Assets**. Zbog toga je svaki od paketa organizovan tako da u **Assets** direktorijumu postoji još jedan direktorijum sa konkretnim imenom paketa, a

onda se u njemu nalaze standardni direktorijumi za skriptove, grafiku, predefinisane objekte i tako dalje. Na ovaj način se prilikom uvoza paketa formiraju zasebni direktorijumi za svakog od njih što poboljšava organizacija projekta.

U ovoj sekciji je predstavljena implementacija 5 različitih komponenti od kojih svaka implementira barem po jednu bitnu strukturu podataka. Te komponente su:

- Komponenta za simulaciju inventara
- Komponenta sa logikom za klasičnu igru XO
- Komponenta za generisanje lavirinta
- Komponenta sa KD-stablom za upravljanje tačkama u 2D prostoru
- Komponenta za igru "Spoji 3" sa logikom i grafikom

U nastavku je predstavljena implementacija svake od ovih komponenti.

### 6.1.1 Komponenta za simulaciju inventara

Inventar predstavlja neizbežan deo igara danas, pre svega zbog velikog broja elemenata kojima treba upravljati u svakom trenutku. U ovom radu napravljena je jedna verzija koja koristi prikaz elemenata pomoću mreže.

U osnovi se za kolekciju elemenata inventara koristi lista. Konkretna dvostruko povezana lista nije implementirana od nule, već je iskorišćena lista koju nudi .NET okvir (*eng. Framework*), a ona se zapravo ponaša kao dvostruko povezana lista. Elementi liste su za svrhe prezentacije predstavljeni u mreži, a njihova pozicija se određuje prostim računom u zavisnosti od broja kolona i redova koji se koriste za prezentaciju.

Prvo je napravljena klasa **Item** koja reprezentuje jedan element koji će biti deo kolekcije, a ona poseduje odgovarajuća polja za opis, cenu, kategoriju i tako dalje. Pored njega ubačena je i **ItemDatabase** klasa koja se koristi kao katalog različitih elemenata koji se koriste u igri. Pri svakom instanciranju nekog elementa, to će se uraditi u odnosu na podatke iz ove klase.

```
using UnityEngine;
using System.Collections;

public class Item
{
  public int? itemID;
  public string itemName;
  public string itemDescription;
  public ItemRarity itemRarity;
  public ItemType itemType;
  public int price;
  public Texture2D itemIcon;

  public enum ItemRarity
  {
    Common,
    Rare,
    Legendary
  }

  public enum ItemType
  {
    Tool,
    Consumable,
    Normal
  }

  public Item()
  {
    this.itemIcon = Resources.Load<Texture2D>("ItemIcons/" + "empty");
  }

  public Item(int id, string name, string description, ItemRarity
   rarity, ItemType type, int price)
  {
    this.itemID = id;
    this.itemName = name;
    this.itemDescription = description;
    this.itemRarity = rarity;
    this.itemType = type;
    this.price = price;
    this.itemIcon = Resources.Load<Texture2D>("ItemIcons/" + name);
```

```
  }
}
```

Skripta 6.1: Izvorni kod klase Item

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

public class ItemDatabase : MonoBehaviour
{

  public List<Item> items = new List<Item>();

  void Start()
  {
    items.Add(new Item(1, "Bread", "Tasty little bread",
   Item.ItemRarity.Common, Item.ItemType.Consumable, 1));
    items.Add(new Item(2, "Carrot", "Ordinary carrot",
   Item.ItemRarity.Common, Item.ItemType.Consumable, 3));
    items.Add(new Item(3, "Cheese", "It's from Swittzerland! Mouses
   also love it", Item.ItemRarity.Rare, Item.ItemType.Consumable, 22));
    items.Add(new Item(4, "Mushroom", "Shitaki!", Item.ItemRarity.Rare,
   Item.ItemType.Consumable, 100));
    items.Add(new Item(5, "RawMeat", "Raw power",
   Item.ItemRarity.Legendary, Item.ItemType.Consumable, 300));
  }
}
```

Skripta 6.2: Izvorni kod klase ItemDatabase

I naravno, tu je klasa **Inventory** koja sadrži metode za ubacivanje elemenata i pretraživanje inventara. Ovde se takođe nalaze i metode za iscrtavanje inventara na ekranu, kao i podrška za pomeranje elemenata unutar inventara i njihovo korišćenje.

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

public class Inventory : MonoBehaviour
{
  private int slotsX = 6;
```

```
private int slotsY = 4;
public List<Item> inventory = new List<Item>();
public List<Item> slots = new List<Item>();
private ItemDatabase database;
private bool inventoryShown;

public GUISkin skin;
private bool draggingItem;
private Item draggedItem;
private int previousIndex;

private bool showTooltip;
private string tooltip;

void Awake()
{
  inventoryShown = false;
}

void Start()
{
  database = GameObject.FindGameObjectWithTag("ItemDatabase")
  .GetComponent<ItemDatabase>();

  for(int i=0; i < slotsX*slotsY; i++)
  {
    slots.Add(new Item());
    inventory.Add(new Item());
  }

  AddItem(1);
  AddItem(2);
  AddItem(3);
  AddItem(4);
  AddItem(5);
}

void Update()
{
  if(Input.GetKeyUp(KeyCode.I))
  {
    inventoryShown = !inventoryShown;
  }
```

```
  if(Input.GetKeyUp(KeyCode.Space) && inventoryShown)
  {
    AddItem(1);
  }
}

void OnGUI()
{

  tooltip = "";

  if(inventoryShown)
    DrawInventory();

  if(draggingItem)
  {
    GUI.DrawTexture(new Rect(Event.current.mousePosition.x,
 Event.current.mousePosition.y, 32, 32 ), draggedItem.itemIcon);
  }

  if(showTooltip)
  {
    GUI.Box(new Rect(Event.current.mousePosition.x + 20,
 Event.current.mousePosition.y, 150, 120 ), tooltip,
 skin.GetStyle("Tooltip"));
  }

}

void DrawInventory()
{
  Event e = Event.current;

  GUI.Box(new Rect(146,116, 36*(slotsX) + 4, 36*(slotsY) + 4), "");
  int i = 0;
  for(int y=0; y < slotsY; y++)
  {
    for(int x=0; x< slotsX; x++)
    {
      Rect slotRect = new Rect(150 + x*36, 120 + y*36, 32, 32);
      GUI.Box(slotRect, "", skin.GetStyle("slot"));
```

```
        slots[i] = inventory[i];

        if(slots[i].itemID != null)
        {
          GUI.DrawTexture(slotRect, inventory[i].itemIcon);

          if(slotRect.Contains(e.mousePosition))
          {
            tooltip = CreateTooltip(inventory[i]);
            showTooltip = true;

            if(e.button == 0 && e.type == EventType.mouseDrag &&
!draggingItem)
            {
                //Left mouse button
              draggingItem = true;

              previousIndex = i;
              draggedItem = slots[i];
              inventory[i] = new Item();
            }
            if(e.type == EventType.mouseUp && draggingItem)
            {
              inventory[previousIndex] = inventory[i];
              inventory[i] = draggedItem;
              draggingItem = false;
              draggedItem = null;
            }
          }
        }
        else
        {
          if(slotRect.Contains(e.mousePosition) )
          {
            if(e.type == EventType.mouseUp && draggingItem)
            {
              inventory[i] = draggedItem;
              draggingItem = false;
              draggedItem = null;
            }
          }
        }
```

```
        if(tooltip == "" || draggingItem)
        {
          showTooltip = false;
        }

        i++;
      }
    }

}

string CreateTooltip(Item item)
{
  tooltip = "<b><size=15>" + item.itemName + "</size></b>";
  tooltip += "\n\n" + item.itemDescription;
  tooltip += "\n\nValue: " + "<color=#ffff00>" + item.price + " Gold
 </color>";
  return tooltip;
}

void AddItem(int id)
{
  int i=0;
  for(; i<inventory.Count; i++)
  {
    if(inventory[i].itemID == null)
    {
      for(int j=0; j < database.items.Count; j++)
      {
        if(database.items[j].itemID == id)
          inventory[i] = database.items[j];
      }
      break;
    }

  }
  if(i == inventory.Count)
    Debug.Log("Inventory is full!");

}

bool InventoryContains(int id)
{
```

```
    for(int i=0; i<inventory.Count; i++)
    {
      if( inventory[i].itemID == id)
        return true;
    }
    return false;
  }
}
```

Skripta 6.3: Izvorni kod klase Inventory

Na skriptama 6.1.1, 6.1.1 i 6.1.1 su prikazane klase korišćene u implementaciji ove komponente.

### 6.1.2 Komponenta sa logikom za klasičnu igru XO

U ovoj komponenti implementiran je algoritam za klasičnu igru XO sa grafičkim interfejsom.

Opet je za prezentaciju na ekranu korišćena obična mreža, tj. višedimenzioni niz, a za određivanje poteza u svakom trenutku Minimaks stablo opisano u poglavlju 3.3.2. Algoritam u zavisnosti od toga koji je igrač na potezu u kom nivou stabla određuje zbirove i vraća minimalnu ili maksimalnu vrednost. Podešavanjem dubine do koje će algoritam ići u pretrazi može se podešavati težina igre, što znači da ako podesimo obilazak do maksimalne dubine protivnički igrač ne može izgubiti partiju već samo izvući nerešeno. Pored klasičnog algoritma, ovde je tebalo voditi računa i o okupiranim poljima i sve to prikazivati na ekranu, kao i obezbediti interakciju korisnika sa poljima u koja se simboli unose.

### 6.1.3 Komponenta za generisanje lavirinta

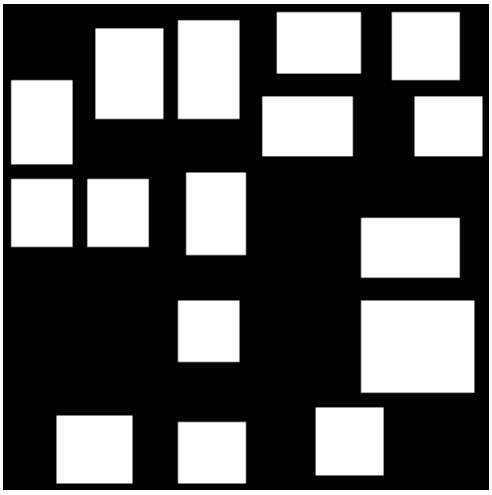
Proceduralno generisani nivoi u igrama daju posebnu dimenziju igrivosti, pre svega zato što igra nikad neće biti ista. Ovakva raznovrsnost je prilično popularna i svaki vid proceduralnog generisanja nivoa je uvek poželjan.

Lavirint se generiše pomoću Q-stabla, tako što u svakom listu stabla pozicioniramo po jednu sobu. Pošto sobe moraju međusobno biti povezane, listove povezujemo među sobom

tako što susede na istom nivou povezujemo jedne sa drugima, a listove koji nemaju suseda na istom nivou povezujemo sa najlevljim listom u susedskom podstablu njegovog oca. Na ovaj način dobijamo povezane sobe, koje nisu povezane svaka sa svakom, ali se iz svake sobe može preći u neku drugu određenim putem.

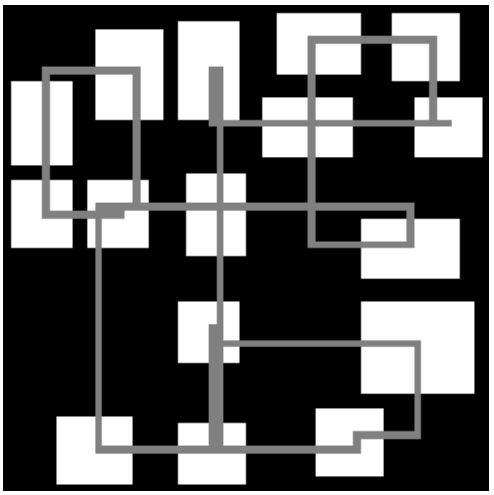
Da bi se ovo realizovalo korišćeno je pravljenje pikselizovanih slika, gde je jedan piksel predstavljao vrednost zida ili prolaza. Ove slike su generisane direktno iz Q-stabla, a konkretan nivo je nakon toga generisan iz konačno dobijene slike (6.4) na kojoj se nalaze sobe, prolazi i zidovi (tamo gde je piksel označavao zid postavljen je visoki kvadar, a tamo gde je označavao prolaz kocka sa teksturom poda). Jedan primer se može videti na slikama 6.1, 6.2, 6.3 i 6.4.

Slika 6.1: Generisano Q-stablo u svojoj grafičkoj reprezentaciji

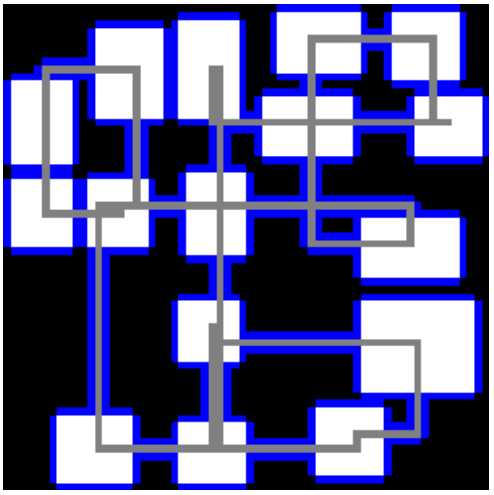


Slika 6.2: Generisane sobe unutar listova Q-stabla

Na ovaj način dobijen je uvek drugačije generisan lavirint kroz koji se igrač može kretati ili u 3 ili u 2 dimenzije, u zavisnosti od toga kako je kamera postavljena. Ovo je i



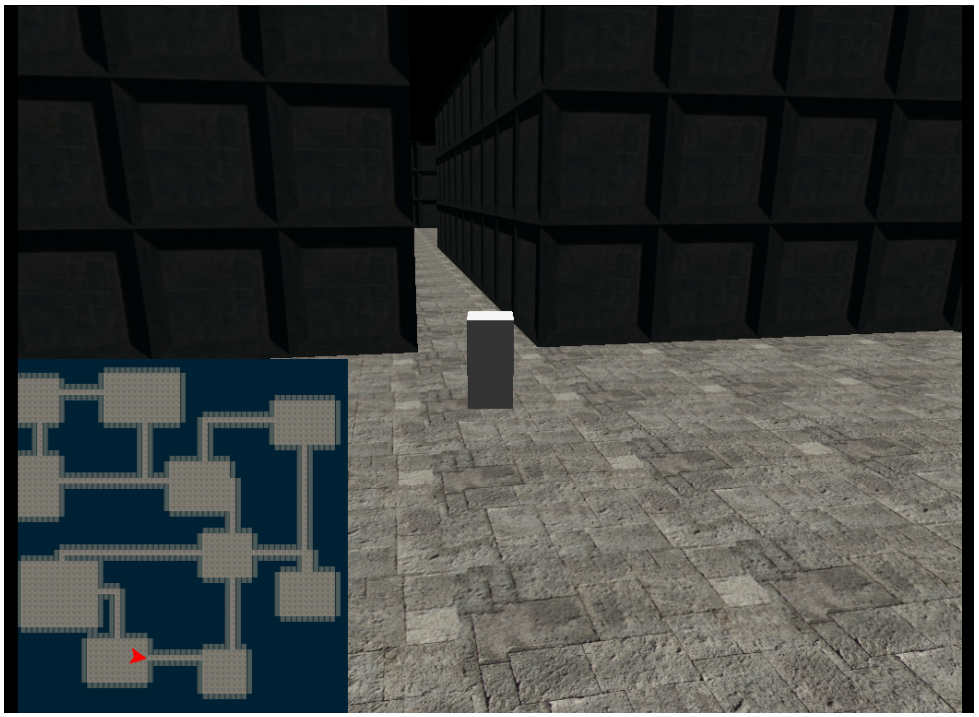
Slika 6.3: Generisani prolazi između soba



Slika 6.4: Zidovi generisani na ivicama soba i prolaza

demonstrirano prikazom mini mape u donjem levom uglu ekrana kao što se vidi na slici 6.5.

Pošto se na ovaj način generiše veliki broj sitnih elemenata za zidove i prolaze, iskorišćena je gotova Unity komponenta "MeshCombineUtility" [15] koja kombinuje veliki broj malih elemenata u jedan veliki, sa kojim je mnogo lakše raditi.



Slika 6.5: Generisani lavirint sa dve pozicije kamere

### 6.1.4 Komponenta sa KD-stablom za upravljanje tačkama u 2D prostoru

Ova komponenta je zamišljena kao osnova za igru odbrambene strategije (*eng. Tower defence*), što dovodi do toga da u svakom trenutku treba voditi računa o velikom broju pokretnih meta koje terba pogoditi. U tu svrhu iskorišćeno je KD-stablo koje pokazuje zavidne performanse na polju pronalaženja najbližih suseda, što je za ovu komponentu krucijalno. KD-stablo je detaljno objašnjeno u odeljku 3.3.5, a konkretna implementacija ne odstupa od tog objašnjenja. Implementirane su metode za nalaženje najbližeg suseda, kao i metoda za nalaženje $n$ najbližih suseda. Posebno je dobro što KD-stablo ovu pretragu obavlja brzo, jer je pri svakom ispaljivanju jednog ili više projektila potrebno preurediti stablo aktuelnim pozicijama protivnika, a zatim i pronaći $n$ najbližih.

Takođe, u ovoj komponenti su implementirani uzorci za projektovanje "Posmatrač" i "Formiranje grupa objekata" koji su detaljno predstavljeni u poglavlju 6.3. "Posmatrač" se koristi za obaveštavanje protivnika o nivou naoružanja koje poseduje kula, da bi protivnici, u slučaju da naoružanje dođe do nivoa 3, mogili da ubrzaju svoje kretanje i postanu opasniji. "Formiranje grupa objekata" je korisno za instanciranje projektila uz efikasno iskorišćenje memorije, pa je za to iskorišćeno i ovde.

### 6.1.5 Komponenta za igru "Spoji 3" sa logikom i grafikom

Nakon velikog uspeha i popularnosti "Spoji 3" žanra, naročito kroz naslove "Bejeweled" i "Candy Crush Saga", ovaj tip igre je počeo da se ubacuje i u razne druge žanrove. Klasičan primer je igra "Puzzle Quest" koja kombinuje RPG (*eng. Role Playing Game*) mehaniku sa već oprobanim "Spoji 3" konceptom.

Na tom talasu kreirana je i ova komponenta, naravno sa ciljem da se lako integriše u neki drugi žanr, a po potrebi i da posluži kao igra za sebe.

Ovde je korišćen dvodimenzioni niz i implementirana je logika za zamenu dva elementa i proveru okolnih elemenata na poklapanje. Za postizanje ovoga bilo je potrebno prepoznati nekoliko šablona u kojima se mogu javiti spojeni elementi, a zatim pametnim proverama kroz dvodimenzioni niz ovi elementi se uklanjaju, transliraju na dole, a u niz se ubacuju novi. Ako se javi slučaj da su spojena bar 4 ista elementa, na njihovom mestu se generiše bonus element te boje koji u kombinaciji sa drugim elementima te osnovne boje uništava ceo red ili kolonu (u zavisnosti od toga kakvo je poklapanje napravljeno).

Ovde je takođe iskorišćena komponenta "GoKit" [16] koja se koristi za glatke animacije elemenata koji se kreću između dve pozicije.

Pored svega ovoga implementirana je i logika koja u slučaju da igrač određeni vremenski period ne odigra potez prikaže pomoć u vidu treptajućih elemenata koji se mogu sklopiti.

Pošto kod ove komponente obiluje različitim parametrima, napravljen je poseban skript koji sadrži sve konstante da bi se prilagođavanje različitim scenarijima odvijalo lako.

### 6.1.6 Izvoz gotovih komponenti

Nakon što je implementacija svih komponenti završena, urađen je izvoz u paketne datoteke koje imaju ekstenziju **.unitypackage**. Ovako napravljene datoteke se pomoću opcije **Asstets -> Import Package -> Custom Package** lako uvoze u željeni projekat.

Takođe, ovako zapakovane komponente lake su za distribuciju jer zauzimaju jako malo memorije (ako u komponentu nije uključena grafika, veličina ne prelazi 1MB).

## 6.2 Implementacija igre

Nakon što su sve komponente završene i spremne za korišćenje potrebno je uraditi završni deo koji predstavlja igru kao celinu. Pre konkretne implementacije neophodno je osmisliti način na koji će se igra igrati (*eng. Gameplay*), što je predstavljeno u sledećoj celini.

### 6.2.1 Ideja igrivosti

S obzirom na to da je većina komponenti napravljena sa dvodimenzionim pristupom u osnovi, prirodan sled događaja je da i cela igra bude u dve dimenzije. Igra je zamišljena iz dva dela gde se u prvom delu svodi na igranje igre "Spoji 3", a u drugom delu na istraživanje automatski generisanog lavirinta uz propratne sadržaje.

U prvom delu, igrač pokušava da napravi što bolji rezultat i sebi kreira što veći broj odbrambenih jedinica (*eng. Turret*) koje će poneti sa sobom u istraživanje lavirinta. Krucijalno je da ovaj rezultat bude što bolji, jer time igrač sam sebi poboljšava šanse za preživljavanje u sledećem nivou. Kada igrač odluči da su mu zalihe dovoljne on kreće u istraživanje.

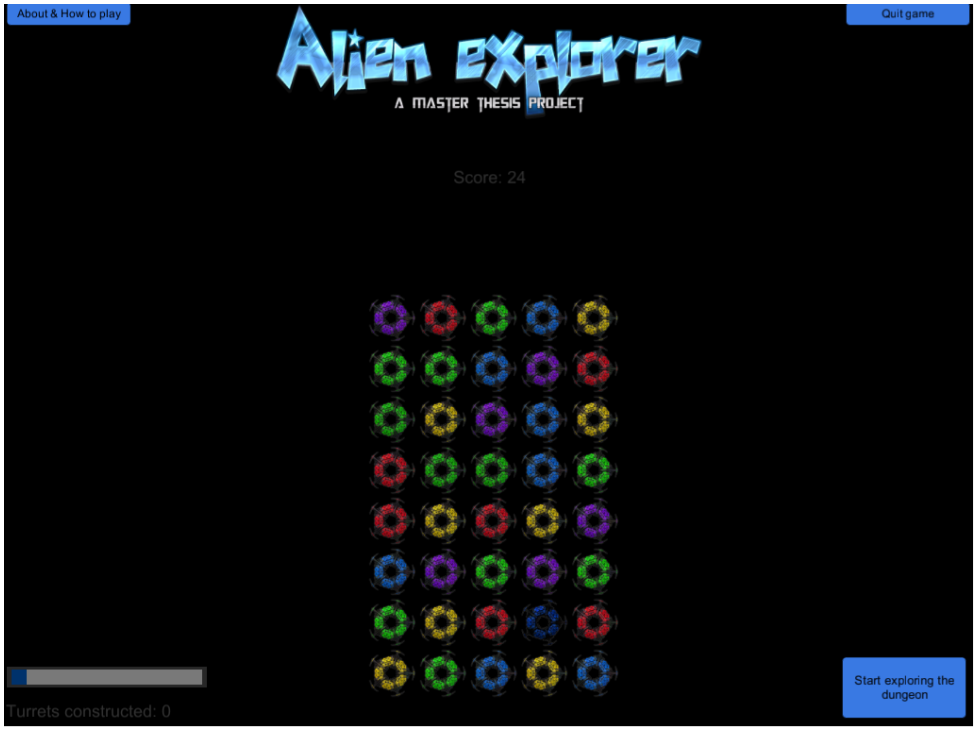
Pored napravljenih odbrambenih jedinica koje prenosi iz prvog nivoa, u svom inventaru igrač ima još nekoliko elemenata koji mu poboljšavaju šanse za preživljavanje. To su paket za popravku vozila (koji se koristi kao dopuna operativne energije) i paket za kreiranje nove defanzivne jedinice, koji jednom kada se upotrebi prikazuje igraču tablu za igru XO, gde igrač mora izvući minimum nerešen rezultat da bi mu odbrambena jedinica bila dodata u inventar. Igrač na ovom nivou ima za cilj da sakupi sve minerale i ako to učini, vraća se na prethodni nivo gde ponovo ima mogućnost da krene u istraživanje novog lavirinta i još više uveća svoje poene.

Naravno, lavirintom lutaju neprijateljske jedinice koje su u konstantnoj potrazi za igračem i trude se da ga unište. Igrač se protiv njih bori postavljanjem odbrambenih jedinica koje ispaljuju projektile na protivnike, ali su takođe ranjive i već posle nekoliko kontakata sa protivnikom postaju neupotrebljive. Igračev cilj je da ostvari što bolji rezultat u vidu ukupnog broja poena. Ovi poeni se dobijaju za svaku pozitivnu akciju u igri, nevezano za to na kom nivou je akcija obavljena. Kada se igračeva operativna energija smanji na nulu igra je završena, ali je opet moguće resetovati postavke i krenuti u novo istraživanje.

### 6.2.2 Integracija

Ispostavlja se da je proces integracije najlakši korak u celokupnom procesu razvoja ove igre. Nakon uvoza svih 5 komponenti napravljeni su dodatni direktorijumi za scene, skripte i grafiku. Kreirane su četiri nove scene i to:

- **Uvodna scena** na koju je pozicionirana "Spoji 3" igra. Na ovu scenu je dodat parametar za računanje poena kao i skala za kreiranje odbrambenih jedinica. Na ovoj sceni se nalazi dugme koje pokreće sledeću scenu, a to je nivo sa proceduralno generisanim lavirintom. Uvodna scena data je na slici 6.6




Slika 6.6: Uvodni ekran igre

- **Scena sa proceduralno generisanim lavirintom** u koju je ubačen najveći deo komponenti. Osnovu za ovu scenu predstavlja proceduralno generisan lavirint sa igračem na kog je zakačen modul za simulaciju kretanja (ovaj modul je iskorišćen gotov, napravljen je od strane Unity zajednice). Pored toga, ovde je repliciran ceo inventar i igra XO, a odbrambene jedinice se ponašaju na isti način kao u ranije napravljenoj komponenti. Sve ovo je lako postignuto prostim premeštanjem objekata sa scena unutar svake komponente na željenu scenu.

Dodatno, ovde je implementirana i skala operativne energije igračevog vozila i prikaz globalnog rezultata.

Jedan kadar ove scene je predstavljen na slici 6.7


Slika 6.7: Nivo sa istraživanjem lavirinta

- **Scena za kraj igre** koja nema nikakve specifičnosti osim što se iz nje može napustiti ili restartovati igra.
- **Scena o osnovnim informacijama** gde su napisane osnovne stvari u vezi sa projektom, kao i uputstvo za igranje igre.

Izmene koje su vršene na komponentama su mahom kozmetičke prirode što uključuje upotrebu novih grafičkih rešenja koja se uklapaju u vizuelni kontekst cele igre. Većina vizuelnog sadržaja je preuzeta besplatno sa sajta [17].

Bitno je naglasiti da korišćene komponente inicijalno nemaju nikakvu međusobnu spregnutost, što je dobro, ali za normalno funkcionisanje igre one ipak moraju biti spregnute. Zbog toga su klase i skriptovi u inventaru i igri "Spoji 3" malo izmenjeni da bi se lakše vodilo računa o globalnom rezultatu, elementima koji se pakuju u inventar, a zatim koriste iz njega i tako dalje. Ipak, ove izmene su jako sitne, a i celokupna poenta napravljenih modula je da ih svako može prilagođavati svojim potrebama bez ikakvih posledica.

Na kraju, kada je celokupan posao obavljen, urađeno je pravljenje izvršne verzije igre u **.exe** formatu koja je spremna za distribuciju na druge Windows mašine. Pored toga, igru je moguće isporučiti i za Linux platformu bez ikakvih dodatnih podešavanja, dok bi isporuka za neku mobilnu platformu iziskivala samo prepravke u vezi sa kontrolama (prilagođavanje na kotrolisanje dodirom).

Na ovaj način je zaokružen jedan celokupan softverski projekat koji je prošao kroz sve faze, uključujući planiranje, dizajniranje, implementaciju, testiranje i isporuku.

## 6.3 Netrivijalni problemi u razvoju

Kroz ceo postupak razvoja igre pojavili su se neki netrivijalni problemi za koje je potreban poseban mehanizam za efikasno rešavanje. U ovom odeljku predstavljeni su neki od tih problema uz analizu njihovog rešenja.

### 6.3.1 Instanciranje objekata

Instanciranje objekata na sceni je jedna od osnovnih akcija koje se obavljaju u toku izvršavanja igre. Takođe, instancirani objekti moraju se uništavati da scene ne postanu pretrpane. Ono što se može javiti kao problem jeste instanciranje jako velikog broja objekata u malom vremenskom intervalu i njihovo odgovarajuće uništavanje. Na ovaj način upošljava se veliki broj resursa platforme na kojoj se igra izvršava, pa dok na desktop računaru i nećemo primetiti veliki pad u performansama, na mobilnom uređaju se mogu javiti značajna usporenja zbog ograničene porcesorske snage i količine memorije. U ovakvim slučajevima sakupljač otpada (*eng. garbage collector*) mora da radi obiman posao što se značajno odražava na performanse. Iz ovih razloga potreban nam je mehanizam za efikasno upravljanje objektima na sceni i štednju memorije na neki način. Jedan takav mehanizam koji se može koristiti je i **formiranje grupa objekata** (*eng. Object pooling*).

#### 6.3.1.1 Uzorak za projektovanje "Formiranje grupa objekata"

Formiranje grupa istorodnih objekata za instanciranje je tehnika optimizacije i uzorak za projektovanje (*eng. Design pattern*) koja poboljšava upravljanje memorijom i optimizuje memorijski prostor. Ona se koristi tako što umesto da svaki put kada nam je potreban

neki objekat vršimo njegovo instanciranje iz kolekcije gotovih objekata, mi celokupno instanciranje vršimo na učitavanju scene. Broj objekata koji će biti instancirani u kolekciji određujemo u skladu sa potrebama, ali takođe možemo napraviti i dinamičko instanciranje kada je potrebno. Na ovaj način možemo uzeti već gotov objekat iz kolekcije i samo ga podesiti kao aktivan bez dodatnog trošenja resursa i nastaviti sa normalnim izvršavanjem igre. Kada objekat završi svoj životni ciklus na sceni, umesto da ga uništimo, samo ćemo ga vratiti u kolekciju gde će on čekati na ponovno instanciranje.

Ovu tehniku je poželjno koristiti na mestima gde imamo generisanje velikog broja istih objekata čiji je životni ciklus kratak. To mogu biti elementi od kojih se proceduralno izgrađuju trkačke staze, razni novčići, projektili i slično. Tehnika može biti implementirana u konretnom objektu koji vrši instanciranje, ali pošto se tehnika intenzivno koristi u modernom razvoju poželjno je napraviti zasebnu klasu i objekat koji će biti zadužen za upravljanje objektima u grupi.

U ovom projektu koristi se ispaljivanje projektila (implementacija modula sa KD-stablom) pa je to idealno mesto da se primeni ovaj uzorak za projektovanje.

U narednom odeljku je prikazana implementacija klase koja daje mehanizam za rad sa već instanciranim objektima.

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

public class ObjectPoolerScript : MonoBehaviour
{

    public static ObjectPoolerScript currentObjectPoolerScript;
    public GameObject pooledObject;
    public int pooledAmount = 10;
    public bool willGrow = true;

    List<GameObject> pooledObjects;

    void Awake()
    {
        currentObjectPoolerScript = this;
    }

    void Start()
```

```
    {
        pooledObjects = new List<GameObject>();

        for (int i = 0; i < pooledAmount; i++)
        {
            GameObject obj = (GameObject)Instantiate(pooledObject);
            obj.SetActive(false);
            pooledObjects.Add(obj);
        }
    }

    public GameObject getPooledObject()
    {
        for (int i = 0; i < pooledObjects.Count; i++)
        {
            if (!pooledObjects[i].activeInHierarchy)
            {
                return pooledObjects[i];
            }
        }

        if (willGrow)
        {
            GameObject obj = (GameObject)Instantiate(pooledObject);
            pooledObjects.Add(obj);
            return obj;
        }

        return null;
    }
}
```

Skripta 6.4: Izvorni kod uzorka Formiranje grupa objekata

Ono što se vidi iz prethodnog dela koda je da se održava lista instanciranih objekata koja se popunjava prilikom inicijalizacije celog skripta. Metoda getPooledObject vraća prvi slobodan element iz hijerarhije koji je neaktivan i samim tim spreman za novu upotrebu.

U posebnom slučaju, kada u grupi nema objekata koji bi mogli da budu instancirani (a objekat je zahtevan), pravi se nova instanca potrebnog objekta i on se dodaje u kolekciju. Ovo je kontrolisano opcionim parametrom *willGrow* koji dozvoljava ili zabranjuje instanciranje novih objekata za vreme izvršavanja.

Ono što je potrebno uraditi sa druge strane je odgovarajuće pozicioniranje konkretne instance u prostoru. Konkretno, u slučaju napravljene igre projektil je potrebno vratiti na početnu poziciju odakle će biti ispaljen. To sve je moguće uraditi u skriptu koji kontroliše samo ispaljivanje dok se za dobijanje konkretne instance koristi samo klasa *ObjectPoolerScript*.

Ovako napisana klasa može se koristiti za instanciranje bilo koje vrste objekata. Potrebno je samo nakačiti skript na odgovarajući objekat i podesiti predefinisani objekat (*eng. prefab*) koji će se koristiti za instanciranje, kao i inicijalni broj objekata koji će se čuvati u memoriji.

### 6.3.2 Reagovanje objekata na događaje

Još jedan od problema na koje se može naići jeste informisanje objekata o akciji koja se dogodila u nekom trenutku igre. Ovaj problem može delovati trivijalno za proste igre koje imaju mali broj objekata, ali kada se taj broj poveća, održavanje i praćenje objekata i akcija koje treba da obave postaje neizvodljivo. Ceo proces se dodatno komplikuje ukoliko se objekti prave dinamički, stoga nam je potreban efikasan mehanizam za obaveštavanje i okidanje akcija na određenim objektima. U ovome nam može pomoći još jedan uzorak za projektovanje pod nazivom *posmatrač* (*eng. Observer*) koji je opisan u narednom odeljku.

#### 6.3.2.1 Uzorak za projektovanje "Posmatrač"

Uzorak za projektovanje "Posmatrač" je softverski šablon u kom jedan objekat, najčešće poznat kao **subjekat**, održava listu od njega zavisnih objekata koji se nazivaju posmatrači. Subjekat automatski obaveštava sve svoje posmatrače o promeni koja se desila, što se najčešće realizuje kroz pozivanje nekog njihovog metoda. Uobičajena je praksa da ne reaguju svi posmatrači na sve događaje koje objavljuje subjekat, već ih filtriraju u zavisnosti od sopstvenih namera i potreba.

Najveću upotrebu ovaj uzorak za projektovanje ima u implementaciji grafičkih korisničkih okruženja, gde je našao mesto u gotovo svim alatima koji se danas koriste.

U nastavku je dat skup klasa koje se koriste za relizaciju posmatrača u konretnom projektu.

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

namespace ObserverPattern
{
    public abstract class Observer
    {
        public abstract void OnNotify();
    }

    public class ConcreteObserver : Observer
    {
        GameObject observerGameObject;
        EnemyEvents observerEvent;

        public ConcreteObserver(GameObject observerGameObject,
   EnemyEvents observerEvent)
        {
            this.observerGameObject = observerGameObject;
            this.observerEvent = observerEvent;
        }

        public override void OnNotify()
        {
            UpdateSpeed(observerEvent.GetSpeed());
        }

        void UpdateSpeed(float speed)
        {
            EnemyScript enemyScript =
   observerGameObject.GetComponent("EnemyScript") as EnemyScript;
            enemyScript.speed = speed;
        }
    }

    public class Subject
    {
        //A list with observers that are waiting for something to happen
        List<Observer> observers = new List<Observer>();

        //Send notifications if something has happened
```

```
        public void Notify()
        {
            for (int i = 0; i < observers.Count; i++)
            {
                //Notify all observers
                observers[i].OnNotify();
            }
        }

        //Add observer to the list
        public void AddObserver(Observer observer)
        {
            observers.Add(observer);
        }

        //Remove observer from the list
        public void RemoveObserver(Observer observer)
        {
        }
    }

    //Events
    public abstract class EnemyEvents
    {
        public abstract float GetSpeed();
    }

    public class SlowSpeed : EnemyEvents
    {
        public override float GetSpeed()
        {
            return 2f;
        }
    }

    public class FastSpeed : EnemyEvents
    {
        public override float GetSpeed()
        {
            return 3f;
        }
    }
}
```

Skripta 6.5: Izvorni kod uzorka Posmatrač

U konkretnom primeru pravljeno je obaveštavanje protivnika u zavisnosti od unapređenja oružja kupole. Kada protivnik (posmatrač u ovom slučaju) primeti da je oružje kupole postalo stepena 3, on automatski putem ovog mehanizma menja brzinu na jednu od dve predefinisane. Ovako je dobijena mogućnost da se različitim tipovima protivnika na različite načine menja brzina u zavisnosti od njihovih karakteristika i sklonosti.

Ovaj uzorak za projektovanje jako je čest u industriji video igara, pre svega zbog svoje primenljivosti i količine problema koju može da reši.

Sa druge strane, on ima i svoje nedostatke. Jedan od najvećih problema može da predstavlja curenje memorije koje se često može javiti, prvenstveno zbog potrebe da se eksplicitno registruju i uklanjaju posmatrači. U celom procesu je lako zaboraviti neaktivne posmatrače koji onda ostaju da vise i izazivaju curenje memorije. Ipak, uz pažljivo korišćenje, Posmatrač može doneti više koristi nego štete pa je njegovo korišćenje svakako preporučljivo.

# Glava 7

# Zaključak

Ovaj rad je koncipiran tako da na lep način prikaže upotrebu struktura podataka kroz moderne alate u razvoju video igara. Implementacione celine su raščlanjene tako da se odvojeno mogu koristiti u bilo kom projektu bez preteranih podešavanja ili prepravki. Osim što će ovako napravljeni moduli doprineti globalnoj zajednici kroz njihovu javnu dostupnost, takođe će svako ko tek ulazi u svet razvoja video igara imati dobru polaznu osnovu uz ovaj rad i njegove implementacione celine. S obzirom da Unity kao alat doživljava ogromnu ekspanziju, pre svega zbog svoje blage krive učenja, ovakvih dokumenata i kodova nikad neće biti dosta.

Kroz rad je takođe prikazana modularna priroda razvojnog okruženja Unity, gde nije bilo nikakvih problema prilikom sklapanja postojećih modula u smislenu celinu. Moduli su se ponašali baš onako kako je predviđeno njihovom izradom što samo potvrđuje robusnost i pouzdanost ovog alata u razvoju video igara.

Pored toga, predložena metodologija razvoja može se pokazati kao adekvatno rešenje već uigranim timovima i iskusnim projektantima.

Iako je u radu predstavljen veliki skup struktura podataka, na ovom polju mogućnosti su praktično neograničene. Jedna od prvih stvari na koju bi trebalo obratiti pažnju su svakako grafovi i njihova upotreba u mašinama stanja koje se koriste za definisanje odlučivanja o odigravanju određenih akcija među objekatima u igri. Pored toga, oblast veštačke inteligencije u igrama takođe zahteva posebne strukture podataka, za koje bi bilo interesantno imati gotove module koji se mogu uvrstiti kao dodaci i na taj način odmah koristiti.

Naravno, pored svega navedenog, igra koja je napravljena poseduje određene manjkavosti koje će u budućem periodu svakako biti unapređene, a sve kroz autorovo konstantno usavršavanje na polju razvoja video igara.

# Literatura


[1] Allen Sherrod. *Data structures and algorithms for game developers*. Charles River Media, 25 Thomson Place Boston, Massachusetts 02210, 2007.

[2] Jonathan G. Campbell. *Algorithms and Data Structures for Games Programming*. Stacy L. Hiquet, 2009.

[3] Unity manual - 2d or 3d, 2016. URL https://docs.unity3d.com/Manual/2Dor3D.html.

[4] Mark Overmars. *A Brief History of Computer Games*. Department of Information and Computing Sciences, Faculty of Science, Utrecht University, 2012. URL http://www.cs.uu.nl/docs/vakken/b2go/literature/history_of_games.pdf.

[5] Mark J.P. Wolf. *The video game explosion*. Greenwood Press, Westport, 2008.

[6] Miodrag Živković. *Algoritmi*. Matematički fakultet, Univerzitet u Beogradu, 2013.

[7] Aske Plaat, Jonathan Schaeffer, Wim Pijls , Arie de Bruin. Best-first fixed-depth minimax algorithms. 1995.

[8] Mladen Nikolić, Predrag Janičić. *Veštačka inteligencija*. Matematički fakultet, Beograd, 2016.

[9] Dave Mount. Geometric data structures for games: Index structures. 2013. URL https://www.cs.umd.edu/class/spring2013/cmsc425/Lects/lect10.pdf.

[10] Hanan Samet. An overview of quadtrees, octrees, and related hierarchical data structures, poglavlje u knjizi theoretical foundations of computer graphics and cad, volume 40 of the series nato asi series pp 51-68.

[11] Ron Penton. *Data structures for game programmers*. Stacy L. Hiquet, 2645 Erie Avenue, Suite 41 Cincinnati, Ohio 45208, 2003.

[12] Jianxiong Pang, Lynne Blair. Refining feature driven development - a methodology for early aspects. 2004.

[13] Clemens Szyperski, Jan Bosch, Wolfgang Weck. Component-oriented programming. 1999.

[14] Olof Wallentin. Component-based entity systems : Modular object construction and high performance gameplay. 2014.

[15] Skin mesh combine utility, 2016. URL http://wiki.unity3d.com/index.php?title=SkinMeshCombineUtility.

[16] Go kit tween library, 2016. URL https://www.assetstore.unity3d.com/en/#!/content/3663.

[17] Open game art, 2016. URL http://opengameart.org/.