ALGORITMI ZA PROCEDURALNO GENERIRANJE SADRŽAJA U RAČUNALNIM IGRAMA

Panić, Marko

VISOKO UČILIŠTE ALGEBRA

DIPLOMSKI RAD

ALGORITMI ZA PROCEDURALNO GENERIRANJE SADRŽAJA U RAČUNALNIM IGRAMA

Marko Panić

Zagreb, rujan 2018.

Sažetak

Glavna tema diplomskog rada je proceduralno generiranje sadržaja u računalnim igrama pomoću programskih algoritama, na praktičnom primjeru dvodimenzionalne igre izrađene pomoću Unity alata. U prvom dijelu rada su opisani odabrani softverski alati korišteni prilikom praktične izrade računalne igre, te njihove osnove komponente. Nakon toga su opisani glavni koncepti i ideja same igre, te osnovni elementi u sklopu Unity-a od kojih je ona izrađena. Svi elementi igre su praćeni s primjerima iz programskog koda, koji pokazuju interakciju između njih i pozadinskih skripti. Središnji dio rada sadrži opis pet različitih algoritama čija je osnovna zadaća kreiranje dijela sadržaja unutar igre pomoću koda u C# programskom jeziku, popraćen s praktičnim primjerima korištenja tih algoritama u igri. Na kraju rada su predstavljeni rezultati analize i usporedbe primijenjenih algoritama u četiri različite kategorije. Kompleksnost algoritama mjeri maksimalni broj koraka potrebnih za izvršavanje algoritama. Performanse algoritama su analizirane u sklopu opterećenja procesora i memorije za vrijeme izvršavanja algoritama tijekom pokretanja igre. Kategorije fleksibilnosti i igrivosti donose nešto subjektivniju analizu algoritama iz perspektive njihovog proširivanja, iskorištavanje u drugim igrama, te kvalitete sadržaja koji je generiran pomoću njih iz perspektive igrača.

Abstract

Main topic of this thesis is procedural generation of content in video games using programming algorithms, with practical application in a two-dimensional game developed using Unity software. First section describes software tools used for developing the game and their basic components. Main idea and concepts behind the game are described in the next section, including the main components in Unity used during game development. All game elements contain code examples, with the purpose of demonstrating interaction between them and programming scripts running in the background. Central section contains the description of five different algorithms used for procedural content generation using C# programming laguange, together with actual code examples showing the practical application of these algorithms during game development. The ending section of this thesis contains analysis and comparison between algorithms used in the game in four different categories. Algorithm complexity shows maximum number of iterations needed for execution of the chosen algorithm. Algorithm performance is shown using processor and memory usage during their running time. Flexibility and playability contain a more subjective analysis of algorithms in terms of their expandability, their ability to be used in other games, and the quality of game content that was generated using the algoritms, from the perspective of the game player.

Sadržaj

1. Uvod

Razvoj računalnih igara obuhvaća sve aktivnosti potrebne za izradu igara i njihovog sadržaja, te njihovu eventualnu isporuku do krajnjih korisnika, odnosno tržišta. Taj proces najčešće ne uključuje samo izradu računalne igre i njezinih elemenata pomoću programerskih i multimedijskih alata, već i sve popratne aktivnosti, od izrade prototipova i pisanja dokumentacije za koncept igre u početnim stadijima, pa sve do izrade promotivnog materijala i sadržaja za već završenu igru.

Proceduralno generiranje podataka u sklopu izrade računalnih igara se odnosi na izradu i primjenu programskih algoritama koji su sposobni samostalno generirati podatke i elemente u samoj igri. Ovo programerima donosi brojne mogućnosti, prvenstveno u pogledu uštede vremena potrebnog za preciznu ručnu izradu i dizajn različitog sadržaja. Proceduralno generiranje otvara i mnogo potencijalnih pogodnosti u igrama, kao što su znatno smanjenje memorijskog prostora potrebnog za spremanje tradicionalno ručno izrađenih elemenata, kao i mogućnost brzog generiranja velike količine sadržaja. Korisnicima i igračima su igre izrađene na ovaj način zanimljivije i manje predvidljive od klasično izrađenih igara, jer su prilikom svakog pokretanja barem malo drugačije (zahvaljujući nasumičnosti).

U sklopu ovog diplomskog rada će biti demonstrirane i analizirane neke od mogućnosti proceduralnog generiranja sadržaja prilikom izrade jednostavne računalne igre korištenjem programskog alata Unity, te popratnog programskog koda napisanog u C# programskom jeziku. Svrha i cilj samog rada bit će izrada i primjena nekoliko algoritama za proceduralno generiranje sadržaja u igri, te će zbog toga u sklopu rada veći naglasak biti stavljen na taj segment razvoja igre, a manje na popratne elemente i sadržaje. Prikazani algoritmi uključivat će i relativno jednostavne načine manipuliranja sadržajem, ali i malo kompleksnije procedure bazirane na postojećim metodama i algoritmima iz područja teorijske matematike. Jedan od glavnih elemenata rada bit će usporedba i analiza primijenjenih algoritama prema nekoliko objektivnih i subjektivnih kriterija, kao što su kompleksnost, performanse, fleksibilnost i igrivost.

U prvom poglavlju rada će biti opisano korištenje proceduralnog generiranja sadržaja kroz povijest razvoja računalnih igara, te razlozi njezina korištenja prilikom razvoja igara. U idućem poglavlju

će ukratko biti predstavljeni softverski alati korišteni u praktičnom dijelu rada, uz kratko objašnjenje njihovog odabira. U trećem poglavlju je opisan koncept slučajnih brojeva, te njihovo korištenje unutar Unity alata i C# programskog jezika. Četvrto poglavlje sadrži opis računalne igre izrađene za potrebe ovog rada, i način izrade individualnih komponenti unutar Unity-a. Peto poglavlje je usmjereno na algoritme za proceduralno generiranje sadržaja unutar igre, te sadrži detaljniji opis svakog od pet korištenih algoritama, popraćeno s dijelovima C# programskog koda. Zadnje poglavlje prikazuje rezultate provedene analize algoritama, fokusirane na kompleksnost, performanse, fleksibilnost i igrivost sadržaja generiranog pomoću primijenjenih algoritama.

2. Proceduralno generiranje sadržaja kroz povijest video igara

Algoritme za proceduralno generiranje sadržaja iz perspektive razvoja računalnih igara počelo se koristiti još 80-ih godina 20. stoljeća. Zbog velikih hardverskih ograničenja (pogotovo u pogledu pohrane podataka), programeri su bili primorani tražiti načine kako isporučiti što više sadržaja uz što veću uštedu memorije. Također se najčešće pokazalo da najviše memorijskog prostora zauzimaju unaprijed kreirane razine (nivoi), karte ili slični objekti koji služe kao reprezentacije okoline i prostora po kojem se igrač kreće i vrši interakciju s objektima.

Korištenjem proceduralnog generiranja većine sadržaja, odnosno kompletnih razina igre, zajedno sa svim objektima potrebnim za predviđeno funkcioniranje igre (npr. razmještanje protivnika, prepreka i ostalih predmeta po tako „slučajno“ kreiranim razinama) potpuno se zaobišla potreba za kreiranjem i spremanjem unaprijed definiranih razina igre. Prilikom samog pokretanja igre, algoritmi u pozadini bi generirali sav potreban sadržaj koji bi privremeno bio pohranjen u radnoj memoriji računala, a prema potrebi (npr. prilikom učitavanja nove razine) bi se ta memorija praznila i punila novo generiranim sadržajem.

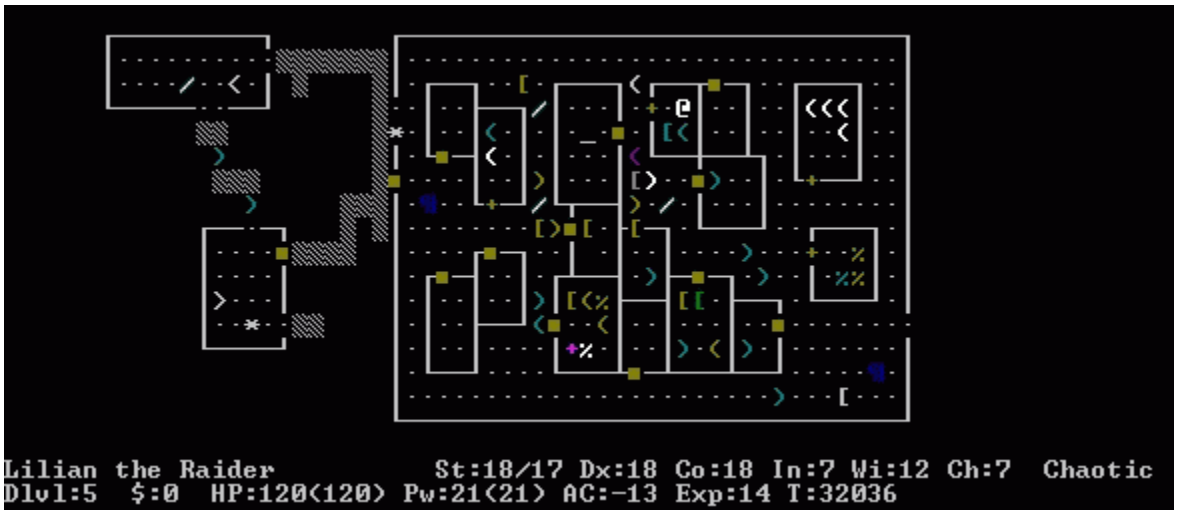
Najveći broj igara s primijenjenim metodama proceduralno generiranog sadržaja su pripadale takozvanom RPG (engl. *roleplaying game*) žanru, odnosno preciznije podžanru RPG igara kojeg su igrači neslužbeno nazvali engl. *Roguelike*. Taj naziv potječe iz igre nazvane Rogue iz 1980. godine, koja je bila prvi pravi predstavnik ovakve, tada nove vrste računalnih igara, te je svojim inovativnim pristupom postavila osnovne temelje žanra koji se i danas mogu vrlo lako uočiti prilikom razvoja sličnih igara.

Cijela igra se nije odvijala u stvarnom vremenu, već u strogo definiranim „potezima“ (po uzoru na šah i slične igre), što je značilo da igrač može izvršiti neku radnju tijekom koje protivnici miruju, a tek nakon toga bi se obavila kretanja ili ostale radnje svih protivnika, i tako stalno u krug. Umjesto klasične grafike, koristila je ASCII (engl. *American Standard Code for Information Interchange*) simbole koji su predstavljali različite objekte i predmete u igri, smještene na dvodimenzionalnoj ploči, odnosno matrici. Svaki simbol je imao svoje jedinstveno značenje unutar

igre, primjerice igrač je bio predstavljen znakom '@', neprohodne prepreke i zidovi znakom '|', a slobodna polja po kojima se igrač može kretati znakom '.'

Kombinacijom ovih (ali i mnogo drugih) ASCII znakova algoritmi su pozadini generirali čitave razine kroz koje je igrač morao prolaziti. Budući da je sve bilo realizirano pomoću proceduralne logike korištenjem slučajnih odnosno pseudo-slučajnih brojeva, svako pokretanje igre i svaka razina je bila drugačija od prethodne, što je rezultiralo time da igrači (bez obzira koliko puta su već igrali ovu igru) uvijek moraju obraćati pozornost i prilagođavati se okolini, bez mogućnosti da „napamet“ nauče sve razine i prepreke na koje mogu naići tijekom igre.

Rogue je bio distribuiran kao potpuno besplatna i nekomercijalna igra, zbog čega je brzo stekla veliku popularnost kod ljubitelja starijih računalnih igara, a pogotovo među studentima računarstva i vezanih informacijsko-komunikacijskih tehnologija. Zahvaljujući tome ubrzo su se počele pojavljivati nove igre rađene po uzoru na Rogue, a većinom su bile napravljene kao amaterski i nekomercijalni hobi projekti studenata. Najpoznatije tzv. Roguelike igre iz 80-ih su bile Moria i Hack, koji je u kasnim 80-im godinama „evoluirao“ u NetHack, čije se osnovno sučelje i izgled može vidjeti na idućoj slici (Slika 2.1).



Slika 2.1 Izgled igre NetHack

NetHack, prvi put službeno izdan 1987. godine, i danas predstavlja najpopularniju (i prema mnogima najbolju) igru ovog žanra. O njezinoj popularnosti i kultnom statusu dovoljno svjedoči činjenica da je u razvoju već preko 30 godina (zadnja stabilna release verzija je iz 27.04.2018.

godine), te je većina osoba koja je sudjelovala na prvoj verziji igre još uvijek donekle uključena u sam projekt. Budući da je igra službeno licencirana pod vlastitom licencom nazvanom NetHack General Public Licence u sklopu neprofitne Open Source Initiative organizacije, njezin čitav izvorni kod je dostupan svima na korištenje. Radi toga postoji jako veliki broj neslužbenih verzija i ogranaka igre, uključujući i verzije za sve postojeće mobilne platforme, i verzije igre koje se mogu igrati direktno iz mrežnog preglednika. Zahvaljujući lako dostupnom izvornom kodu, NetHack poslužio je mnogim igračima i kao motivacija za učenje programiranja i razvoja računalnih igara.

Osim uštede prostora, druga najveća prednost korištenja proceduralnog generiranja sadržaja je i mogućnost stvaranja uistinu velikog broja objekata unutar same igre. Najpoznatiji, a ujedno i najekstremniji primjer ovakvog pristupa razvoju igara je igra Elite iz 1984. godine. Za razliku od prethodno spomenutih Roguelike igara na kojima su se igrači kretali po relativno jednostavnim dvodimenzionalnim površinama, Elite je bio potpuno trodimenzionalna igra, koristeći takozvane žičane odnosno wire-frame modele za reprezentaciju okoline. Cilj igre bilo je istraživanje svemira, a zahvaljujući vrlo kreativnom korištenju proceduralnog generiranja sadržaja, igra je sadržavala čak 8 različitih galaksija, svaka s 256 planeta. Svi planeti i galaksije su morali biti proceduralno generirani, budući da jednostavno nije bilo moguće unaprijed pohraniti ni približno dovoljan broj unaprijed dizajniranih objekata.

Prilikom inicijalnog pokretanja igre, algoritam bi izvršio proces kreiranja čitavog svemira prikazanog u igri. Generirao bi se jedinstveni broj kojeg još nazivamo *seed*, a on bi se proslijedio prema algoritmu za proceduralno generiranje svemira, a taj algoritam bi pak na temelju tog broja generirao čitav niz drugih brojeva, od kojih bi svaki predstavljao jedinstveni ključ odnosno identifikator planeta u igri. Na temelju tog pseudo-nasumičnog broja planeta, igra bi znala točno u kojoj galaksiji i na kojoj poziciji mora smjestiti i prikazati planet određenih dimenzija i izgleda.

3. Alati

Algoritmi za proceduralno generiranje sadržaja u video igrama u pravilu ne zahtijevaju gotovu igru u kojoj se koriste, već mogu biti razrađeni i kao pretežno teorijski koncepti, koji se zatim prilagođavaju igrama ili platformama na kojima se primjenjuju. Unatoč tome, u sklopu rada je napravljena dvodimenzionalna računalna igra po uzoru na starije igre iz 80-ih godina (opisano u poglavlju 2), s ciljem demonstriranja algoritama u praktičnoj upotrebi. Budući da je naglasak na samim algoritmima, a ne na izradi računalne igre, odlučeno je ostale elemente igre zadržati na što jednostavnije mogućoj razini, i tako osigurati da konačni rezultati analize i prikaza algoritama budu što jasniji i pregledniji, bez previše suvišnih informacija i objekata.

Za izradu računalne igre i svih njezinih elemenata odabran je softverski alat Unity (verzija 2017.2), te C# programski jezik u kojem će biti napisan sav programski kod potreban za izvršavanje i izvođenje igre. Pisanje programskog koda u C# jeziku je napravljeno pomoću Visual Studio Community 2017 IDE (engl. *integrated development environment*) alata. Prilikom razmatranja odabira alata za izradu igre, dva glavna kriterija su bila cijena (skup alata mora biti potpuno besplatno dostupan za korištenje u nekomercijalne edukacijske svrhe), te pristupačnost odnosno popularnost, što bi značilo da sam alat već ima veliku i dobro izgrađenu zajednicu korisnika oko sebe, te detaljnu i široku dostupnu dokumentaciju i ostalu popratnu literaturu koja opisuje njegove funkcionalnosti.

Zbog svega navedenog, Unity se pokazao kao idealno rješenje za realiziranje teme ovog diplomskog rada, budući da je trenutno najpopularniji i najrašireniji alat (odnosno skup alata) na tržištu za izradu računalnih igara. Unity uključuje nativnu podršku za C# programski jezik, zbog čega je on odabran kao programska podloga prilikom izrade igre. Visual Studio Community je odabran kao alat za pisanje C# programskog koda zato što je besplatan u nekomercijalne svrhe, te nudi daleko najbolje mogućnosti prilikom samog programiranja, ali i automatske integracije s Unity alatom. Iako je Unity prvenstveno razvijen i koristi se prilikom izrade igara na konvencionalni način, s ručno izrađenim komponentama i elementima, jedan od ciljeva rada je demonstriranje mogućnosti korištenja programske logike i koda za stvaranje određenih dijelova igre.

3.1. Unity

Unity je alat, odnosno skup alata, namijenjenih prvenstveno za izradu dvodimenzionalnih i trodimenzionalnih računalnih igara i svih njihovih ključnih komponenti. Razvijen je od strane Unity Technologies kompanije, a prvi puta je izdan na tržište 2005. godine. Službeno je kategoriziran kao takozvani pokretač ili jezgra igre (engl. *game engine*). Game engine zapravo označava cijelo softversko okruženje namijenjeno izradi i pokretanju računalnih igara. Glavni razlog razvoja i korištenja game engine-a je u tome što oni nude čitavi skup alata integriranih u sklopu svoje razvojne okoline, čime se znatno ubrzava i olakšava cjelokupni proces izrade računalnih igara. Ovakvi se alati često nazivaju i kategoriziraju kao engl. „middleware“, budući da iz profesionalnog gledišta predstavljaju svojevrsnu poveznicu, odnosno srednji sloj između osobe ili kompanije koja se bavi profesionalnom izradom računalnih igara, te samih korisnika i tržišta.

Osnovne funkcionalnosti velike većine game engine-a, pa tako i Unity-a, uključuju 2D i 3D *renderer* za grafički prikaz elemenata igre, podršku za reprodukciju zvuka unutar igre, podršku za rudimentarnu fiziku koja određuje interakcije između objekata (npr. kako se dva objekta ponašaju ako dođe do međusobnog sudara), mogućnost izrade animacija za objekte unutar igre, te mogućnost pisanja programskog koda odnosno skripti u nekom programskom jeziku čime se proceduralno definira kompletno ponašanje i izvođenje igre. Trenutno se Unity softversko okruženje za razvoj može instalirati i koristiti na Windows, Linux i MacOS operacijskim sustavima, s tim da je verzija za Linux još uvijek u testnoj fazi, te ne sadrži sve mogućnosti koje imaju ostale dvije verzije. Od grafičke podrške za interakciju s grafičkim karticama i procesorima, Unity nudi API sučelja za Direct3D (grafičko programsko sučelje za Microsoft Windows), OpenGL (prisutan na gotovo svim operacijskim sustavima za osobna računala), WebGL (programsko sučelje koje omogućava prikaz 2D i 3D grafike unutar samog mrežnog preglednika), te Vulkan (novo multi-platformsko grafičko programsko sučelje namijenjeno kao svojevrsna nadogradnja i zamjena za OpenGL).

Iako je originalno namjena Unity-a bila izrada igara isključivo za OS-X (operacijski sustav korišten na Apple Mac seriji osobnih računala), ubrzo se podrška proširila i na ostale popularne platforme. Danas Unity podržava izvođenje računalnih igara na čak 27 različitih platformi,

uključujući i sve postojeće mobilne operativne sustave na pametnim telefonima. Jedan od glavnih fokusa je i na podršci za istovremeni razvoj igara za više različitih platformi, te je vrlo jednostavno (naravno uz razmatranje hardverskih ograničenja svakog sustava) jednom završenu igru pokrenuti na više operacijskih sustava i platformi, bez ikakve potrebe za razvojem ili primjenom dodatnih programskih rješenja. Osim podrške za izradu računalnih igara, Unity se sve više proširuje i povezuje s raznim alatima za 3D modeliranje, te CAD alatima korištenim za izradu detaljnih 2D i 3D tehničkih crteža, modela i dokumentacije. Također je moguća izrada modela i sadržaja za virtualnu stvarnost u sklopu sustava za virtualnu stvarnost poput Oculus Rift i HTC Vive sustava, augmentiranu stvarnost (engl. *augmented reality*), te izradu aplikacija za pametne televizore i slične uređaje.

U sklopu sustava dolazi i niz pomoćnih alata i servisa za korisnike, poput Unity Analytics servisa za detaljnu statističku analizu podataka vezanih uz isporučenu računalnu igru (npr. kako se najčešće igrači ponašaju, koje opcije biraju ili izbjegavaju, i slično), Unity Ads servisa za vrlo laganu integraciju sustava plaćenog oglašavanja unutar igre (vrlo često korišteno prilikom razvoja igara za mobilne platforme), ili pak Unity Cloud Build alata pomoću kojeg je moguće brzo isporučivanje gotove igre na različite cloud computing platforme (računalstvo u oblaku). Za skriptiranje Unity nudi API (engl. *application programming interface*) za C# programski jezik, te sadrži vlastiti alat za prevođenje (engl. *compiler*) C# programskog koda. Do sredine 2017. godine službeno je bila podržana i verzija JavaScript programskog jezika, posebno adaptirana za Unity pod nazivom UnityScript, ali je ta podrška ukinuta počevši od 2017.1 verzije. Za pisanje programskog koda se može koristiti bilo koji postojeći program za uređivanje teksta, kao i bilo koji dostupni IDE alat. Visual Studio nudi i niz brojnih prednosti, kao što je automatsko prepoznavanje svih ključnih riječi prilikom programiranja, te mogućnost pokretanja debug alata (alat za analizu i otkrivanje grešaka u kodu) paralelno s pokretanjem i izvođenjem same igre.

4. Slučajni brojevi i njihova primjena

Slučajni (odnosno nasumično odabrani) brojevi čine srž i osnovu svakog procesa, procedure ili algoritma kojem je zadaća proceduralno generiranje podataka. Pomoću njih se određuje smjer „kretanja“ algoritma, te postupci koje on poduzima prilikom procesiranja informacija. Slučajne brojeve generiraju različiti algoritmi i procesi, i mogu biti realizirani kao generatori pravih slučajnih brojeva (engl. *true random number generator*), ili pak generatori takozvanih pseudo-slučajnih brojeva (engl. *pseudo-random number generator*). Iako obje vrste algoritama na prvi pogled rade istu stvar i daju vrlo slične rezultate, zapravo su bazirani na vrlo različitom pristupu prema cijelom procesu.

Generatori pravih slučajnih brojeva su bazirani na različitim prirodnim pojavama i fizikalnim procesima. Događaji poput bacanja kocke, novčića ili pak mjerenja različitih zvukova i šumova izazvanih prirodnim procesima u zemljinoj atmosferi daju rezultate koji se mogu okarakterizirati kao pravi slučajni brojevi i mjerenja. Tako primjerice prilikom bacanja klasične kocke sa šest strana, vjerojatnost da se pokaže neki broj je uvijek slučajna (te ista za svaki ishod), te čak i ako ponavljamo bacanja mnogo puta i zapisujemo rezultate, ne postoji mogućnost da će se pojaviti neki uzorak prema kojem bi mogli predviđati buduće rezultate, ili uz čiju pomoć bi mogli ponoviti mjerenja i dobiti svaki put identične rezultate.

Računalno generiranje pravih slučajnih brojeva mora na potpuno nepredvidiv način odabirati brojeve u nekom rasponu, te se ne smiju generirati nizovi brojeva koji prate određena pravila ili uzorke. Zbog ovoga, nije u potpunosti moguće generirati prave slučajne brojeve koristeći algoritme i slične programske metode na računalima. Sam proces definiranja algoritama i neke popratne proceduralne logike znači da rezultat nije potpuno slučajan, te su se prilikom njegovog generiranja slijedila određena unaprijed definirana pravila. Jedini pravi način za dobivanje potpunih i pravih slučajnih brojeva na računalima je pomoću mjerenja i uzimanja uzoraka iz bilo kakvih prirodnih entropijskih izvora. Takvi brojevi se obično koriste samo u ekstremnim slučajevima, najčešće u području kriptografije i informacijske sigurnosti. Zbog sveg navedenog, većina slučajnih brojeva generiranih pomoću računala spada u kategoriju pseudo-slučajnih brojeva. Oni se generiraju pomoću različitih algoritama koji na prvi pogled daju prave slučajne

vrijednosti, ali njihovi su rezultati zapravo određeni s pomoću specifičnog broja ili ključa, koji se najčešće naziva *seed*.

Svaki *seed* zapravo definira određeni jedinstveni identifikator, formulu ili pak uzorak prema kojem će se generirati čitav niz naizgled nasumično odabranih vrijednosti, te je uz njegovo poznavanje moguće svaki puta dobiti identičan niz naizgled slučajnih podataka. Iako je mogućnost reproduciranja rezultata nasumičnog odabira naizgled loša ako želimo što veću moguću slučajnost, postojanje *seed-a* može biti vrlo korisno. Iz perspektive izrade video igara, igračima se može namjerno omogućiti pristup *seed* broju, kojeg onda oni mogu iskoristiti za ponovno generiranje nekog sadržaja koji im se najviše svidio. Isto tako je moguće prilikom prelaska s jedne na drugu razinu (od kojih je svaka proceduralno i slučajno generirana), potpuno obrisati prethodnu razinu i pohraniti samo *seed* vrijednost korištenu za njihovo generiranje, a ona se zatim može iskoristiti ako se kasnije igrač poželi vratiti na prethodnu razinu kako bi se ona ponovno generirala (Watkins, 2016).

4.1. Slučajni brojevi u sklopu Unity-a

Većina programskih jezika i alata za razvoj softvera sadrži vlastite gotove metode za generiranje pseudo-slučajnih brojeva. Prilikom razvoja igre koristeći Unity i C# programski jezik, postoje dva osnovna paketa odnosno postojeća načina za dobivanje slučajnih rezultata. To su `System.Random` biblioteka funkcija koja dolazi uz C# programski jezik, te `UnityEngine.Random` biblioteka funkcija u sklopu samog Unity alata. Osnovna razlika je tome što je `UnityEngine.Random` definiran kao statička klasa sa već implementiranim generatorom slučajnih brojeva. To znači da ga nije potrebno kreirati nove objekte ovo klase ako želimo dobiti slučajni broj, već odmah možemo pozivati i koristiti njezine metode, kao što se može vidjeti na idućem primjeru:

```
int broj = Random.Range(1, 101);
```

`System.Random` klasa nije statička. Zbog toga je prije korištenja metoda za dohvaćanje slučajnih brojeva potrebno ili uvijek kreirati nove objekte ove klase ili pak na neki način ručno definirati statički pristup objektu te klase, kako bi se on mogao koristiti na više različitih mjesta u

programskom kodu. Primjer definiranja slučajnog broja pomoću `System.Random` klase se može vidjeti u nastavku:

```
Random rand = new Random(); int broj = rand.Next(1, 101);
```

5. Izrada igre

U sklopu rada izrađena je dvodimenzionalna računalna igra koristeći Unity softverski alat, a služi kao podloga za praktičnu primjenu algoritama za proceduralno generiranje sadržaja unutar nje. U nastavku poglavlja će biti opisan osnovni koncept igre, glavni elementi samog Unity-a korišteni prilikom izrade osnovnih komponenti igre, te način na koji su same komponente izrađene i uklopljene u igru.

5.1. Osnovni koncept i ideja igre

Računalna igra (radni naslov *Random Rogue*) je dvodimenzionalna igra u kojoj igrač preuzima ulogu istraživača davno napuštenih dvoraca i tamnica u njima, s ciljem pronalaska i sakupljanja što više zlatnika. Prilikom istraživanja igrač dolazi u sukob s brojnim neprijateljima koji čuvaju tamnice, te pokušavaju spriječiti igrača da dođe do blaga i sigurno pobjegne iz tamnice. Sakupljene zlatnike igrač može iskoristiti za kupovinu bolje opreme, čime će mu se znatno olakšati borba protiv neprijatelja, a samim time i istraživanje svih tamnica u dvorcu. Nakon svakog poraženog neprijatelja igrač automatski dobiva bodove koji predstavljaju njegovo "iskustvo" u istraživanju i borbi, a prelaskom određenih bodovnih pragova u iskustvu igrač prelazi na višu "razinu", te može primiti "više štete" od neprijatelja.

Osnovne mogućnosti igrača su slobodno kretanje u dvije dimenzije, te korištenje oružja za borbu s neprijateljima ili uništavanje prepreka. Kao što se može vidjeti i na slici (Slika 5.1), na početku igre igrač je smješten u prizemlju dvorca, a od tamo pristupa i ulazi u ostale razine dvorca. Svaka razina (odnosno tamnica u dvorcu) sadrži zlatnike koje igrač može sakupiti, neprijatelje, te jedan ili nekoliko izlaza. Ako igrač uspješno stigne do izlaza, vraća se ponovno na osnovnu razinu dvorca, odakle opet može pristupiti nekoj drugoj razini unutar dvorca.

Slika 5.1 Slika početne razine igre

Iako je velika većina sličnih igri napravljena po principu igara na poteze (engl. *Turn-based game*), akcije u ovoj igri se odvijaju u stvarnom vremenu, što znači da igrač i njegovi protivnici mogu istovremeno izvoditi određene radnje, bez potrebe za čekanjem. Ovaj pristup je odabran zato da bi se ubrzalo testiranje i analiziranje algoritama, jer nije potrebno čekati na izvršavanje asinkronih poteza unutar igre.

5.2. Elementi i dizajn igre

Prilikom prvog pokretanja i kreiranja 2D projekta za računalnu igru, Unity alat sadrži samo osnovne prazne elemente i objekte, te je potrebno ručno izraditi sve što je potrebno za pokretanje i izvršavanje igre. U tu svrhu Unity nudi mnogo različitih pogleda, prozora i perspektiva putem kojih se može dobiti uvid u svaki ključni element igre, te je stoga bitno razumjeti njihovu ulogu i međusobnu povezanost.

5.2.1. Osnovni elementi Unity alata

Osnovni alati i prozori unutar Unity alata potrebnih za izradu igre čine sljedeći elementi:

- Ekran za prikaz scene
- Hijerarhijski pregled objekata

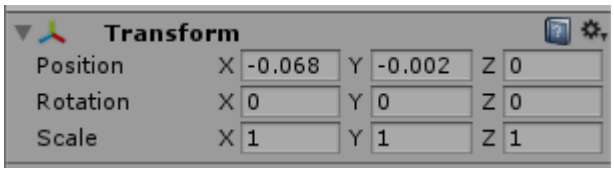
- Inspektor
- Ekran za prikaz same igre
- Pregled strukture projekta
- Radna traka
- Konzolni prozor

Ekran za prikaz scene (engl. *Scene view window*) je osnovni prozor za vizualnu interakciju s igrom. Scena sadrži sve elemente igre, uključujući i pomoćne elemente poput izbornika, te je igračima moguće prikazati ili omogućiti interakciju samo s onim objektima koji su smješteni negdje na sceni. Igra može sadržavati praktički neograničen broj scena, ali u jednom trenutku samo jedna scena može biti aktivna, odnosno prikazana igraču. Jedan od glavnih načina za uporabu scena je upravljanje različitim razinama igre, i to strukturiranjem elemenata igre tako da svaka scena zapravo predstavlja jednu razinu igre. Ovako je moguće vrlo jednostavno kontrolirati razinu koja je trenutno učitana i aktivna, ili odrediti koju razinu učitati kasnije.

Glavni element koji omogućava vizualni prikaz bilo kojeg elementa na sceni je kamera. Ona određuje perspektivu iz koje "snimamo" elemente igre, te što je vidljivo igračima igre (sve unutar dimenzija kamere) ili sakriveno od igrača (sve što se nalazi izvan vidokruga kamere). Kamera je potpuno dinamična, što znači da se tijekom izvođenja igre može slobodno kretati u tri dimenzije, mijenjati dimenzije ili perspektive, te tako određivati što igrači vide. Kamere podržavaju dva osnovna načina prikaza elemenata na sceni – perspektiva ili ortografski prikaz. Perspektiva (engl. *Perspective camera*) znači da kamera vodi računa o udaljenosti objekata od kamere, te će se udaljeniji objekti činiti manjim, a bliži većima, što je prvenstveno namijenjeno izradi 3D igara. Ortografska kamera (engl. *Orthographic camera*) sve objekte na sceni prikazuje u točno onoj veličini i dimenzijama koje su određene prilikom njihovog pozicioniranja na scenu, što je čini idealnim za korištenje u 2D igrama.

Hijerarhijski pregled objekata (engl. *Hierarchy window*) sadrži sve objekte i elemente koji se nalaze u trenutno aktivnoj sceni. Osnovni gradivni element svake scene predstavljaju *GameObject* elementi, koji predstavljaju svojevrsni ekvivalent temeljne *Object* klase u većini objektno orijentiranih programskih jezika. *GameObject*, sam po sebi, nema nikakvih mogućnosti niti izgleda, već se pomoću dodavanja različitih komponenti određuje njegova uloga i ponašanje. Tako,

primjerice, dodavanjem kamera komponente na prazni *GameObject* on zapravo postaje kamera unutar scene, a dodavanjem *Sprite Renderer* komponente na prazni objekt određujemo kako će on izgledati i prikazivati se na sceni. Osnovno svojstvo svakog objekta kojeg se koristi na sceni su njegova transformacijska svojstva (engl. *transform property*), i to pozicija, rotacija i skala (veličina). To je ujedno i jedino obavezno svojstvo svakog `GameObject` elementa, te uvijek mora biti prisutno na njemu (Gibson, 2014).



Slika 5.2 Određivanje transfornmacijskih svojstava u Unity-u

Definiranje navedenih svojstava objekata unutar Unity-a se može vidjeti na prethodnoj slici (Slika 5.2). Pozicija elemenata određuje njegove X, Y i Z koordinate unutar scene, odnosno udaljenost od središnje točke unutar scene koja predstavlja ishodište koordinatnog sustava s tri osi. Rotacija određuje stupanj za koji je neki objekt zakrenut oko bilo koje od tri osi koje predstavljaju njegove tri dimenzije (dužina, širina i visina). Skala određuje relativnu veličinu elemenata u odnosu na druge elemente i osnovno određenu veličinu. Tako primjerice elementi sa skalom jedan predstavljaju osnovnu veličinu svakog objekta, dok skala dva predstavlja točno duplo veći objekt od osnovnog.

Svi elementi na sceni su organizirani prema roditelj-dijete hijerarhijskoj strukturi (engl. *Parent-child relationship*). Svaki element može biti postavljen kao dijete nekog drugog elementa, ali isto tako može imati i niz elemenata koji čine njegovu djecu. Elementi postavljeni kao djeca automatski prate poziciju svojeg roditelja, što znači da ako se kamera postavi kao dijete glavnog lika igrača, ona će uvijek pratiti igrača i biti fokusirana na njega. Isto tako ako želimo na vrlo brz način pozicionirati elemente koje igrač nosi sa sobom (npr. neko oružje ili sličan predmet), dovoljno ih je definirati kao djecu igrača, te će oni automatski uvijek biti vezani uz njega.

Inspektor (engl. *inspector window*) je prozor koji sadrži sve karakteristike i svojstva željenog ili označenog elementa. Preko ovog prozora moguće je definirati sva svojstva nekog elementa, od

njegovog imena i transformacijskih svojstava, pa sve do programskih skripti i koda kojim se upravlja njegovim ponašanjem.

Ekran za prikaz igre (engl. *Game view window*) prikazuje kako će točno izgledati igra prilikom njezinog prvog pokretanja, ili prilikom pokretanja određene scene. On zapravo prikazuje ono što kamera definirana u sklopu scene, vidi i prikazuje, te je vrlo koristan ako se želi vidjeti kako elementi scene i kamera utječu na sam izgled i prezentaciju igre.

Prozor za prikaz strukture projekta pokazuje sve spremljene i korištene elemente projekta, te njihovu lokaciju unutar strukture projekta. Preko ovog prozora moguće je lako kreirati nove mape, te smještati nove ili već postojeće elemente unutar njih. Radna traka (engl. *toolbar*) je najčešće smještena iznad prozora za prikaz scene i igre, te sadrži osnovne kontrole za interakciju sa scenom i igrom. Preko radne trake moguće je pokrenuti izvođenje igre, pauzirati već pokrenutu igru, regulirati prikaz objekata na sceni, te direktno manipulirati njihovim transformacijskim svojstvima.

Konzolni prozor (engl. *console window*) prikazuje sve poruke generirane od strane Unity-a tijekom izvođenja igre, uključujući sve greške i poruke upozorenja. Korištenje konzolnog prozora je ključno ako se otkriju greške ili neželjena ponašanja u igri, te ga je moguće povezati s *debug* alatom iz Visual Studio softvera za detaljnu analizu izvođenja programskog koda svih C# programskih skripti.

5.3. Glavni elementi igre

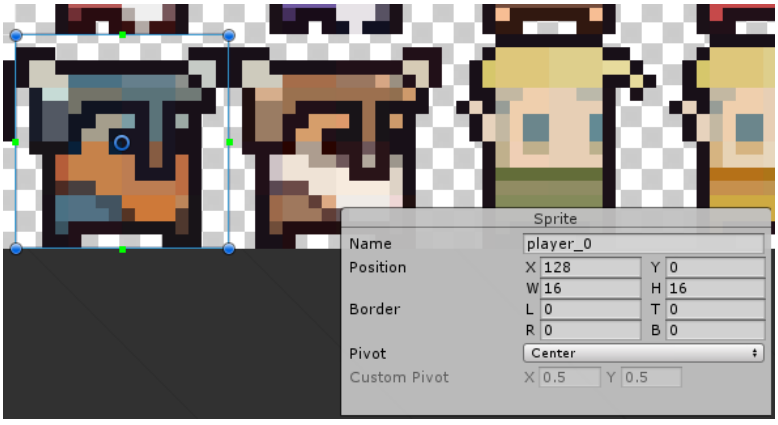
Prvi i osnovni korak prilikom izrade igre je kreiranje glavnih objekata koji će se koristiti u gotovo svim scenama i situacijama. Takve se objekte u sklopu Unity-a naziva engl. *prefab*. Svaki *prefab* je predložak za izradu komponente igre, odnosno objekt s definiranim karakteristikama poput veličine, grafičkog prikaza ili animacije, te svih popratnih komponenata poput programskih skripti. Jednom napravljen i spremljen predložak je zatim moguće dodati u scenu direktno, ili preko programskog koda, svaki puta kada nam zatreba njegova funkcionalnost. Glavna prednost korištenja *prefab* komponenti u usporedbi s klasičnim kopiranjem objekata iz jedne scene u drugu je u tome što je svaka promjena na njemu automatski primijenjena i na svim objektima koji koriste taj predložak (Lintrami, 2018).

Ako se primjerice odluči promijeniti izgled protivnika u igri, dovoljno je pronaći *prefab* koji predstavlja protivnika, te na njemu promijeniti grafički izgled, nakon čega će se ta promjena automatski vidjeti na svim protivnicima, odnosno objektima koji koriste ovaj predložak. U idućem poglavlju će biti opisani osnovni elementi, odnosno *prefab* objekti izrađeni i korišteni u samoj igri, uključujući njihove osnovne komponente i glavne dijelove i isječke programskog koda nužnog za njihovo ispravno funkcioniranje.

5.3.1. Grafika i slojevi

Prije same izrade elemenata potrebno je kreirati ili pripremiti već gotove grafičke komponente koje će određivati izgled odnosno grafiku igre. Sva grafika korištena u ovoj igri je napravljena korištenjem gotove besplatne grafičke palete preuzete sa sljedeće lokacije - https://0x72.itch.io/16x16-dungeon-tileset.

Ova grafička paleta je zapravo jedna velika slika u .png. formatu koja se sastoji od mnogo manjih elemenata često fiksne veličine, u ovom slučaju 16x16 piksela. Ovako definirani grafički element se najčešće zove i teksturni atlas (engl. *texture atlas).* Ako se neki od elemenata žele primijeniti na izgled komponenata unutar igre, potrebno je otvoriti atlas u *Sprite Editor* alatu unutar Unity-a, te "izrezati" željene manje elemente, nakon čega ih je moguće koristiti za definiranje grafičkog izgleda, kao što je vidljivo na sljedećoj slici (Slika 5.3).



Slika 5.3 Izrezivanje elemenata iz teksturnog atlasa

Još jedan vrlo važan aspekt prilikom definiranja grafičkog izgleda i ponašanja objekata su slojevi (engl. *layers*). Slojevi u Unity-u određuju kako se objekti iscrtavaju i prikazuju na sceni, te kako

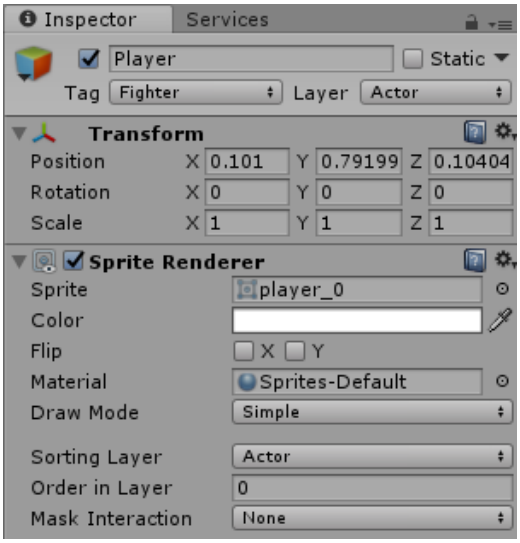
ih kamera prikazuje. Za izradu 2D igara su najvažniji slojevi za sortiranje objekata (engl. *sorting layers*), pomoću kojih se definira točan redoslijed prikaza objekata na sceni. Slojevi mogu imati proizvoljan naziv, te ne postoji strogo ograničen broj slojeva za korištenje u igri.

Slika 5.4 Korišteni slojevi za sortiranje u igri

Na prethodnoj se slici (Slika 5.4) mogu vidjeti slojevi za sortiranje i njihov redoslijed korišten unutar igre. Prvi sloj (*floor*) će biti primijenjen na sve objekte koji čine pod, odnosno osnovnu podlogu po kojoj se igrači kreću. Prepreke (poput zidova) mogu biti stavljene pod *wall* ili *interactable* sloj, čime se osigurava da će one biti prikazane "nakon" prvog sloja, odnosno gledano iz 2D perspektive zidovi će biti smješteni iznad poda. *Actor* sloj bit će korišten za prikaz glavnih aktera igre, odnosno glavnog lika i njegovih protivnika. Zadnji će sloj (*weapon*) biti rezerviran za prikaz oružja kojeg glavni lik koristi za borbu s protivnicima, a budući da se želi stvoriti 2D "iluzija" nošenja oružja, taj sloj će biti iscrtan nakon likova.

5.3.2. Glavni lik

Osnovni element igre je glavni lik kojeg igrači kontroliraju tijekom igranja. Definiranje grafičkog izgleda igrača može se vidjeti na prethodnoj slici (Slika 5.3). Svaki objekt koji treba imati grafički prikaz unutar igre mora imati definiranu *sprite renderer* komponentu, te unutar nje definiran konkretan *sprite* koji služi kao grafički prikaz objekta unutar scene. Na idućoj slici (Slika 5.5) možemo vidjeti kako je prethodno izrezani element pod nazivom `player_0` iskorišten za definiranje izgleda glavnog lika, te sloj za sortiranje koji je odabran za prikaz lika.



Slika 5.5 Dodavanje sprite renderer komponente na glavnog lika

Na glavnog lika je također dodana i *Box Collider 2D* komponenta. To je transparentni 2D okvir određenih dimenzija smješten oko objekata, a služi za upravljanje s kolizijama i sličnom interakcijom među objektima. Objekti bez definirane *collider* komponente ponašaju se kao da su nevidljivi, odnosno moguće je slobodno prolaziti direktno kroz njih, što nije željeno ponašanje za glavni lik, protivnike ili prepreke.

Za definiranje ponašanja i upravljanja glavnim likom, izrađena je `Player.cs` skripta s pripadajućim C# programskim kodom. Budući da je dosta ponašanja vezanih uz glavni lik potrebno i kasnije primijeniti nad protivnicima (i potencijalno sličnim objektima), u tu svrhu su napravljene i pomoćne klase nazvane `Mover` i `Fighter`. Klasa `Mover` sadrži osnovne metode za upravljanje kretanjem objekata, dok klasa `Fighter` sadrži programsku logiku za upravljanje borbom. Ove klase i njihove metode nisu korištene direktno, već preko nasljeđivanja, zbog čega će i sve njihove metode biti virtualne. Virtualne metode u sklopu C# programskog jezika označavaju svaku metodu čija se funkcionalnost prema potrebi može promijeniti odnosno premostiti (engl. *override*) unutar naslijeđene klase. Budući da igrači moraju upravljati glavnim likom preko tipkovnice, skripta pridružena glavnom liku nasljeđuje `Mover` klasu, zbog čega može koristiti programski kod za upravljanje kretanjem definiran unutar te klase. Kretanje se u Unity igrama određuje pomoću objekata `Vector3` klase, koja predstavlja vektor s 3 prostorne

koordinate (x, y i z os). Prije upravljanja kretanjem, potrebno je, prema potrebi, promijeniti orijentaciju glavnog lika, tako da on "gleda" u onom smjeru u kojem se kreće.

```
original = transform.localScale;
public float yAxisSpeed = 0.75f;
public float xAxisSpeed = 1.0f;
moveVector = new Vector3(input.x * xAxisSpeed, input.y *
yAxisSpeed, 0);
if (moveVector.x > 0) {
 transform.localScale = original;
}
else if (moveVector.x < 0) {
 transform.localScale = new Vector3(original.x * -1, original.y,
original.z);}
```

Kôd 5.1 Programski kod za promjenu orijentacije kretanja likova

Kao što se može vidjeti iz prethodnog isječka koda (Kôd 5.1), prvo se u varijablu `original` sprema originalna orijentacija glavnog lika, namijenjena za kretanje u desno. Ako je x koordinata vektora za kretanje veća od 0, znači da se lik kreće po pozitivnoj x osi (s lijeva na desno), zbog čega mu se zadržava originalna orijentacija, a ako je taj broj negativan, onda mu je orijentaciju potrebno promijeniti množenjem sa -1. Pomoćne varijable `yAxisSpeed` and `xAxisSpeed` služe za lako upravljanje brzinom kretanja po koordinatnim osima, konkretno u ovom slučaju je brzina kretanja po Y osi 25% sporija nego kretanje po X osi. Kako bi se doznao smjer kretanja (i da li uopće treba pomaknuti lika), potrebno je stalno voditi računa o interakciji igrača s glavnim likom. U tu svrhu postoje već dvije unaprijed definirane metode unutar Unity API-a, pod imenima `Update` i `FixedUpdate`. Sav programski kod smješten unutar tih metoda se prema potrebi izvršava nekoliko puta u sekundi (ovisno o brzini izvođenja igre i količini prikaza slika u sekundi), zbog čega je vrlo koristan za ovakve provjere.

```
float x = Input.GetAxisRaw("Horizontal");
float y = Input.GetAxisRaw("Vertical");
UpdateMover(new Vector3(x, y, 0));
```

Prethodni isječak koda je smješten unutar `FixedUpdate` metode u programskoj skripti vezanoj uz glavni lik igre. To znači da će se nekoliko puta u sekundi dohvaćati x i y koordinate smjera

ulaznih parametara (npr. ako igrač pritisne strelicu za desno na tipkovnici, to se smatra ulaznim parametrom s pozitivnom x osi), te automatski pozivati metoda za kretanje s tim parametrima, čiji je glavni kod već prikazan ranije (Kôd 5.1).

Još jedna vrlo bitna stvar prije omogućavanja kretanja je provjera kolizija. Ako se igrač želi kretati u smjeru koji ga dovodi u koliziju s neprohodnim objektima (npr. zid), potrebno je zaustaviti kretanje. Najčešći način provjera kolizije prilikom razvoja računalnih igara je pomoću tehnike nazvane projekcija zraka (engl. *ray cast*). Ona uključuje analizu kretanja određene zamišljene zrake (npr. zraka svjetlosti) u željenom smjeru, te praćenje svih površina s kojima ona dolazi u kontakt (Madhav, 2014). Unity također sadrži objekte `RaycastHit2D` klase, koji omogućavaju projekciju kretanja objekta u nekom smjeru, te potencijalnu detekciju sudara tog objekta s nekim drugim objektom, pod uvjetom da oba dva objekta imaju definirane *collider* komponente.

```
protected RaycastHit2D hit;
boxCollider = GetComponent<BoxCollider2D>();
hit = Physics2D.BoxCast(transform.position, boxCollider.size, 0, new
Vector2(0, moveVector.y), Mathf.Abs(moveVector.y * Time.deltaTime),
LayerMask.GetMask("Actor", "Blocking"));
```

Prethodni odsječak koda pokazuje provjeru slobode kretanja po vertikalnoj osi pomoću projekcije. Ako ova projekcija dovede do sudara s bilo kojom *collider* komponentom objekta pridruženog actor ili blocking sloju, vratit će povratne informacije o sudaru. Budući da se glavni lik može kretati i po horizontalnoj osi, identična provjera je napravljena i za taj slučaj (korištenjem `moveVector.x` varijable). Ako varijabla `hit` ostane prazna, znači da nema sudara, te je moguće dopustiti kretanje lika po vertikalnoj osi.

```
if (hit.collider == null) {

 transform.Translate(0, moveVector.y * Time.deltaTime, 0); }
```

Kretanje, kao i sve ostale radnje u igri, odvijaju se u stvarnom vremenu, što znači da je potrebno simulirati kretanje u dimenziji vremena, a ne samo prostora. Upravo zbog toga je u prethodnom odsječku koda `y` koordinata vektora kretanja pomnožena sa `Time.deltaTime` varijablom, koja označava vrijeme u sekundama potrebno za grafički prikaz određene radnje. Ako primjerice vektor kretanja po vertikalnoj osi iznosi 5, množenjem s ovom varijablom zapravo govori igri da je potrebno pomaknuti glavnog lika za 5 piksela sjeverno, i to u jednoj sekundi.

5.3.3. Protivnici

Protivnici u igri trebaju imati slične mogućnosti i funkcionalnosti kao i glavni lik, zbog čega i *prefab* koji definira protivnike koristi većinu identičnih komponenti kao i glavni lik. Glavna razlika je u samom kretanju, budući da kretanjem glavnih likova upravljaju direktno igrači preko tipkovnice, dok ponašanje protivnika kontrolira računalo, u čiju svrhu je napravljena `Enemy.cs` skripta pridodana svakom protivniku. Ona također nasljeđuje `Mover` klasu, jer su protivnici zamišljeni kao objekti koji se kreću unutar igre.

```
startingPosition = transform.position;
playerTransform = GameManager.instance.player.transform;
public float triggerLength = 1;
public float chasingLength = 5;
private bool isChasing;
if (Vector3.Distance(playerTransform.position, startingPosition)
< chasingLength) {
 if(Vector3.Distance(playerTransform.position, startingPosition)
< triggerLength) {
 isChasing = true; }
 if (isChasing) {
  UpdateMover((playerTransform.position -
transform.position).normalized); }
 else{
  UpdateMover(startingPosition - transform.position); }}
```

Kôd 5.2 Programski kod za upravljanje kretanjem protivnika

Prikazani dio koda (Kôd 5.2) je smješten unutar FixedUpdate metode (što znači da se izvodi više puta u sekundi), te koristi niz pomoćnih varijabli za upravljanje kretanjem protivnika. Varijabla `startingPosition` pamti početnu poziciju svakog protivnika, `playerTransform` označava glavnog lika, varijable `triggerLength` i `chasingLength`, dok varijabla `isChasing` označava da li se protivnik trenutno treba kretati prema igraču. Ako udaljenost između pozicije igrača i pozicije protivnika postane manja od zadanih pomoćnih varijabli, protivnik se počinje kretati prema glavnom liku. Ako glavni lik uspješno pobjegne od protivnika

(udaljenost mu postane veća od zadanih varijabli), protivnik prestaje kretanje i vraća se na svoju početnu poziciju.

5.3.4. Kamera

Glavna funkcionalnost kamere (osim samog prikaza scene), je njezino automatsko praćenje glavnog lika. Bez toga, lik bi mogao izaći izvan kadra, što bi učinilo igru gotovo neigrivom i nefunkcionalnom. Zbog toga je kameri pridodana skripta naziva `CameraControl`, čija je osnovna funkcionalnost prikazana u idućem dijelu programskog koda (Kôd 5.3).

```
Vector3 delta = Vector3.zero;
Transform target = GameObject.Find("Player").transform;
float borderX = 0.15f;
float borderY = 0.05f;
float deltaX = target.position.x - transform.position.x;
 if (deltaX > borderX || deltaX < - borderX) {
  if (transform.position.x < target.position.x) {
   delta.x = deltaX - borderX; }
  else {
   delta.x = deltaX + borderX; }}
float deltaY = target.position.y - transform.position.y;
 if (deltaY > borderY || deltaY < - borderY) {
  if (transform.position.y < target.position.y) {
   delta.y = deltaY - borderY; }
  else {
   delta.y = deltaY + borderY; }}
transform.position += new Vector3(delta.x, delta.y, 0);
```

Kôd 5.3 Programski kod za automatsko praćenje kamere

Pomoćne varijable `borderX` i `borderY` služe kao zamišljene granice koje glavni lik ne smije prijeći. Zbog toga je potrebno prvo naći matematičku razliku između koordinata igrača i koordinata kamere. Ako je ta razlika u koordinatama veća od granica (bilo u pozitivnoj ili negativnoj vrijednosti), potrebno je pomaknuti kameru u pozitivnom ili negativnom smjeru te osi (ovisno da li se lik kretao u lijevo ili u desno) za iznos razlike i granice, kako bi glavni lik opet bio

centriran. Navedene provjere i radnje je potrebno izvršiti dva puta, jednom za svaku os potencijalnog kretanja kamere i likova.

5.3.5. Kovčeg sa zlatom

Pronalazak kovčega sa zlatom je jedan od glavnih ciljeva igre, te predstavlja jedini način za prikupljanje zlatnika korištenih za kupovinu bolje opreme. Kovčezi su izrađeni na vrlo sličan način kao već prethodno opisani elementi glavnog lika i protivnika, a glavna razlika između njih leži u programskom kodu koji je definiran unutar `Chest` skripte pridružene ovom elementu. Osnovna jednostavna logika upravljanja kovčezima uključuje detektiranje kolizije između glavnog lika i kovčega. Ako se taj sudar stvarno dogodi, potrebno je zamijeniti trenutni grafički prikaz kovčega sa zlatom s praznim kovčegom (kako bi grafički bilo prikazano da je igrač već pokupio zlatnike), te pridodati igraču količinu zlatnika koja je bila u kovčegu.

Za detektiranje kolizija Unity API nudi metodu `OnCollide` koja se automatski izvrši svaki put kada dva objekta s definiranim collider komponentama dođu u međusobni kontakt.

```
protected override void OnCollide(Collider2D coll) {
 if (coll.name == "Player"){
  OnCollect();}}
```

Iz prethodnog koda se može vidjeti primjer korištenja te metode. Ako se u ovom slučaju detektira kolizija između objekta koji sadrži skriptu u kojoj se poziva ova metoda (u ovom slučaju je to kovčeg) i *collider* komponente pridružene objektu naziva `Player` (glavni lik), poziva se pomoćna metoda `OnCollect`.

```
protected override void OnCollect(){
 GetComponent<SpriteRenderer>().sprite = emptyChest;
 GameManager.instance.gold += goldAmount; }
```

Unutar ove se metode zatim upravlja akcijama potrebnim za interakciju između glavnog lika i kovčega, što uključuje zamjenu grafičkog prikaza kovčega, i dodjeljivanja zlatnika igraču. Za dodjelu zlatnika se koriste pomoćne metode unutar GameManager klase o kojoj će više riječi biti kasnije.

5.3.6. Prepreke

U igri se koriste dvije osnovne vrste prepreka – zidovi i kutije. Glavna namjena prepreka je sprečavanje i ograničavanje kretanja glavnog lika (ali i protivnika), a razlika između zidova i kutija je u tome što je kutije moguće uništiti korištenjem oružja, dok su zidovi potpuno neuništivi i ne podržavaju bilo kakvu interakciju s likom. Zbog toga su jedine komponente koje je potrebno definirati na zidu *Sprite Renderer* (za grafički prikaz), te *Box Collider 2D* (kako bi se detektirala kolizija s likovima, i time spriječilo njihovo kretanje kroz zidove). Budući da se kolizijama upravlja iz skripti definiranih u sklopu elemenata igrača i protivnika, nije potrebno pridruživati ikakav programski kod ovoj prepreci.

Kutije igrač može uništiti, te je ovu funkcionalnost nužno implementirati preko koda sadržane definiranog unutar `Crate` skripte. Sama mehanika uništavanja objekata je zajednička svim sudionicima borbe unutar igre, zbog čega je napravljena već prethodno spomenuta pomoćna klasa `Fighter`, pa tako i kutija može koristiti tu programsku logiku ako naslijedi tu klasu.

```
protected virtual void ReceiveDamage(Damage dmg) {
 hitpoint -= dmg.damageAmount;
 if (hitpoint <= 0) {
   hitpoint = 0;
   Death();}}
```

Prethodni programski kod pokazuje osnovnu logiku borbe unutar metode `ReceiveDamage` smještene u `Fighter` pomoćnoj klasi. Prilikom svakog kontakta dva `Fighter` objekta, napadač šalje informacije o učinjenoj šteti prema napadnutom objektu putem `ReceiveDamage` metode. Tada se bodovi koji predstavljaju zdravlje objekta (`hitpoint` varijabla) smanjuju za učinjenu štetu, a ako oni padnu na iznos manji ili jednak nuli, napadnuti objekt treba uništiti. U tu svrhu se poziva `Death` metoda.

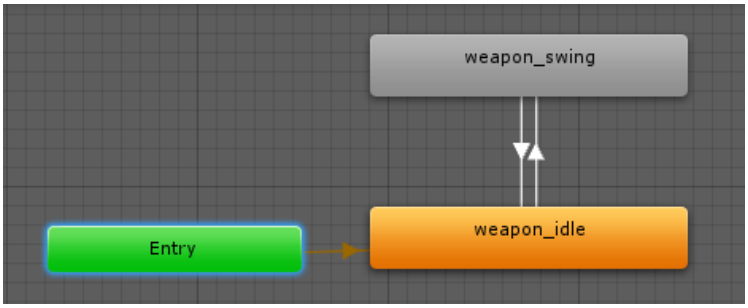
```
protected override void Death(){
 Destroy(gameObject);  }
```

Unutar nje se poziva predefinirana metoda Unity API-a `Destroy`, koja uklanja proslijeđeni objekt sa scene.

5.3.7. Oružje, animacija i mehanika borbe

Borba između glavnog lika i njegovih protivnika odvija se korištenjem oružja koja su izrađena kao zasebni elementi unutar igre. Slično kao i kod većine prethodno opisanih elemenata igre, oružje mora imati *Sprite Renderer* i *Box Collider 2D* komponente na sebi, budući da se mora grafički iscrtavati na sceni, te je potrebno detektirati potencijalne sudare između oružja i „napadnutih" elemenata igre, poput protivnika ili kutija. Oružje pripada glavnom liku, te treba stalno biti smješteno uz njega, zbog čega je element koji predstavlja oružje definiran kao element dijete glavnog lika, čime se osigurava njihova povezanost.

Još jedna bitna komponenta potrebna oružju je i *Animator*, koji omogućava definiranje različitih animacija vezanih uz ovaj objekt, te njihovo pokretanje pomoću programskog koda iz skripte. Borba bi bila moguća i bez animacija oružja, ali u tom slučaju bi vizualno izgledalo kao da nevidljiva sila napada protivnike (jer se oružje ne kreće u prostoru), što ne bi bilo dobro za igrivost i preglednost igre. Izrada animacija za 2D objekte u Unity-u je jednostavna, te se može podijeliti u dva osnovna koraka – izrada svojevrsnog konačnog automata stanja (engl. *state machine*) pomoću *Animator* alata, te izrada samih animacija u sklopu *Animation* alata. Animator omogućava definiciju stanja koja željeni objekt treba poprimiti, te definiciju uvjeta i mehanizama kada i kako se obavlja promjena između tih stanja. Pošto je jedina funkcija oružja zamahivanje odnosno napad, dovoljna su dva osnovna stanja – stanje mirovanja i stanje zamaha.



Slika 5.6 Definiranje prijelaza između stanja u animatoru

Strelice između stanja prikazane na prethodnoj slici (Slika 5.6), označavaju uvjete za prelazak između tih stanja. Zamah oružjem treba se dogoditi samo ako je igrač pritisnuo neku tipku, te je stoga definiran okidač (engl. *trigger*) naziva `Swing`, koji pokreće animaciju zamaha. Taj je okidač, ako se detektira da je odgovarajuća tipka pritisnuta od strane igrača, namijenjen za direktno pozivanje iz programskog koda. Prelazak iz stanja zamahivanja natrag u stanje mirovanja nema definiran ikakav okidač ili poseban uvjet, što znači da će čim završi animacija oružje automatski otići u stanje mirovanja. Nakon definiranja stanja i prelazaka između njih, potrebno je izraditi samu animaciju za `weapon_swing` stanje unutar Animation alata. Animacije se definiraju na vremenskoj crti koja sadrži sekunde, te nije potrebno definirati posebno stanje za svaku jedinicu vremena. Dovoljno je odrediti početnu i konačnu poziciju objekta kojeg se animira (u ovom slučaju je to identična pozicija - vertikalno usmjereno oružje uz tijelo glavnog lika), te na sredini vremenske crte odrediti željeno konačno stanje (promjena rotacije oružja tako da bude horizontalno ispruženo), a Unity će sam animirati i prikazati promjenu između tih stanja, što u stvarnosti izgleda kao jedan zamah oružja, te povratak u početnu poziciju. Budući da oružje služi za napad, a on je reguliran pomoću detekcija sudara između oružja i nekog drugog objekta, potrebno je tijekom animacije promijeniti i *collider* komponentu oružja, tako da prati promjene rotacije oružja prilikom zamaha.

Za kontrolu i definiranje funkcionalnosti oružja izrađena je `Weapon` programska skripta. Prva i najvažnija mogućnost oružja je napad (odnosno zamahivanje oružjem), koje se izvodi kada igrač pritisne željenu tipku na tipkovnici, a provjeru da li je uopće potrebno zamahnuti oružjem može se konstantno obavljati unutar Update metode (slično kao što se i vršila provjera kretanja kod likova).

```
protected override void Update(){
 if (Input.GetKeyDown(KeyCode.Space)) {
  Swing();}}
```

Prethodni dio koda pokazuje jednostavnu detekciju pritiska razmaknice na tipkovnici (`KeyCode.Space` označava razmaknicu unutar Unity API-a), te ako je pritisnuta tipka stvarno detektirana, poziva se `Swing` metoda koja upravlja zamahivanjem oružja. Unutar te metode je dovoljno pozvati okidač koji pokreće animaciju zamahivanja oružjem.

```
Animator anim = GetComponent<Animator>();
 anim.SetTrigger("Swing");
```

Okidač `Swing` je definiran kao uvjet za promjenu stanja oružja iz mirovanja u zamah (Slika 5.6), te izvođenje animacije zamaha. Kako bi se upravljalo samom mehanikom borbe, potrebno je detektirati sudar između oružja i protivnika, što se može napraviti unutar predefinirane metode `OnCollide`.

```
protected override void OnCollide(Collider2D coll){
   if (coll.tag == "Fighter"){
    coll.SendMessage("ReceiveDamage", 1); }}
```

Kao što se može vidjeti iz prethodnog koda, ako oružje dođe u sudar s objektom koji ima collider komponentu, i čiji naziv (odnosno *tag*) je `Fighter`, pozvat će se predefinirana metoda iz Unity API-a naziva `SendMessage`. Ovako je vrlo jednostavno regulirati koji objekti mogu sudjelovati u borbi, jer je dovoljno dodati prethodni tag na sve takve objekte (poput protivnika ili kutija). `SendMessage` metoda prima dva parametra, prvi je naziv metode koja se poziva unutar objekta koji je u ovom slučaju došao u koliziju s oružjem igrača, a vrijednost je parametar koju ta metoda prima. Primjer ReceiveDamage metode je već prikazan u prethodnom poglavlju o izradi protivnika (5.3.3).

5.3.8. Upravljanje igrom

Često nije lagano upravljati različitim elementima igre, te omogućiti njihovu interakciju i pravovremeno pozivanje programskog koda i metoda iz skripti pridruženih tim objektima. Jedan od najlakših i najčešće korištenih načina za upravljanje izvođenjem igre i njezinim komponentama je izrada posebne komponente koja će preuzeti ulogu menadžera ili glavnog upravitelja igrom. U tu svrhu je napravljen element igre nazvan `GameManager`, koji će biti prisutan na svakoj sceni i u svakoj situaciji. Budući da je to element igre koji radi „iz pozadine“, odnosno nije direktno izložen prema igračima, ne treba imati ikakve vizualne komponente na sebi ili grafički prikaz na sceni, već jedino programsku skriptu koja određuje njegovo ponašanje. Unutar programske skripte (također nazvana `GameManager`) zatim se definiraju metode koje kontroliraju generalni tok izvođenja igre, poput promjene scene, prikaz elemenata grafičkog sučelja igre (npr. koliko štete je

igrač primio od protivnika tijekom borbe), spremanje i učitavanje igre, i mnogo drugih potrebnih funkcionalnosti.

```
public void SaveState(){
 string save = "";
 save += "0" + "|";
 save += gold.ToString() + "|";
 save += experience.ToString() + "|";
 PlayerPrefs.SetString("SaveState", save); }
```

Kôd 5.4 Programski kod za spremanje trenutnog stanja igre

Prethodni programski kod (Kôd 5.4) pokazuje primjer spremanja trenutnih informacija o igri, kako bi ih kasnije mogli ponovo učitati. U ovom slučaju definira se prazna tekstualna varijabla `save`, u koju se zatim zapisuju sve željene informacije u određenom formatu, poput količine zlatnika koju igrač posjeduje, te njegovu razinu iskustva i oružja. Sve se informacije prilikom zapisa odvajaju tekstualnim znakom '|', kako bi ih kasnije prilikom učitavanja bilo lakše razdvojiti. Nakon što su sve informacije pohranjene u varijablu, dovoljno je iskoristiti gotovu metodu `SetString` unutar predefinirane klase Unity API-a nazvane `PlayerPrefs`, koja omogućava pohranu varijabli pod nekim nazivom (koristi se rječnik odnosno *dictionary* struktura podataka). Ovako definirane metode spremanja stanja unutar `GameManager` skripte mogu se pozivati bilo kada (na pritisak određene tipke, prilikom promjene scene, prilikom prvog pokretanja igre, i sl.), sve ovisno o potrebama i generalnom dizajnu igre. Kasnije, prilikom učitavanja, dovoljno je dohvatiti spremljene vrijednosti, i primijeniti ih ponovno na glavni lik.

```
String load = PlayerPrefs.GetString("SaveState");
```

Trenutno zdravlje glavnog lika je prikazano u sklopu generalnog korisničkog sučelja igre, pomoću vertikalne crvene linije. Svaki put kada igrač bude napadnut od strane protivnika i time „primi štetu“, potrebno je osvježiti prikaz tog dijela sučelja, što je također moguće vrlo lako definirati unutar glavnog upravitelja igrom.

```
public void OnHitpointChange(){
 float ratio = (float)player.hitpoint / (float)player.maxHitpoint;
 hitpointBar.localScale = new Vector3(1, ratio, 1); }
```

Ovako definiranu metodu unutar `GameManager` skripte dovoljno je pozvati jednom, prilikom primanja štete, kako bi se osvježilo sučelje. Prvo se izračuna omjer između trenutnog zdravlja igrača i maksimalne vrijednosti, i zatim se `hitpointBar` elementu na korisničkom sučelju (definiran kao vertikalna crvena linija) promijeni veličina ovisno o toj izračunatoj vrijednosti.

6. Algoritmi za proceduralno generiranje sadržaja

Zadaća algoritama za proceduralno generiranje sadržaja u igri je korištenje ručno izrađenih elemenata igre opisanih u prethodnom poglavlju, te na određeni način automatsko kreiranje razine koje igrač mora istraživati u potrazi za blagom. To uključuje raspoređivanje prepreka, kreiranje protivnika i kovčega s blagom, te generiranje izlaza koji vraća glavni lik natrag u početnu prostoriju dvorca koji istražuje. Kako bi se demonstrirale mogućnosti proceduralnog kreiranja razina unutar 2D igre u Unity-u, izrađeno je pet različitih algoritama koji će biti detaljnije opisani u nastavku ovog poglavlja:

- Board algoritam
- Infinity algoritam
- Pathfinder algoritam
- Tilemap algoritam
- Rogue algoritam

Programski će kod svakog algoritma biti pohranjen u zasebnoj programskoj skripti „rezerviranoj" za njega, a svaka od tih skripti bit će pridružena kao komponenta jednom praznom objektu. Tako definirani *prefab* elementi (prazni objekti s programskom skriptom) zatim će biti pridodani glavnom upravitelju igrom – `GameManager` objektu, koji će po potrebi pozivati jedan od tih algoritama i upravljati generiranjem sadržaja. Za potrebe svakog algoritma također će biti napravljena i zasebna scena na kojoj se on koristi, kako bi se lakše razdvojilo njihovo ponašanje. Budući da su algoritmi zaduženi za kreiranje razina, idealno vrijeme za njihovo korištenje je prilikom učitavanja nove scene (kako bi kreirali sav sadržaj u njoj). Prelazak na novu scenu dešava se kada glavni lik dođe u kontakt sa stepenicama, smještenim na početnoj glavnoj razini dvorca, na kojima se nalazi nevidljivi objekt s pridruženom *Box Collider 2D* komponentom, te programskom skriptom za učitavanje novih scena.

```
public string[] sceneNames;

protected override void OnCollide(Collider2D coll){

 if (coll.name == "Player"){
```

```
string sceneName = sceneNames[Random.Range(0, sceneNames.Length)];
SceneManager.LoadScene(sceneName); }}
```

Varijabla `sceneNames` sadrži imena svih scena unutar igre, te ako dođe do kolizije s glavnim likom, slučajnim odabirom se izabire jedna od tih imena. `SceneManager` klasa je predefinirana klasa Unity API-a, te sadrži gotove metode za jednostavno upravljanje scenom. LoadScene metoda kao ulazni parametar prima ime scene, te izvršava učitavanje scene identičnog naziva, nakon čega algoritam može početi s proceduralnim generiranjem.

6.1. Board algoritam

Ovaj algoritam je inspiriran izgledom klasičnih podloga i ploča (engl. *game board*) za igre na poteze i različite društvene igre (npr. ploča za šah), zbog čega je i dobio naziv Board. Za pokretanje i upravljanje ovim algoritmom (ali i svim ostalima) je zadužena `OnSceneLoaded` metoda unutar GameManager elementa.

```
public void OnSceneLoaded(Scene s, LoadSceneMode mode) {
 if (s.name == "Board"){
  boardController.SetupScene(); }}
```

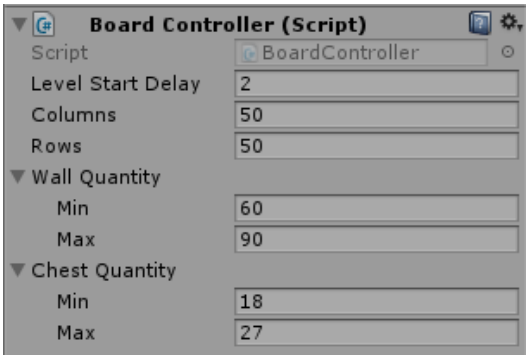
Ova metoda se automatski poziva nakon svakog učitavanja scene putem `SceneManager` klase (prikazano u prošlom poglavlju), te ako se detektira da se učitava scena određenog naziva, pokreće se izvođenje odgovarajućeg algoritma koji je zadužen za kreiranje elemenata na toj sceni. Board algoritam je ujedno i najjednostavniji algoritam korišten za proceduralno generiranje sadržaja, te se njegova funkcionalnost može prikazati u tri osnovna koraka. Prvi korak uključuje generiranje neprohodnih prepreka oko čitave razine, kako bi se spriječila mogućnost da likovi izađu izvan granica razine.

```
public int columns = 16;
public int rows = 16;
public List<GameObject> floors;
public List<GameObject> outWalls;
private Transform boardMain;
void GameBoardSetup(){
```

```
 boardMain = new GameObject("Board").transform;
 OutWallSetup();}
private void OutWallSetup(){
 for (int x = -1; x < columns + 1; x++){
  for (int y = -1; y < rows + 1; y++){
   int randomFloorIndex = Random.Range(0, floors.Count);
   int randomOutWallIndex = Random.Range(0, outWalls.Count);
   GameObject toCreate = floors[randomFloorIndex];
   if (x == -1 || y == -1 || x == columns || y == rows) {
    toCreate = outWalls[randomOutWallIndex]; }
    GameObject instance = Instantiate(toCreate, new Vector3(x *
0.16f, y * 0.16f, 0f), Quaternion.identity) as GameObject;
    instance.transform.SetParent(boardMain); }}}
```

Kôd 6.1 Programski kod Board algoritma za generiranje vanjskog okvira i podloge razine

Pomoćne varijable definirane na početku prethodnog koda (Kôd 6.1) koriste se prilikom izvođenja algoritma, te omogućavaju lakšu kontrolu i podešavanje samog algoritma. Ovako definirane varijable moguće je podešavati direktno iz samog Unity alata, što znatno olakšava testiranje algoritama i dodavanje novih elemenata. Kao što se može vidjeti na slici (Slika 6.1), sve pomoćne varijable koje su javno dostupne (definirane kao `public` unutar koda) automatski se pojavljuju na Unity objektu koji sadrži pripadajuću programsku skriptu, te ih je moguće proizvoljno mijenjati ček i tijekom samog izvođenja igre. Ako se žele promijeniti dimenzije ploče, dovoljno je promijeniti vrijednosti `columns` i `rows` parametara, i algoritam će automatski generirati razinu drugačijih dimenzija.



Slika 6.1 Ručna promjena ulaznih parametara algoritma u Unity-u

Unutar `GameBoardSetup` metode (Kôd 6.1) poziva se metoda `OutWallSetup` koja kreira vanjske granice ploče. Budući da se ploča sastoji od unaprijed određenog broja stupaca i redaka, pomoću `for` petlji se prolazi kroz sve elemente (ćelije) ploče, te se na svakoj poziciji kreira slučajno odabrani element podloge iz `floors` kolekcije, što je određeno pomoću `randomFloorIndex` varijable, koja predstavlja slučajno odabrani indeks iz te kolekcije. Ukoliko trenutno odabrana ćelija pripada jednom od četiri ruba ploče (ako su `x` ili `y` koordinate jednake -1, ili pak zadnjoj koordinati stupca ili retka), umjesto podloge potrebno je kreirati vanjsku prepreku odnosno zid, koji je također izabran slučajno iz `outWalls` kolekcije elemenata koji sadrže zidove. Kreiranje objekta izvršava se pozivanjem predefinirane `Instantiate` metode, kojoj je potrebno proslijediti objekt kojeg se želi kreirati (u ovom slučaju varijabla `toCreate`), te koordinate na kojima će taj novi objekt biti smješten. `Quaternion.identity` se često koristi prilikom kreiranja 2D objekata, kako bi se definiralo da oni nemaju početnu rotaciju oko treće osi.

U idućem koraku algoritma, potrebno je spremiti lokacije svake ćelije kreiranih u prethodnom koraku, kako bi se na njihovim lokacijama kasnije mogli kreirati ostali elementi igre. Nakon izvršavanja tog dijela koda prikazanog u nastavku (Kôd 6.2), varijabla grid će sadržavati vektorske koordinate svake ćelije na ploči, pomnožene sa 0.16, jer svi elementi korišteni za grafički prikaz objekata unutar igre su veličine 16x16 piksela.

```
private List<Vector3> grid = new List<Vector3>();
void InitializeGrid(){
 grid.Clear();
 for (int x = 1; x < columns - 1; x++){
  for (int y = 1; y < rows - 1; y++){
  Vector3 position = new Vector3(x * 0.16f, y * 0.16f, 0f);
  grid.Add(position); }}}
```

Kôd 6.2 Kod za spremanje lokacije svake ćelije na ploči unutar Board algoritma

Treći korak algoritma prolazi po svim ćelijama na ploči, te ovisno o slučajno generiranim brojevima odabire jednu ćeliju, i na njoj kreira željeni objekt (npr. protivnika ili kovčeg sa zlatom).

```
void CreateRandomObjects(List<GameObject> tiles, int min, int
max) {
```

```
 int objectQuantity = Random.Range(min, max + 1);
 for (int i = 0; i < objectQuantity; i++){
  Vector3 randomVector = RandomVector();
  GameObject tile = tiles[Random.Range(0, tiles.Count)];
  Instantiate(tile, randomVector, Quaternion.identity); }}
Vector3 RandomVector(){
 int randomIndex = Random.Range(0, grid.Count);
 Vector3 randomVector = grid[randomIndex];
 grid.RemoveAt(randomIndex);
 return randomVector; }
```

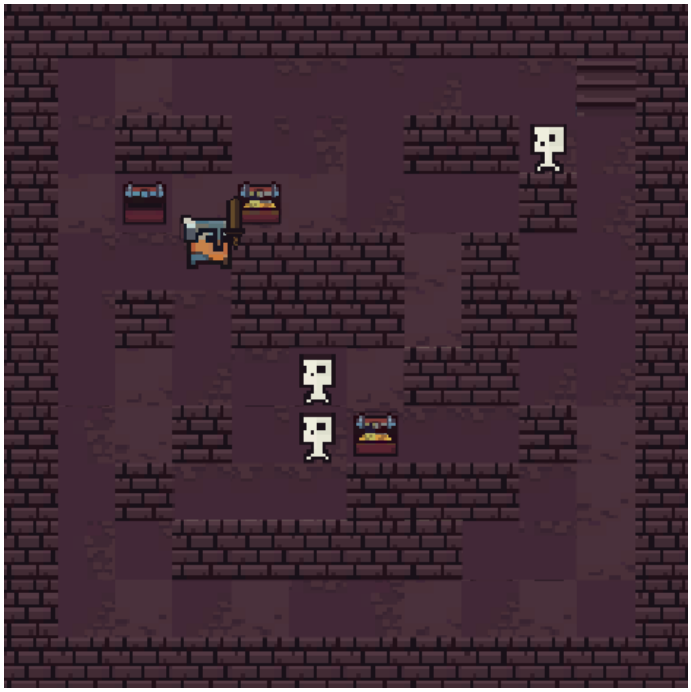
Kôd 6.3 Kreiranje elemenata igre u Board algoritmu slučajnim odabirom

Varijabla `objectQuantity` predstavlja količinu objekata koji će se kreirati (ovisno o `min` i `max` ulaznim parametrima metode), a varijabla `randomVector` sadrži slučajno odabranu poziciju na ploči korištenjem pomoćne `RandomVector` metode. Varijabla `tile` sadrži slučajno odabrani element iz kolekcije ulaznih parametara `tiles` kojeg je potrebno kreirati na ploči. Metode iz ovog dijela koda (Kôd 6.3) mogu se koristiti za proceduralno generiranje svih potrebnih elemenata igre, dovoljno je jedino proslijediti drugačiju kolekciju objekata koje se želi generirati u `CreateRandomObjects` metodu. Jedina preostala stvar je pozvati sve prethodno opisane korake u ispravnom redoslijedu, što je određeno unutar `SetupScene` metode, čije se korištenje može vidjeti na početku ovog poglavlja. Ako se želi slučajno generirati određeni broj protivnika, dovoljno je u sklopu te metode pozvati CreateRandomObjects metodu i proslijediti joj kolekciju objekata koji predstavljaju protivnike.

Za pozicioniranje igrača na scenu, kreira se početna točka na određenoj lokaciji (npr. prva ćelija na ploči), te se postavlja pozicija glavnog lika na te koordinate. Na identičan način se može kreirati i pozicionirati izlaz iz nivoa.

```
GameObject spawn;
Instantiate(spawn, new Vector3(1 * 0.16f, 1 * 0.16f, 0F),
Quaternion.identity);
GameObject.Find("Player").transform.position =
spawn.transform.position;
```

Na slici (Slika 6.2) može se vidjeti primjer razine sastavljene od 100 ćelija (10 redaka i stupaca), generirane korištenjem Board algoritma. Kako je kreiranje i pozicioniranje elemenata (osim vanjskog ruba ploče) određeno pomoću slučajnih brojeva, svaka razina stvorena pomoću algoritma će biti drugačija.



Slika 6.2 Primjer nasumično kreirane razine pomoću Board algoritma

6.2. Infinity algoritam

Infinity algoritam se tako zove jer omogućava gotovo pa beskonačno generiranje elemenata unutar igre. Glavno svojstvo ovog algoritma je što se pomoću njega kreiranje sadržaja vrši tijekom samog izvođenja igre, a ne unaprijed kao kod ostalih algoritama. Osnovni koncept algoritma se zasniva na definiranju određene površine ispred glavnog lika, koju se može zamisliti kao vidno polje igrača. Kako se glavni lik pomiče u jednom od četiri smjera, tako se unutar dimenzija tog vidnog polja slučajnim odabirom generiraju elementi igre. Sve izvan vidnog polja igrača ostaje „prazno“, odnosno na tim pozicijama nije ništa generirano. Ako se igrač počne kretati u smjeru koji je već prethodno bio istražen (u vidno polje uđu elementi koji su već kreirani ranije), treba spriječiti novo kreiranje elemenata, jer bi inače igra bila vrlo nekonzistentna i teško igriva. Za pohranu već

kreiranih elemenata odabran je rječnik (engl. *dictionary*) kao struktura podataka, pošto podržava laganu pretragu po ključevima za brz pronalazak elemenata.

```
public void BoardSetup(){
 boardHolder = new GameObject("Board").transform;
 for (int x = 0; x < columns; x++){
  for (int y = 0; y < rows; y++) {
   gridPositions.Add(new Vector2(x * pixelMulti, y *
pixelMulti), new Vector2(x * pixelMulti, y * pixelMulti));
   GameObject toInstantiate = floorTiles[Random.Range(0,
floorTiles.Length)];
   GameObject instance = Instantiate(toInstantiate, new
Vector3(x * pixelMulti, y * pixelMulti, 0f),
Quaternion.identity) as GameObject;
   instance.transform.SetParent(boardHolder); }}
   GameObject.Find("Player").transform.position = new
Vector3((columns / 2) * pixelMulti, (rows / 2) * pixelMulti,
0f); }
```

Kôd 6.4 Programski kod za kreiranje početnog stanja Infinity algoritma

Metoda `BoardSetup` (Kôd 6.4) poziva se prilikom prvog pozivanja algoritma kako bi se postavilo početno stanje igre. Varijable `columns` i `rows` sadrže početnu veličinu ekrana koji će biti otkrivena prilikom učitavanja scene, te se stoga na tim pozicijama kreiraju slučajno odabrani elementi podloge (kolekcija `floorTiles`). Svaku lokaciju na kojoj se kreira element igre (u ovom slučaju podloga) potrebno je dodati u rječnik `gridPositions`, koji sadrži sve kreirane objekte na sceni. Nakon kreiranja podloge i spremanja svih lokacija, postavlja se početna pozicija glavnog lika na sredinu početne površine. Vidno polje glavnog lika je određeno kao pravokutnik širine 2 i visine 3 elementa igre (svaki element igre ima dimenzije 16x16 piksela), te se unutar njega tijekom kretanja igrača stalno provjeravaju postojeći elementi, i prema potrebi kreiraju novi.

```
public void addToBoard(int horizontal, int vertical) {
 GameObject player = GameObject.Find("Player");
 float transX = (player.GetComponent<Player>().transX) *
pixelMulti;
 float transY = (player.GetComponent<Player>().transY) *
pixelMulti;
```

```
if (horizontal == 1) {
 float x = transX;
 float sightX = x + (2 * pixelMulti);
 for (x += pixelMulti; x <= sightX; x += pixelMulti) {
  float y = transY;
  float sightY = y + pixelMulti;
  for (y -= pixelMulti; y <= sightY; y+= pixelMulti) {
   addTiles(new Vector2(x, y)); }}}
```

Kôd 6.5 Dio infinity algoritma za proceduralno generiranje sadržaja prilikom kretanja u desno

Metoda `addToBoard`, prikazana na prethodnom isječku koda (Kôd 6.5), prihvaća dva parametra - horizontalni i vertikalni smjer kretanja igrača. Ako je horizontalna vrijednost jednaka 1, to znači da se igrač kreće u desno, te je stoga potrebno kreirati elemente unutar vidnog polja igrača smještenog desno od igrača. Varijable `sightX` i `sightY` definiraju granice tog vidnog polja, te se stoga pomoću njih može iterirati po elementima tog zamišljenog polja unutar `for` petlji. `X` i `Y` varijable unutar tijela svake petlje određuju jednu od šest mogućih pozicija unutar vidnog polja (pošto je ono definirano kao pravokutnik veličine 2x3), te se za svaku poziciju poziva metoda `addTiles`. Pomoćne varijable `transX` i `transY` služe za pretvaranje generalnih koordinata u točne koordinate u pikselima, uzimajući u obzir veličinu svakog elementa. Gotovo identična programska logika se koristi i za provjeru kretanja za preostala tri smjera, te poziva metode za kreiranje elemenata unutar vidnog polja.

```
private void addTiles(Vector2 tileToAdd) {
 if (!gridPositions.ContainsKey(tileToAdd)) {
  gridPositions.Add(tileToAdd, tileToAdd);
GameObject toInstantiate = floorTiles[Random.Range(0,
floorTiles.Length)];
 GameObject instance = Instantiate(toInstantiate, new
Vector3(tileToAdd.x, tileToAdd.y, 0f), Quaternion.identity) as
GameObject;
 instance.transform.SetParent(boardHolder); }}
```

Kôd 6.6 Proceduralno generiranje podloge u sklopu Infinity algoritma

Unutar metode `addTiles`, vidljive iz prethodnog dijela koda (Kôd 6.6) prvo se vrši provjera da li na zadanoj poziciji već postoji kreirani element (ako postoji, `gridPositions` kolekcija će

sadržavati ključ koji odgovara toj lokaciji). Ako ne postoji, trenutna koordinata se dodaje u tu kolekciju (kako se kasnije ne bi ponovno nešto kreiralo na toj lokaciji), te se slučajnim odabirom uzima jedan element podloge iz `floorTiles` kolekcije, te kreira na proslijeđenim koordinatama unutar scene. Sve se ostale elemente igre prema želji može kreirati na sličan način odmah unutar ove metode.

```
if (Random.Range(0, 9) == 1) {
 toInstantiate = wallTiles[Random.Range(0, wallTiles.Length)];
 instance = Instantiate(toInstantiate, new Vector3(tileToAdd.x,
tileToAdd.y, 0f), Quaternion.identity) as GameObject;
 instance.transform.SetParent(boardHolder); }
```

Prethodni isječak koda prikazuje primjer proceduralnog generiranja zidova na sceni. Kreiranje će se izvršiti samo ako slučajno generirani brojač u rasponu između 0 i 9 poprimi vrijednost 1, što u pravilu znači da će se u 10% slučajeva na mjestu podloge generirati prepreka izabrana iz `wallTiles` kolekcije. Slika (Slika 6.3) prikazuje primjer razine koja se generira pomoću ovog algoritma. Tamne površine označavaju neistraženi prostor, budući da na tim pozicijama nema kreiranih elemenata igre. Zeleni pravokutnik je dodan na sliku kako bi prikazao inače nevidljivo vidno polje igrača unutar kojeg se generira sadržaj. Kao što je vidljivo, igrač se kretao u desno, te stoga samo u tom polju ispred igrača postoji vidljiv sadržaj.

Slika 6.3 Dio razine generiran pomoću Infinity algoritma

6.3. Pathfinder algoritam

Pathfinder algoritam je dobio takav naziv jer funkcionira na principu pronalaska ispravnog puta između dvije pozicije ili točke unutar razine. U domeni teorije grafova postoji već mnogo gotovih algoritama čija je zadaća pronalazak optimalnog puta između dvije točke ili dva čvora na grafu, te je većina takvih algoritama adaptirana u druge svrhe, kao što je i izrada računalnih igara. Najpoznatiji primjer algoritma za pronalazak putanje je Dijkstra algoritam, razvijen 1956. godine, koji pronalazi najkraće putanje između određenog čvora na grafu, te svih ostalih postojećih čvorova na njemu. Zbog svoje efikasnosti pronašao je brojne uporabe i u sklopu informacijskih tehnologija, poput usmjeravanja mrežnog prometa unutar računalne mreže (Cormen et al., 2009).

Iako se ovaj i slični algoritmi mogu primijeniti prilikom proceduralnog generiranja razina unutar igre, oni bi uvijek stvorili najkraću moguću putanju između dvije točke. To bi rezultiralo vrlo sličnim i pomalo dosadnim razinama, jer se igrača ne bi poticalo na istraživanje ili skretanje s optimalnog puta. Zbog toga Pathfinder algoritam ne pronalazi najkraće ili optimalne putanje između točaka, već samo osigurava da neka putanja postoji (jer inače ne bi bilo moguće doći do kraja razine), a svi ostali aspekti će biti određeni pomoću slučajnih brojeva.

Ovaj je algoritam zamišljen za generiranje nivoa koji „teku“ sa zapada na istok, te se stoga na početku slučajnim odabirom definira početna pozicija negdje uz krajnji lijevi rub razine, a isto se tako odabire i krajnja točka uz desni rub. Idući se korak odnosi na pronalazak i generiranje putanje između te dvije točke. Kako bi se postiglo što više varijacija unutar putanje, algoritam svaki put može skrenuti i sjeverno ili južno, a ne samo vršiti kretanje na istok prema krajnjoj točki. Ovako kreirane razine, koje bi se sastojale samo od jedne putanje prema izlazu, i dalje bi bile vrlo dosadne. Kako bi se to izbjeglo, algoritam još jednom prolazi po prethodno određenoj putanji, te slučajnim odabirom kreira sporedne putanje (grane) unutar razine. Nakon toga se na svim pozicijama koje nisu unutar određenih putanja generiraju neprohodne prepreke kako bi se stvorila svojevrsna ograda oko kreirane razine. Na kraju je moguće ponovno prolaziti po svim točkama putanje i slučajnim odabirom na te pozicije stavljati ostale željene elemente igre, poput protivnika. Sada će svaka razina generirana ovim algoritmom osim glavne putanje imati i čitav niz sporednih putanja koji se granaju iz nje, što rezultira mnogo zanimljivijim ali i otvorenijim razinama, koje potiču igrače na skretanje s optimalnog puta.

Programski kod algoritma nalazi se unutar `PathfinderDungeonManager` programske skripte, koja je pridružena *prefab* elementu naziva `Pathfinder`. Za pohranu elemenata igre odabran je rječnik, u kojem će ključevi predstavljati poziciju elementa u prostoru, a vrijednost ključa će sadržavati vrstu elementa na toj poziciju.

```
public Dictionary<Vector2, TileType> pathPositions = new
Dictionary<Vector2, TileType>();
```

Kako bi se olakšalo praćenje vrste elemenata na pozicijama, napravljena je pomoćna enumeracija naziva `TileType`, koja može poprimiti vrijednosti poput `main` ili `empty` koji govore algoritmu što se nalazi na toj poziciji. Također je kreirana i pomoćna klasa `PathElement`, koja predstavlja svaki potencijalni element unutar putanje.

```
public class PathElement {
 public TileType type;
 public Vector2 position;
 public List<Vector2> nearPathElements; }
```

Kao što je vidljivo iz prethodnog koda, svaki element putanje također sadrži i listu svih susjednih elemenata koji se nalaze direktno uz njega. Ovime se znatno olakšava kasnije generiranje sporednih grana razine, te stvaranja granica.

```
public List<Vector2> getNextPath(float min, float max,
Dictionary<Vector2, TileType> currentTiles) {
 List<Vector2> pathTiles = new List<Vector2>();
 if (position.y + pixelMultiplier < max &&
!currentTiles.ContainsKey(new Vector2(position.x, position.y +
pixelMultiplier))) {
  pathTiles.Add(new Vector2(position.x, position.y +
pixelMultiplier)); }
 if (position.y - pixelMultiplier > min &&
!currentTiles.ContainsKey(new Vector2(position.x, position.y -
pixelMultiplier))) {
  pathTiles.Add(new Vector2(position.x, position.y -
pixelMultiplier)); }
return pathTiles; }
```

Kôd 6.7 Programski kod za dohvaćanje susjednih vertikalnih elemenata

Metoda `getNextPath` (Kôd 6.7) popunjava listu susjednih elemenata na `y` osi za svaki element putanje. Varijable `min` i `max` označavaju vanjske dimenzije čitave razine, te se koriste samo kao provjera koja osigurava da algoritam ne izlazi iz tih dimenzija. Za svaki element se vrši provjera njegove trenutne pozicije uvećane ili umanjene za jedan „korak“ po vertikalnoj osi (`pixelMultiplier` varijabla), te ako ta pozicija već ne postoji, kreira se i dodaje u kolekciju. Na identičan način se radi i provjera pozicija na horizontalnoj osi.

Idući korak uključuje generiranje glavne putanje, od ulaza do izlaza iz razine, a programski kod algoritma se može vidjeti u nastavku (Kôd 6.8).

```
private void CreateMainPath(){
 int randomY = Random.Range(0, maxBound + 1);
 PathElement mainPath = new PathElement(TileType.main, new
Vector2(0, randomY * pixelMultiplier), minBound, maxBound,
pathPositions);
startPos = mainPath.position;
int boundTracker = 0;
while (boundTracker < maxBound) {
 pathPositions.Add(mainPath.position, TileType.empty);
 int adjacentTileCount = mainPath.nearPathElements.Count;
 int randomIndex = Random.Range(0, adjacentTileCount);
 Vector2 nextMainPathPos;
 if (adjacentTileCount > 0) {
  nextMainPathPos = mainPath.nearPathElements[randomIndex]; }
 else {
  break; }
 PathElement nextMainPath = new PathElement(TileType.main,
nextMainPathPos, minBound, maxBound, pathPositions);
 if (nextMainPath.position.x > mainPath.position.x ||
(nextMainPath.position.x == (maxBound * pixelMultiplier) -
pixelMultiplier && Random.Range(0, 2) == 1)) {
 ++boundTracker;}
mainPath = nextMainPath; }}
```

Kôd 6.8 Kod za generiranje glavne putanje u Pathfinder algoritmu

Na početku metode se slučajnim odabirom definira `randomY` varijabla, koja predstavlja početnu točku uz lijevu granicu razine. Zatim se definira pomoćni brojač `boundTracker`, koji unutar `while` petlje osigurava da se algoritam izvršava sve dok se ne dođe do krajnje desne koordinate razine. Unutar petlje prvo se dodaje trenutna pozicija, odnosno objekt `PathElement` klase, u kolekciju svih elemenata, te se dohvaća ukupni broj njegovih susjednih elemenata. Prilikom prvog izvođenja algoritma, trenutna pozicija će biti jednaka početnoj poziciji, budući da generiranje putanje kreće iz ishodišta. Nakon toga se slučajnim odabirom uzima jedan od susjednih elemenata trenutne pozicije, te na se na njegovoj lokaciji kreira idući element putanje (varijabla `nextMainPath`). Ako je `x` koordinata idućeg elementa putanje veća od `x` koordinate trenutnog elementa, to znači da se algoritam pomaknuo u desno, te je potrebno inkrementirati vrijednost `boundTracker` brojača za 1. Tako na kraju petlje idući element putanje postaje trenutni element, te se u idućem koraku petlje opet pronalaze njegovi susjedni elementi, i tako sve dok se ne dođe do granice razine.

Idući korak algoritma uključuje generiranje sporednih putanja na prethodno generiranoj glavnoj putanji. Ta funkcionalnost se može vidjeti na sljedećem isječku programskog koda (Kôd 6.9).

```
private void CreateRandomPath() {
        List<PathElement> pathQueue = new List<PathElement>();
        foreach (KeyValuePair<Vector2, TileType> tile in
pathPositions) {
            Vector2 tilePos = new Vector2(tile.Key.x,
tile.Key.y);
            pathQueue.Add(new PathElement(TileType.random,
tilePos, minBound, maxBound, pathPositions)); }
pathQueue.ForEach(delegate (PathElement tile) {
            int adjacentTileCount = tile.nearPathElements.Count;
            if (adjacentTileCount != 0) {
                if (Random.Range(0, 4) == 1) {
                    CreateRandomChamber(tile); }}}); }
```

Kôd 6.9 Dio programskog koda za generiranje sporednog puta u Pathfinder algoritmu

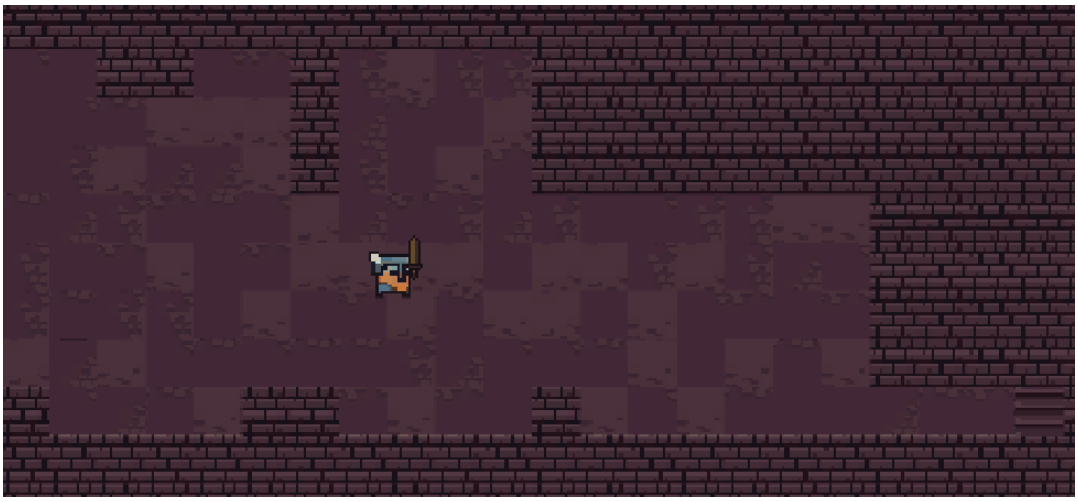
Pomoćna se lista `pathQueue` popunjava svim elementima glavne putanje, s razlikom što je ovaj put svaki element definiran kao `random` tip putanje. Nakon toga se prolazi po svim elementima

pomoćne liste, te se za svaki element definira 25% vjerojatnosti da će se pozvati metoda `CreateRandomChamber`. Ta metoda prima bilo koji objekt klase `PathElement`, te počevši od njega, generira elemente podloge u određenom vidnom polju oko tog početnog elementa, na vrlo sličan način već opisan u Infinity algoritmu.

Zadnji korak algoritma se sastoji od pozicioniranja podloge na scenu, na mjesto svih prethodno generiranih elemenata. Budući da je ključ svakog elementa putanje njegova pozicija, smještanje na scenu je vrlo jednostavno.

```
foreach (KeyValuePair<Vector2, TileType> tile in pathPositions) {
            toInstantiate = floorTiles[Random.Range(0,
floorTiles.Length)];
            instance = Instantiate(toInstantiate, new
Vector3(tile.Key.x, tile.Key.y, 0f), Quaternion.identity) as
GameObject; }
```

Pozicioniranje granica radi se na vrlo sličan način. Algoritam prolazi po svim elementima putanje, i provjerava njihove susjedne elemente. Ako su oni nedefinirani, to znači da na tim pozicijama nije ništa generirano, te je tu potrebno kreirati zid (ili neku drugu neprohodnu prepreku), kako bi se onemogućilo da igrač izađe izvan dimenzija same razine. Ako je potrebno generirati ostale elemente igre, poput kovčega sa zlatom, moguće je po potrebi ponovno proći po svim elementima putanje, i slučajnim odabirom ih razmjestiti na njoj. Kao što se može vidjeti na idućoj slici (Slika 6.4), razine stvorene pomoću ovog algoritma sastoje se od mnogo otvorenih površina oko svoje sredine koje su nastale proširivanjem osnovne putanje, te stvaraju prirodan dojam toka igre s lijeva na desno, prema izlazu.



Slika 6.4 Dio razine kreiran pomoću Pathfinder algoritma

6.4. Tilemap algoritam

Tilemap algoritam je baziran na principima algoritama staničnih automata (engl. *cellular automata*), koji služe za modeliranje odnosno prikaz ponašanja i razvoja različitih sustava kroz vrijeme. Ti algoritmi definiraju svoj svijet koji se sastoji od niza dvodimenzionalnih ćelija (stanica) smještenih u određeni broj redaka i stupaca, a svaka od tih osnovnih stanica može biti u nekom unaprijed određenom stanju. Tijekom izvođenja algoritam upravlja stanjima svih stanica, te ih ovisno o nekim pravilima, uvjetima ili matematičkim funkcijama prebacuje iz jednog stanja u drugo. Te radnje je moguće ponavljati više puta, pri čemu proceduralna logika algoritma definira svojevrsnu evoluciju stanica kroz vrijeme.

Jedan od najpoznatijih algoritama ovog tipa je Game of Life algoritam razvijen 1970. godine, kod kojeg svaka stanica u sustavu može biti određena kao „živa“ ili „mrtva“. Algoritam zatim prolazi po svim stanicama, te ovisno o stanju stanice i njezinih susjeda određuje koje će žive stanice umrijeti (npr. žive stanice koje nemaju niti jednog živog susjeda) ili pak koje će mrtve stanice oživjeti (npr. stanice okružene živim susjedima). Budući da ovi algoritmi rade s elementima unutar 2D prostora, lako ih je iskoristiti prilikom izrade računalnih igara. Tilemap algoritam je dobio naziv po *Tilemap* funkcionalnosti Unity alata, koja služi za brzu izradu dvodimenzionalnih nivoa. Tilemap podržava definiranje palete koja se sastoji od niza elemenata korištenih prilikom izrade nivoa, te korištenje tih elemenata za „crtanje“ po pozadini koja se sastoji od niza ćelija. Iako je

prvenstvena namjena *Tilemap*-a ubrzavanje ručne izrade sadržaja, moguće je osnovnim funkcionalnostima upravljati i proceduralno preko koda, što će se vidjeti u nastavku.

Sav programski kod Tilemap algoritma nalazi se unutar `TileGenerator` skripte, pridružene *prefab* elementu istog naziva. Za definiranje stanja svih elemenata razine, tijekom izvođenja algoritma korišteno je dvodimenzionalno polje naziva `mainMap`.

```
private int[,] mainMap;
```

Prvi korak algoritma sastoji se od inicijalnog popunjavanja ovog polja s vrijednostima. Slično kao i kod prethodno opisanog Game of Life algoritma, svaki element polja može poprimiti dva stanja – aktivno i pasivno, odnosno 1 i 0. Aktivno stanje (vrijednost 1) označava da na njegovom mjestu treba generirati prepreku, dok pasivno stanje (vrijednost 0) označava da će na njegovom mjestu biti podloga razine. Inicijalno elementi polja nemaju određeno stanje, te se stoga slučajnim odabirom prilikom prvog popunjavanja postavlja prvo stanje elemenata.

```
for (int x = 0; x < width; x++){
 for (int y = 0; y < height; y++){
  mainMap[x, y] = Random.Range(1, 101) < startChance ? 1 : 0;}}
```

Varijabla `startChance` definira inicijalnu šansu da će svaki element poprimiti vrijednosti 1 ili 0, a prilikom prvog pokretanja postavljena je na 50, što znači da će u prosjeku na početku pola elemenata biti aktivno. U idućem se koraku kreira metoda koja kao parametar prima prethodno definirano polje, prolazi kroz sve njegove elemente, te postavlja novo stanje svakom elementu. Kako bi se odredilo da li neki element treba postati aktivan ili pasivan, prvo je potrebno dohvatiti broj aktivnih susjednih elemenata. Svaki element ima maksimalno osam susjednih elemenata – lijevi, desni, gornji i donji susjedni element, te četiri dijagonalna. Idući dio programskog koda (Kôd 6.10) prikazuje upravljanje susjednim elementima u sklopu algoritma.

```
int neighbor;
BoundsInt bounds = new BoundsInt(-1, -1, 0, 3, 3, 1);
 for (int x = 0; x < width; x++){
  for (int y = 0; y < height; y++){
   neighbor = 0;
   foreach (var bound in bounds.allPositionsWithin) {
```

```
    if (bound.x == 0 && bound.y == 0) {
     continue; }
    if (x+bound.x >= 0 && x+bound.x < width && y+bound.y >= 0 &&
y+bound.y < height) {
    neighbor += map[x + bound.x, y + bound.y]; }
    else{
     neighbor++;}}}}
```

Kôd 6.10 Postavljanje susjednih elemenata u Tilemap algoritmu

Za lakše pronalaženje susjednih elemenata definirana je Varijabla `bounds`, kao objekt klase `BoundsInt`, koja se u sklopu Unity API-a koristi za brzo definiranje pravokutnika ili kvadra koji sadrži sve elemente oko nekih koordinata. Prva tri parametra prilikom inicijaliziranja varijable označavaju početne x, y i z koordinate pravokutnika (u ovom slučaju -1, -1 i 0), dok zadnja tri parametra označavaju veličinu odnosno duljinu stranica pravokutnika (u ovom slučaju 3). Ovime je zapravo definiran objekt dimenzija 3x3, kojem središnji element ima koordinate 0 na x i y osi, te sadrži osam susjednih elemenata. Unutar for petlji prolazi se po svim elementima polja, a za svaki element se zatim određuje broj njegovih susjeda. Kod unutar if uvjeta osigurava da se tijekom zbrajanja preskoče elementi koji su već u sredini (koordinate 0), te da se ne zbrajaju susjedni elementi koji bi bili izvan granica širine i visine čitave razine. Pošto pasivni elementi imaju vrijednost 0, njihovim pribrajanjem se ne mijenja stanje brojača, dok svaki pronađeni aktivni susjed inkrementira brojač za 1. Elementi uz sam rub razine uvijek se tretiraju kao aktivni, kako bi se automatski kreirale prepreke uz sam rub, koje sprečavaju igrača da napusti razinu.

Sljedeći korak algoritma prati trenutni status svakog elementa i prethodno izračunat broj aktivnih susjeda, te im po potrebi mijenja stanje.

```
if (map[x, y] == 1) {
 if (neighbor < passiveLimit) {
  map[x, y] = 0; }
 else {
  map[x, y] = 1; }}
if (map[x, y] == 0) {
 if (neighbor > activeLimit) {
  map[x, y] = 1; }
 else {
```

```
map[x, y] = 0; }}
```

Kôd 6.11 Postavljanje statusa elemenata mape u Tilemap algoritmu

Kao što se vidi iz prethodnog dijela koda (Kôd 6.11), ako je trenutni element aktivan ali je broj aktivnih susjednih elemenata manji od neke određene granice pasivnih elemenata (varijabla `passiveLimit`), i taj element postaje pasivan. Isto tako, ako je neki element pasivan,ali je broj aktivnih susjeda veći od željene granice (varijabla `activeLimit`), i on postaje aktivan.

Zadnji korak algoritma uključuje korištenje prethodno definiranih elemenata i njihovih statusa, te generiranje razine u igri korištenjem *Tilemap* komponenti. Svaki element može imati dva stanja, te je stoga najjednostavnije definirati i dvije TileMap komponente koje odgovaraju tim stanjima, te pomoćne varijable koje predstavljaju pojedinačne ćelije unutar mape.

```
public Tilemap activeTilemap;

public Tilemap passiveTilemap;

public Tile activeTile;

public Tile passiveTile;
```

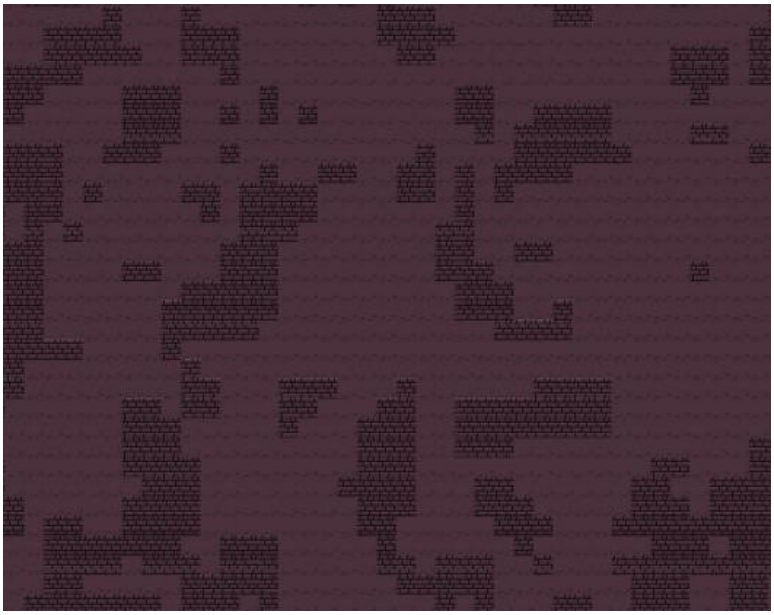
Nakon ovoga je dovoljno proći po svim elementima polja, te ako je element aktivan, na njegovim koordinatama kreirati aktivnu ćelija, a u suprotnom se kreira pasivna. Izgled i ponašanje aktivnih i pasivnih ćelija se definira direktno unutar Unity alata, te ako aktivne ćelije trebaju predstavljati prepreke (kao što je slučaj u ovom algoritmu), dovoljno je pridružiti te elemente `activeTile` varijabli.

```
if (mainMap[x,y] == 1) {

 activeTilemap.SetTile(new Vector3Int(-x + width / 2, -y + height /2, 0),
activeTile); }
```

Za postavljanje ćelije na *Tilemap* komponentu, dovoljno je iskoristiti `SetTile` metodu koja prima koordinate i vrstu ćelije koja se kreira. Izgled razina generiranih ovim algoritmom jako ovisi o definiranim parametrima granica aktivnih i pasivnih elemenata, broju inicijalnih aktivnih elemenata, te koliko puta je algoritam ponavljao postupak određivanja aktivnih elemenata. Na slici (Slika 6.5) se može vidjeti primjer dijela razine generirane nakon jednog ciklusa, s inicijalnom

šansom za kreiranje aktivnih elemenata 25%, te granica za aktivne i pasivne elemente postavljene na 2.



Slika 6.5 Primjer slučajnog rasporeda aktivnih i pasivnih elemenata u Tilemap algoritmu

6.5. Rogue algoritam

Glavna inspiracija za izradu ovog algoritma je izgled starih računalnih igara s proceduralno generiranim sadržajem, već opisanim u drugom poglavlju ovog rada. Naziv algoritma je uzet prema glavnom nazivu podžanra tih igara – *Roguelike*. Većina ranih igara tog tipa odvijala se kroz razine sastavljene od pravokutnih prostorija različitih dimenzija, koje su međusobno bile povezane uskim hodnicima, te će stoga i cilj ovog algoritma biti generiranje vrlo sličnog izgleda razina. Algoritam generira praznu pravokutnu prostoriju (sobu) određenih dimenzija, a nakon toga uzima jednu od četiri strane sobe kao točku izlaza. Iz te se točke izlaza generira hodnik određene duljine, a završna točka hodnika ujedno predstavlja i točku od koje će krenuti kreiranje nove sobe. Ovi koraci se ponavljaju određeni broj puta, a konačni rezultat je zapravo mreža soba različitih dimenzija, međusobno povezanih hodnicima. Glavni dio programskog koda algoritma je smješten unutar `RogueGenerator` skripte. Kreirane su i posebne pomoćne klase naziva `Room` i

`Corridor`, koje sadrže programsku logiku vezanu uz upravljanje sobama i hodnicima koji ih povezuju, uključujući x i y koordinate na sceni svake sobe i hodnika.

Za pohranu osnovnih informacija o svakom elementu razine koristi se dvodimenzionalno polje, čije dimenzije odgovaraju inicijalno zadanim dimenzijama čitave razine. U prvom koraku algoritma inicijalizira se veličina tog polja za obje dimenzije.

```
TilePick[][] mainTiles = new TilePick[columns][];
 for (int i = 0; i < mainTiles.Length; i++) {
  mainTiles[i] = new TilePick[rows]; }
```

Enumeracija `TilePick` može poprimiti vrijednosti poput `Wall` ili `Floor`, te služi za lakše definiranje vrste elemenata. U idućem se koraku izvršavaju metode za kreiranje soba i hodnika. Za preglednije strukturiranje koda i lakše upravljanje različitim elementima, definirana su dva polja, koja sadrže objekte pomoćnih klasa `Room` i `Corridor`, te pomoćne varijable koje određuju njihove dimenzije, odnosno količinu soba koju je potrebno kreirati.

```
rooms = new Room[numberOfRooms.Random];
corridors = new Corridor[rooms.Length - 1];
```

Na početku algoritma prvo se kreiraju nulti elementi svake kolekcije (prva soba i prvi hodnik). Unutar pomoćne klase `Room` smještena je programska logika kreiranja prve sobe, a ona uključuje slučajni odabir vrijednosti širine i visine sobe, dok se njezine početne koordinate smještaju na sredinu razine.

```
roomWidth = widthRange.Random;
roomHeight = heightRange.Random;
xPosition = Mathf.RoundToInt(columns / 2f - roomWidth / 2f);
yPosition = Mathf.RoundToInt(rows / 2f - roomHeight / 2f);
```

Varijable `xPosition` i `yPosition` označavaju koordinate donjeg lijevog kuta sobe, a na temelju je dimenzija sobe onda moguće odrediti i sve ostale koordinate unutar njezinih granica. Na temelju početnih pozicija i duljine hodnika također je moguće izračunati i završne koordinate odgovarajućeg hodnika. Kreiranje hodnika je definirano unutar `Corridor` klase, a dio metode za kreiranje se može vidjeti u nastavku (Kôd 6.12).

```
public void CreateCorridor(Room room, RandomNumberGenerator
length, RandomNumberGenerator roomWidth, RandomNumberGenerator
roomHeight, int columns, int rows, bool isFirstCorridor) {
direction = (Direction)Random.Range(0, 4);
corridorLength = length.Random;
switch (direction) {
  case Direction.Up:
   startXPosition = Random.Range(room.xPosition, room.xPosition
+ room.roomWidth - 1);
   startYPosition = room.yPosition + room.roomHeight;
   break;}
```

Kôd 6.12 Dio programskom koda za generiranje hodnika unutar Rogue algoritma

Na početku metode se odabire jedan od četiri moguća smjera iz pomoćne `Direction` enumeracije, koja određuju u kojem će pravcu krenuti hodnik. Parametri duljine hodnika određeni su slučajnim odabirom, pomoću slučajno generiranih brojeva. Unutar `switch-case` provjera se definiraju početne koordinate hodnika ovisno o smjeru kretanja. Ako je odabran smjer prema gore, znači da početna x koordinata hodnika može biti smještena bilo gdje uz gornji rub sobe, dok je y koordinata uvijek fiksna, te odgovara postojećoj y koordinati sobe uvećanoj za njezinu visinu (jer koordinate sobe definiraju donji lijevi kut sobe). Na identičan način algoritam vrši provjeru i definiciju koordinata za ostala tri smjera.

Nakon kreiranja prvih elemenata, izvršava se kod i za kreiranje preostalih soba i hodnika. Ostali hodnici se kreiraju na identičan način već prikazan u ovom poglavlju (Kôd 6.12), jedino se na početku vrši dodatna provjera smjera kako bi se osiguralo da novi hodnik ne ide u istom smjeru kao prethodni. Ostale sobe se generiraju drugačijom metodom od početne sobe, a dio tog koda se može vidjeti i u nastavku (Kôd 6.13).

```
public void CreateRoomWithCorridor(RandomNumberGenerator
widthRange, RandomNumberGenerator heightRange, int columns, int
rows, Corridor corridor) {
 inputCorridor = corridor.direction;
 roomWidth = widthRange.Random;
 roomHeight = heightRange.Random;
 switch (corridor.direction) {
```

```
 case Direction.Up:
 roomHeight = Mathf.Clamp(roomHeight, 1, rows -
corridor.EndPositionY);
 yPosition = corridor.EndPositionY;
 xPosition = Random.Range(corridor.EndPositionX - roomWidth + 1,
corridor.EndPositionX);
 xPosition = Mathf.Clamp(xPosition, 0, columns - roomWidth);
 break; }}
```

Kôd 6.13 Dio programskog koda za određivanje koordinata sobe u Rogue algoritmu

Jedan od ulaznih parametara ove metode je i prethodno stvoreni hodnik. Ako je taj hodnik išao prema gore, početna y koordinata nove sobe mora biti jednaka završnoj y koordinati hodnika, dok je x koordinata nove sobe određena slučajnim odabirom između završne x koordinate hodnika, i te iste vrijednosti umanjene za širinu nove sobe. Na identičan način se radi provjera i određivanje koordinata za preostala tri smjera.

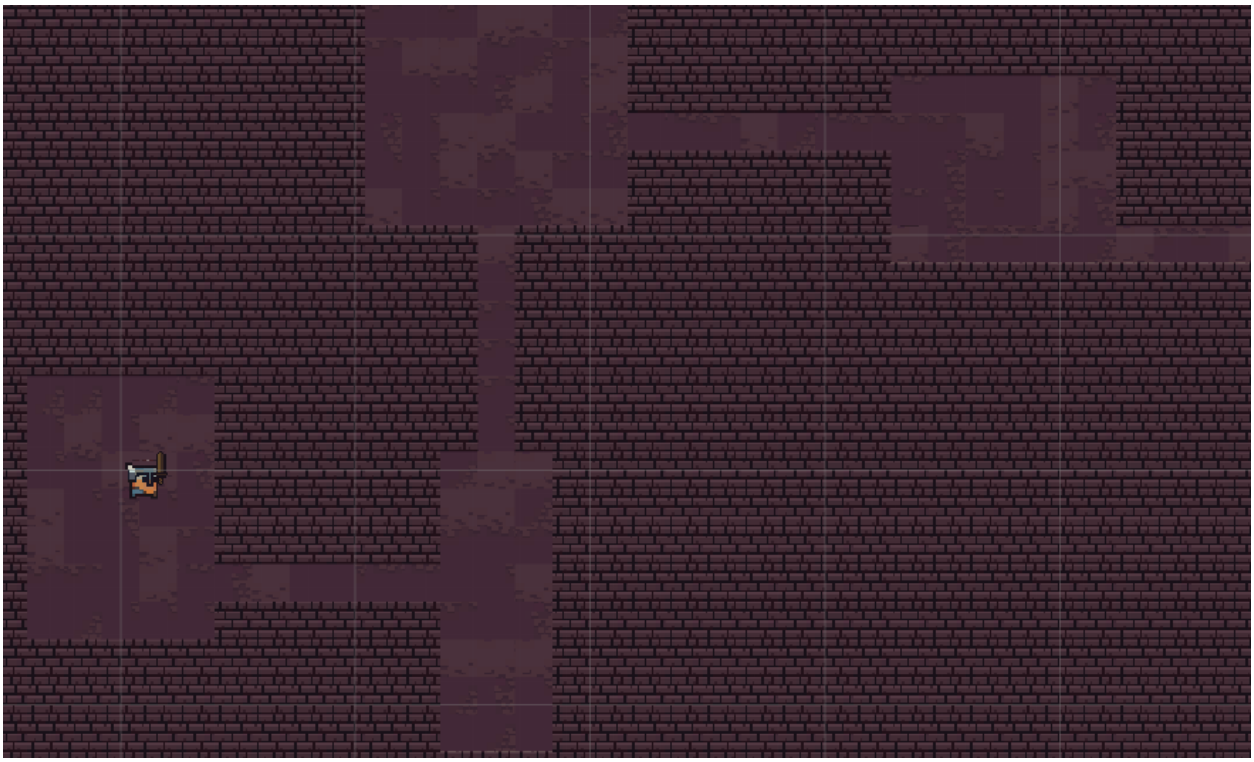
Prethodno opisani koraci algoritma definiraju koordinate svakog elementa na sceni, te ih spremaju unutar polja koja sadržavaju hodnike i sobe. Idući korak algoritma prolazi po svim elementima tih polja, te im postavlja tip željenog elementa kojeg treba kreirati na toj poziciji, što se može vidjeti na idućem dijelu programskog koda (Kôd 6.14).

```
for (int i = 0; i < rooms.Length; i++){
 Room currentRoom = rooms[i];
 for (int j = 0; j < currentRoom.roomWidth; j++){
  int xCoordinate = currentRoom.xPosition + j;
  for (int k = 0; k < currentRoom.roomHeight; k++){
   int yCoordinate = currentRoom.yPosition + k;
   mainTiles[xCoord][yCoord] = TilePick.Floor; }}}
```

Kôd 6.14 Kod za definiranje vrste elementa kojeg je potrebno kreirati unutar sobe

Varijable `xCoordinate` i `yCoordinate` su koordinate jedne od „ćelija“ unutar sobe, te se stoga na tim istim koordinatama unutar `mainTiles` polja postavlja `TilePick.Floor` kao tip elementa kojeg je potrebno kreirati, budući da sobe moraju biti prohodne i omogućiti kretanje igrača. Na isti način se i na koordinatama svakog hodnika definiraju elementi podloge. Na kraju ovog koraka, 2D polje `mainTiles` sadrži sve koordinate razine, te vrstu elementa kojeg je

potrebno kreirati na toj koordinati. Stoga je dovoljno proći po svim elementima polja, te pozvati `Instantiate` metodu Unity API-a za dodavanje elemenata na scenu. Primjer razine generirane ovim algoritmom se može vidjeti na idućoj slici (Slika 6.6).



Slika 6.6 Povezani sustav soba i hodnika kreiran pomoću Rogue algoritma

7. Analiza algoritama

Analiza prethodno opisanih i implementiranih algoritama je izvršena u iduće četiri glavne kategorije:

- Kompleksnost
- Performanse
- Fleksibilnost
- Igrivost

U nastavku će poglavlja biti detaljnije opisana svaka kategorija, što ona označava u pogledu algoritama za proceduralno generiranje sadržaja u igrama, te načini uspoređivanja, rangiranja i ocjenjivanja algoritama u navedenim kategorijama.

7.1. Kompleksnost

Analiza kompleksnosti algoritama je fokusirana na vremensku kompleksnost, koja označava vrijeme potrebno da se neki algoritam u potpunosti izvrši. Najčešće se označava pomoću tzv. notacije veliko O (engl. *big O notation*), koja prikazuje vrijeme potrebno za izvođenje algoritma u najgorem mogućem scenariju. Primjerice, metoda koja prima polje brojeva i uvijek vraća prvi element polja ima kompleksnost jednaku O(1), što znači da će vrijeme izvršavanja uvijek biti konstantno bez obzira na veličinu ulaznih parametara. Ako se u toj istoj metodi koristi *for* petlja unutar koje se prolazi po svim elementima polja veličine *n*, tom algoritmu se povećava kompleksnost na O(n), jer će broj puta izvršavanja te petlje unutar algoritma u najgorem slučaju biti jednak broju elemenata u samom polju, odnosno linearno će se povećavati kako raste i veličina polja.

Board algoritam ima kompleksnost $O(n^2)$, jer prilikom postavljanja vanjske granice razine, te kasnije prilikom definiranja ostalih elemenata igre prolazi po svim redovima i stupcima od kojih se sastoji igraća „ploča“, što se radi pomoću vanjske i unutarnje petlje:

```
for (int x = 1; x < columns - 1; x++){
 for (int y = 1; y < rows - 1; y++){
```

```
Vector3 position = new Vector3(x * 0.16f, y * 0.16f, 0f);}
```

Kao što je vidljivo iz prethodnog dijela koda iz Board algoritma, kompleksnost svake petlje je O(n), ali budući da je jedna smještena unutar druga, kompleksnost čitavog algoritma postaje $O(n^2)$.

Infinity algoritam također ima kompleksnost $O(n^2)$. Glavni dio algoritma je zadužen za provjeru smjera kretanja glavnog lika, te ovisno o tome kreira sadržaj unutar vidnog polja igrača, te se dio tog koda može vidjeti u prethodnom dijelu rada (Kôd 6.5). Unutar if uvjeta se pomoću dvije petlje (jedne unutar druge) prolazi po svim koordinatama vidnog polja glavnog lika, te dodaju elementi igre na te lokacije.

Kompleksnost Pathfinder algoritma je $O(n^3)$. Najkompleksniji dio algoritma se odnosi na provjeru glavne putanje, te generiranje sporednih puteva. Nakon što algoritam u prvom koraku kreira osnovnu putanju s lijeva na desno, ponovno se prolazi po svim kreiranim elementima. Zatim se ovisno o slučajno odabranom broju poziva metoda za kreiranje sporednih grana razine, nazvane `CreateRandomChamber`, što se može vidjeti iz prethodno opisanog dijela koda (Kôd 6.9). U toj metodi se unutar dvije petlje prolazi po dimenzijama širine i visine nove grane, te na te lokacije smještaju potrebni elementi igre. Budući da se poziv te metode čija je kompleksnost $O(n^2)$ već odvija unutar petlje (kompleksnost O(n)), u najgorem slučaju vrijeme potrebno za izvršavanje algoritma raste na $O(n^3)$.

Kompleksnost Tilemap algoritma je također $O(n^3)$. Vremenski najzahtjevniji dio algoritma uključuje postavljanje aktivnih elemenata u pasivne (i obrnuto), a poziv te metode se odvija za svaku iteraciju željenog broja ponavljanja algoritma:

```
for (int i = 0; i < numR; i++){
 terrainMap = genTilePos(terrainMap); }
```

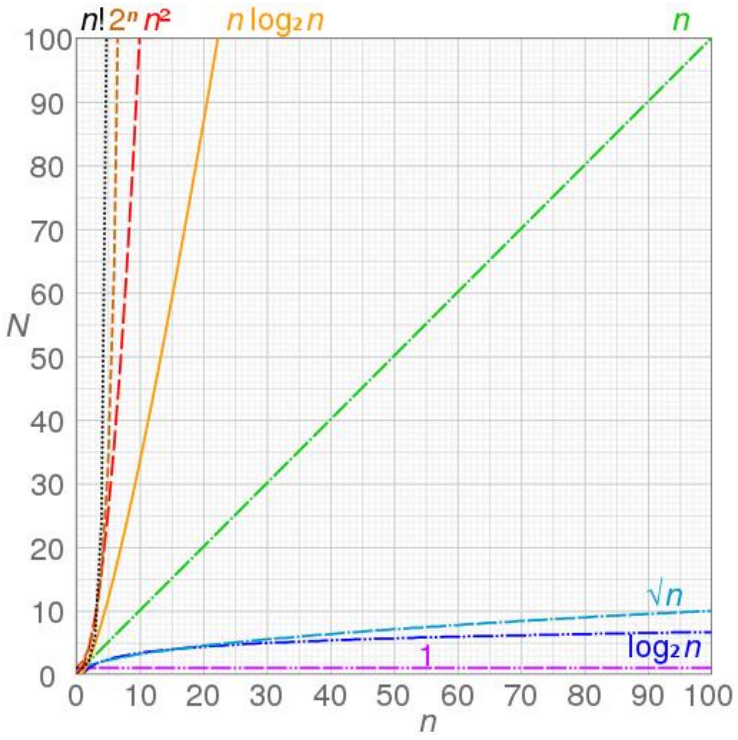
Programski kod glavnog dijela `genTilePos` metode se može vidjeti u već prethodno navedenom kodu (Kôd 6.11) prilikom opisa Tilemap algoritma. U tom dijelu algoritam prolazi po svim dimenzijama razine, računa broj aktivnih susjeda svakog elementa, te ovisno o tom broju određuje njihov status. Taj dio koda je realiziran pomoću dvije petlje, što znači da konačni rezultat analize kompleksnosti uključuje tri petlje (smještene jedna unutar druge), te je kompleksnost algoritma $O(n^3)$. Važno je istaknuti da iako se unutar tih petlji radi prolaz po svim susjednim elementima, to ne povećava kompleksnost algoritma na $O(n^4)$.

```
foreach (var b in myB.allPositionsWithin) {
 …}
```

Varijabla `myB` je uvijek konstantne veličine, te ne ovisi o bilo kakvim ulaznim parametrima, što znači da je kompleksnost samo ovog dijela koda zapravo O(1), a time kompleksnost ukupnog algoritma sastavljenog od te četiri petlje je jednaka O(n) * O(n) * O(n) * O(1), što je jednako $O(n^3)$.

Rogue algoritam ima kompleksnost $O(n^3)$. Dio algoritma s najvećim brojem ponavljanja se izvodi prilikom postavljanja elemenata igre unutar dimenzija sobe. To uključuje prolazak po kolekciji svih soba, te je zatim za svaku sobu potrebno proći po njezinoj duljini i širini kako bi se dobile sve moguće koordinate pozicija unutar nje, te smjestili željeni elementi igre na te koordinate. Taj dio algoritma je prikazan u prethodnom poglavlju (Kôd 6.14). Budući da se programski kod sastoji od 3 petlje smještene jedna unutar druge, vremenska kompleksnost je jednaka $O(n^3)$.

Iz ovog se može zaključiti da svi korišteni algoritmi relativno vremenski zahtjevni, što se može vidjeti i iz idućeg grafa (Slika 7.1) koji prikazuje generalni rast vremenske kompleksnosti ovisno o notaciji veliko O.



Slika 7.1 Utjecaj kompleksnosti na povećanje broja operacija N ovisno o broju ulaznih parametara n

Glavni razlog tomu je što su grafikoni razvijeni za sličnu svrhu – proceduralno generiranje sadržaja unutar 2D igre. Najkompleksniji dijelovi svakog korištenog algoritma su upravo oni dijelovi gdje se radi smještanje elemenata igre na lokacije unutar scene, odnosno razine igre. Korištenjem konvencionalnih metoda programiranja i Unity API-a, za svaki element je potrebno definirati njegove koordinate na x i y osi unutar razine, što se ostvaruje pomoću više petlji. To znači da će vremenska kompleksnost čak i najjednostavnijih algoritama uvijek biti barem $O(n^2)$. Često je i kod kompliciranijih algoritama potrebno iterativno prolaziti i po nekim drugim pomoćnim kolekcijama kako bi pronašli ispravne elemente, te s time kompleksnost automatski rase na $O(n^3)$.

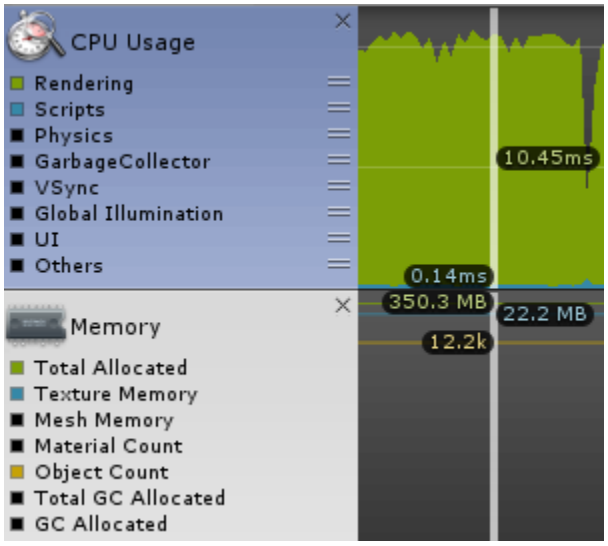
7.2. Performanse

Svaka igra je napravljena na drugačiji način, koristeći različite principe, te je teško ili gotovo nemoguće unaprijed znati kako i koliko resursa računala će igra koristiti. Zbog toga je bitno mjeriti i pratiti resurse i njihovo zauzeće tijekom izvršavanja igre, čime se stječe bolji uvid u performanse, i lakše identificiraju područja koja bi trebalo dodatno optimizirati. U tu svrhu Unity nudi alat naziva *Profiler*, koji se automatski pokreće zajedno s igrom, te u pozadini vrši detaljna mjerenja poput iskorištavanja radne memorije, trošenja procesorskog vremena ili vremena potrebnog za iscrtavanje svih elemenata na sceni (Smith et al., 2015).

U sklopu ovih algoritama odabrana su tri relevantna područja mjerenja: opterećenje procesora tijekom učitavanja proceduralno generirane razine, opterećenje procesora tijekom igranja te razine, te memorijsko opterećenje nakon učitavanja razine. U svrhu mjerenja sve će generirane razine biti veličine 50x50 (maksimalni broj kreiranih elemenata igre je 250), budući da to predstavlja približno stvarnu veličinu razina koje se koriste u sličnim igrama ovog tipa. Sve mjerenja su vršena na računalu s Intel Core i7-6700HQ procesorom s 8 jezgri, radnog takta 2.60 GHz, NVIDIA GeForce GTX 1060 grafičkom karticom, te 16 GB RAM-a.

Opterećenje procesora i memorije u sklopu Unity *Profiler* alata mjeri mnogo različitih aspekata igre, od kojih mnogi nisu relevantni prilikom proceduralnog generiranja 2D igara, te je stoga mjerenje fokusirano na četiri potkategorije – vrijeme generiranja i učitavanja razine, procesorsko vrijeme alocirano za iscrtavanje elemenata na sceni tijekom igranja, ukupno zauzeće memorije,

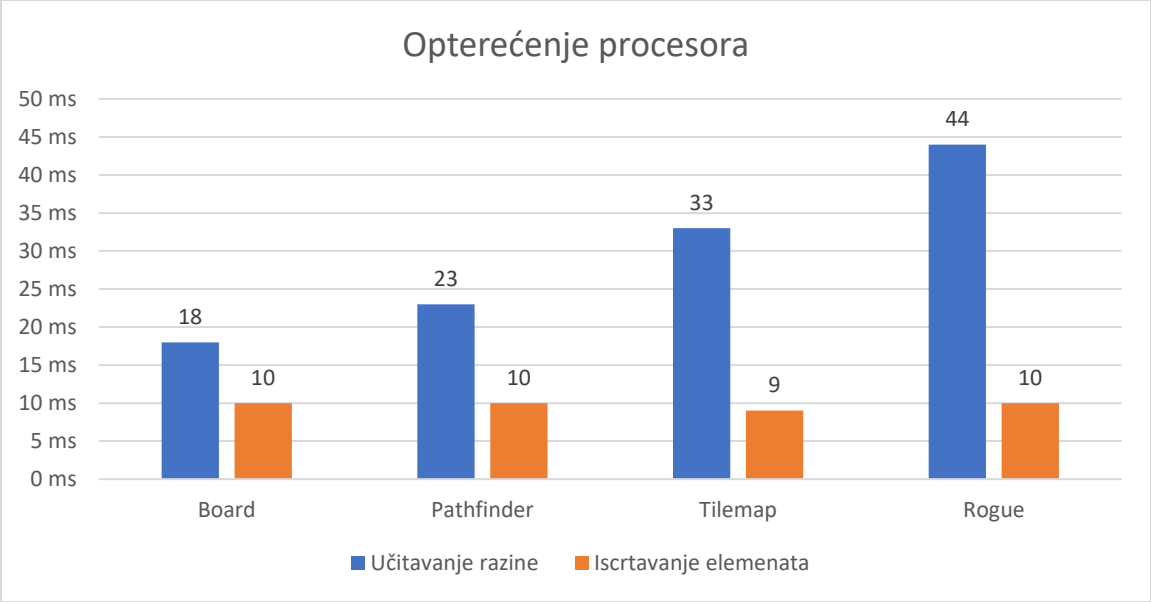
količina tekstura u memoriji, te ukupan broj objekta u memoriji. Iduća slika (Slika 7.2) prikazuje generalni izgled Profiler alata i rezultata prilikom mjerenja navedenih kategorija.



Slika 7.2 Prikaz rezultata mjerenja performansi unutar Profiler alata

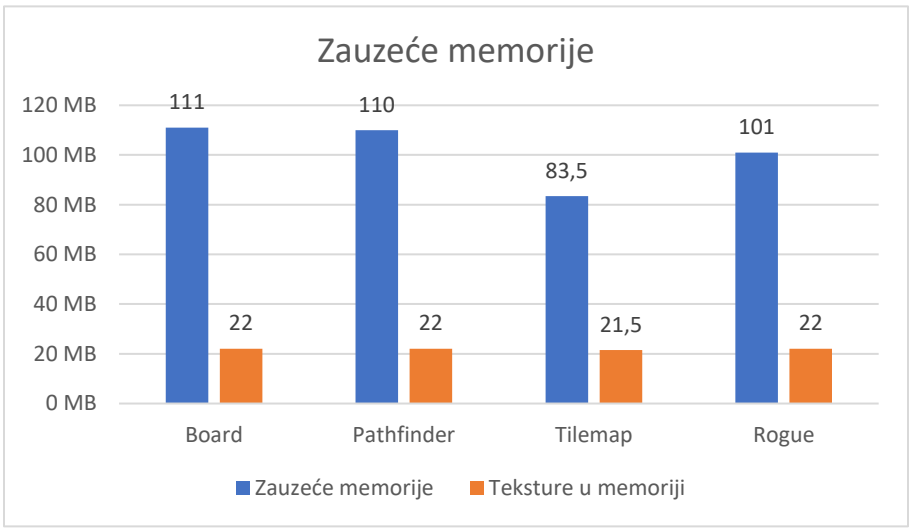
Sve mjerenja su ponovljena pet puta, kako bi se eliminirale potencijalne anomalije ili nepredviđeni „ekstremi“ u rezultatima, te je kao rezultat uzet prosjek svih mjerenja. Ako je vrijednost vrlo mala (npr. nekoliko milisekundi), zaokružena je na najbližu cjelobrojnu vrijednost. Infinity algoritam je izostavljen iz mjerenja jer funkcionira na znatno drugačiji način od ostalih algoritama, te se stoga njegovi rezultati ne mogu realno uspoređivati s druga četiri načina generiranja sadržaja. On je prilikom inicijalnog generiranja jako brz, budući da se skoro ništa ne generira u prvom koraku, a kasnije mu performanse znatno opadaju ovisno o tome koliko površine ekrana je igrač istražio. On je također namijenjen za „beskonačno“ generiranje, te se stoga ne može koristiti za definiranje razina fiksnih dimenzija prilikom mjerenja.

Na grafikonu (Slika 7.3**Pogreška! Izvor reference nije pronađen.**) se mogu vidjeti rezultati mjerenja brzine u milisekundama potrebne za kreiranje razine, te prosječno procesorsko vrijeme utrošeno za iscrtavanje elemenata tijekom igre.



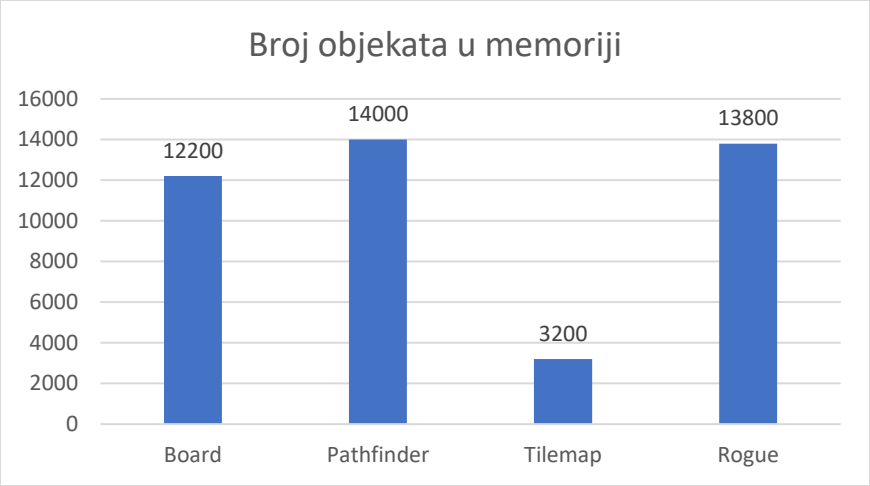
Slika 7.3 Rezultati mjerenja procesorskog vremena prilikom izvođenja algoritama

Na grafikonu (Slika 7.4) prikazani su rezultati mjerenja zauzeća memorije tijekom izvršavanja igre, te količina memorije alocirane za pohranu tekstura na razini.



Slika 7.4 Rezultati mjerenja zauzeća memorije prilikom proceduralnog generiranja

Grafikon (Slika 7.5) prikazuje količinu pozadinskih objekata, koje Unity procesuira tijekom izvršavanja igre, pohranjenih u radnoj memoriji.



Slika 7.5 Rezultati mjerenja količine objekata u memoriji tijekom izvođenja igre

Kao što se može vidjeti iz prikazanih rezultata, board algoritam je najbrži prilikom učitavanja razina, što je očekivano s obzirom na njegovu jednostavnost. Rogue algoritam je najsporiji, budući da u pozadini izvršava posebne kalkulacije za svaku prostoriju i svaki hodnik, kako bi se osigurao ispravan raspored i definirale točne koordinate na sceni. Kod iscrtavanja elemenata svi algoritmi su davali gotovo identične rezultate, iz čega se može zaključiti da nemaju značajnog utjecaja na taj dio procesorskog vremena.

Kod zauzeća memorije, te količine objekata u memoriji, *Tilemap* se pokazao najefikasnijim. To je zato što on koristi *Tilemap* komponentu unutar Unity-a za definiranje i smještanje elemenata igre, a ona je prvenstveno namijenjena i optimizirana za izradu velikih 2D razina. Ostali algoritmi u ovom području daju slične rezultate, s tim da je korištenje Board algoritma ipak stvorilo nešto manji broj elemenata u memoriji u usporedbi s Pathfinder i Rogue algoritmom. Teksture u memoriji u svim slučajevima zauzimaju gotovo identičnu količinu memorije, te se može zaključiti da izbor algoritama nema bitniji utjecan na taj dio opterećenja memorije.

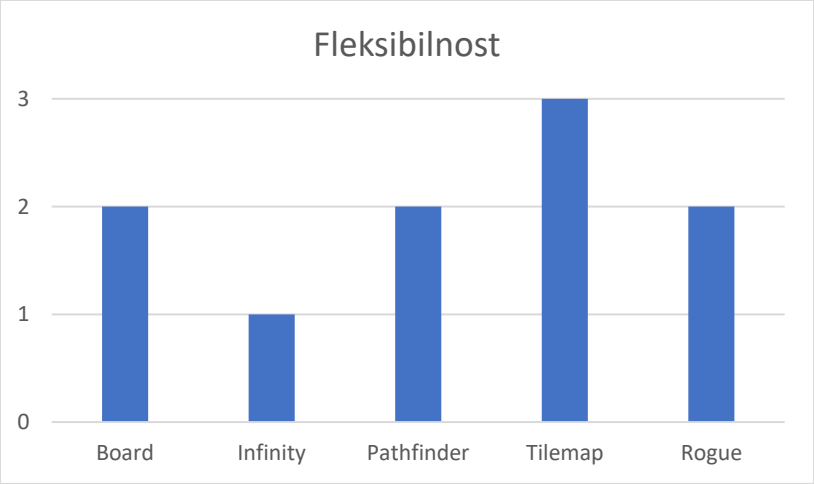
7.3. Fleksibilnost

Fleksibilnost predstavlja sposobnost algoritama i njihovih sastavnih komponenti da se prilagode novim zahtjevima ili potrebama. U sklopu algoritama korištenih u ovom radu, fleksibilnost opisuje da li je algoritam lako proširiti novim elementima, da li ga je moguće primijeniti za izradu igara

druge vrste, ili ga čak i prilagoditi za korištenje izvan originalnog nauma (primjerice za izradu 3D igara ili komponenti). Na fleksibilnost algoritma znatno utječe i međusobna povezanost i ovisnost komponenti. Ako su različiti dijelovi algoritma dobro funkcionalno razdvojeni, lako je unositi promjene u sam programski kod ili ga pak prilagoditi drugoj namjeni (Nystrom, 2014).

Ocjenjivanje se vrši ocjenama od 1 do 3, poredanih od najlošije prema najboljoj, prema sljedećim kriterijima:

- Ocjena 1 – Algoritam je nefleksibilan, dodavanje novih elemenata je otežano, vrlo je teško primjenjiv na igre ostale vrste, te ga nije moguće bez drastičnih promjena prenamijeniti za izradu 3D sadržaja
- Ocjena 2 – Algoritam je moguće primijeniti na ostale igre uz djelomičnu izmjenu funkcionalnosti i programskog koda, dodavanje novih elemenata nije teško, te ga je moguće primijeniti i prilikom izrade 3D elemenata
- Ocjena 3 – Algoritam je vrlo fleksibilan, može se prilagoditi različitim namjenama bez potrebe za vrlo velikim restrukturiranjem i promjenama u kodu, dodavanje novih elemenata u igru je vrlo jednostavno i zahtijeva tek par dodatnih linija koda



Slika 7.6 Ocjene fleksibilnosti algoritama

Na prethodnoj slici (Slika 7.6) se mogu vidjeti ocjene fleksibilnosti dodijeljene algoritmima. Infinity algoritam je ocijenjen s 1. Iako je dosta jednostavan, zbog čega se na prvi pogled može činiti fleksibilnim, njegova namjena je dosta usko vezana uz ovaj tip igara, te ga je vrlo teško primijeniti na drugačije vrste ili žanrove. Iako dodavanje novih elemenata nije nemoguće, nije ih

najjednostavnije uklopiti u princip funkcioniranja algoritma, budući da se sadržaj ne generira unaprijed. Dodavanje treće dimenzije u infinity algoritam bi također bilo vrlo kompliciranо zbog konstantnog generiranja novog sadržaja, što je još jedan faktor za nisku ocjenu fleksibilnosti.

Rogue algoritam je ocijenjen s 2. Njegova namjena je također usko vezana uz ovaj tip igara, te njegova primjena nije baš logična ili jednostavna prilikom izrade drukčije vrste igara, ali ga je stoga relativno jednostavno prilagoditi za izradu 3D igara. U tom se slučaju on može koristiti za izradu igara drukčije vrste, te je dosta prvih 3D igara s početka 90-ih godina koristilo princip dizajniranja razina sastavljenih od otvorenih prostorija povezanih s uskim hodnicima. Dodavanje novih elemenata unutar postojećih razina je vrlo jednostavno i brzo, ali adaptiranje algoritma za generiranje drugačijeg sadržaja (npr. sobe koje više nisu pravokutnog oblika) zahtijevalo bi dosta veliku promjenu logike, te u tom pogledu algoritam nije fleksibilan.

Pathfinder algoritam je također ocijenjen s 2. On se može iskoristiti za izradu sličnih igara, pogotovo platformskih 2D igara, budući da su one dizajnirane s ciljem igranja „s lijeva na desno“, što odgovara konceptu na koji ovaj algoritam generira sadržaj. Dodavanje novih elemenata je poprilično jednostavno i ne zahtijeva velike promjene u kodu. Teško ga je iskoristiti za izradu smislenog 3D sadržaja bez većih promjena, te je to glavni razlog zašto nije dobio ocjenu 3.

Board algoritmu je također dodijeljena ocjena 2, budući da ga je zbog svoje jednostavnosti vrlo lako primijeniti za izradu svih vrsta igara koje se mogu odvijati na predefiniranoj 2D ploči, ali je zato gotovo neupotrebljiv za druge stvari. Dodavanje novih elemenata nije kompliciranо, ali je sav budući sadržaj i dalje strogo određen dimenzijama i strukturom glavne „ploče“ za igru. Prilagođavanje za podršku treće dimenzije također nije kompliciranо.

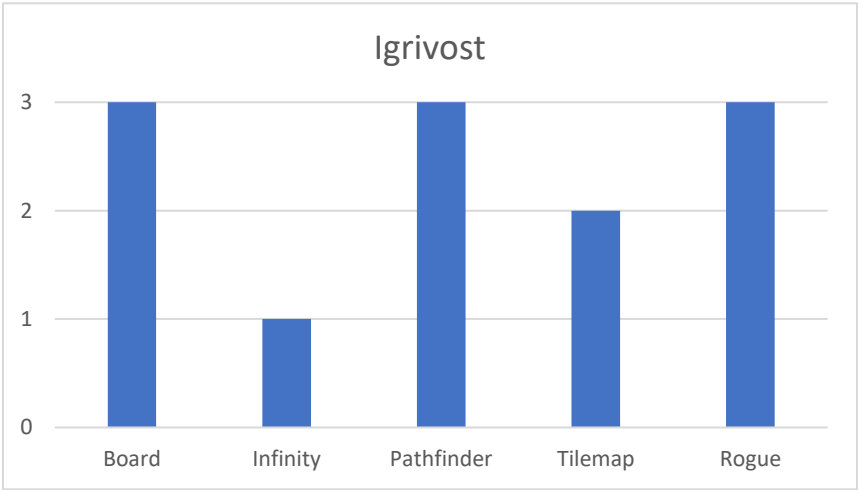
Tilemap je ocijenjen sa 3. Pošto on koristi *Tilemap* sustav unutar Unity-a, izuzetno je fleksibilan te ga je moguće adaptirati za kreiranje sadržaja za gotovo sve vrste 2D igara, te je generirani sadržaj moguće pretvoriti u 3D komponente koristeći neke od mogućnosti sadržanih unutar Unity-a. *Tilemap* komponente su i vrlo interaktivne, te je sve Tile elemente moguće vrlo lako dodavati ili uklanjati iz postojeće mape, čak i nakon što je razina već kreirana. Također je svaku razinu kreiranu pomoću ovog algoritma vrlo lako pohraniti kao gotovi *prefab* element, te je odmah iskoristiti za neku drugu namjenu, ili učitati unutar neke druge igre.

7.4. Igrivost

Igrivost podrazumijeva djelomično subjektivnu procjenu kvalitete same igre, odnosno sadržaja igre koji je proceduralno generiran pomoću algoritama. Osnovni faktori procjene igrivosti odnose se na sposobnost algoritma da generira razine koje odgovaraju ovoj vrsti 2D igara, mogućnosti da igrač ispuni ciljeve igre (dođe od početka do kraja razine), te osjećaju zabave i iznenađenja koji proizlazi iz činjenice da je svaka razina igre uvijek drugačija i unaprijed nepoznata.

Ocjenjivanje se vršiti ocjenama od 1 do 3, poredanih od najlošije prema najboljoj, prema sljedećim kriterijima:

- Ocjena 1 – Igra je djelomično ili potpuno neigriva, nije moguće izvršavanje osnovnih ciljeva igre, ili je znatno otežano njihovo ispunjavanje
- Ocjena 2 – Igra je djelomično igriva, moguće je ispunjavanje ciljeva igre, ali generirani sadržaj ne odgovara ovom tipu igre, ili je snalaženje u igri neintuitivno zbog čega igra nije zabava
- Ocjena 3 – Igra je potpuno igriva, generirani sadržaj odgovara vrsti igre, te je kretanje i navigiranje kroz njega vrlo intuitivno i zabavno



Slika 7.7 Ocjene igrivosti algoritama

Na prethodnoj slici (Slika 7.7) su prikazane ocjene igrivosti svih algoritama korištenih u sklopu rada. Board algoritam ima ocjenu 3, jer unatoč svojoj jednostavnosti generira vrlo pregledne i dobro strukturirane razine, s jasno definiranim preprekama i ostalim elementima igre. Snalaženje

u igri je vrlo jednostavno i intuitivno, te se igrač može brzo orijentirati unutar razine. Korištenje ovog algoritma bi ipak bilo puno prikladnije za izradu ovakvih igara za mobilne uređaje, prvenstveno zbog ograničenosti prostora i malih dimenzija ekrana koji zahtijevaju dobru preglednost unutar igre.

Infinity algoritam ima ocjenu 1, jer sama igrivost i preglednost igre znatno opada ako se sadržaj generira tijekom izvođenja igre, a ne unaprijed kao kod ostalih algoritama. Ovaj algoritam također nekad može generirati bitne elemente tako da budu blokirani iza prepreka, te na slične načine sprječavati igrača da dođe do cilja. Zbog konstantnog i naizgled beskonačnog generiranja sadržaja orijentiranje po razinama može biti vrlo teško i zbunjujuće, pogotovo ako je igrač već proveo dosta vremena istraživajući.

Ocjena 3 je dodijeljena Pathfinder algoritmu zato što je sadržaj koji je pomoću njega stvoren vrlo pregledan i logički raspoređen, te je naglašen prirodan tok razina s lijeve na desnu stranu. Prisutna je dovoljna količina varijacije da potakne igrača na istraživanje, ali su i dalje svi ciljevi igre lako dostupni, te je interakcija sa sadržajem vrlo intuitivna. Kreirani teren, te raspored svih prepreka izgledom podsjeća na špilje, što odgovara ovakvom tipu igara.

Tilemap algoritam je ocjenjen s 2. Sadržaj koji je generiran pomoću njega je dosta nasumičan, te jako ovisi o svim ulaznim parametrima.. Iako je sposoban generirati vrlo igrive i dobro dizajnirane razine slične Pathfinder algoritmu, često daje i suprotan rezultat. Razine mogu djelovati vrlo isprekidano, igrač se teško snalazi na njima, a prepreke mogu blokirati pristup bitnijim elementima igre.

Rogue algoritam ima ocjenu 3, te je odmah vidljivo da se ovakva vrsta strukturiranja sadržaja s razlogom zadržala kao „zlatni“ standard ovog tipa igara. Igre koje koriste ovakav pristup strukturiranju i generiranju razina vrlo su igrive, pregledne i intuitivne, a kombiniranje otvorenih prostorija sa skučenim hodnicima daje igračima osjećaj predaha između istraživanja razine, ali i stvara stalnu želju za napredovanjem kroz igru i ispunjavanjem ciljeva.

8. Zaključak

Današnja računala i ostali uređaji namijenjeni pokretanju računalnih igara imaju i više nego dovoljno memorije za pohranu svog sadržaja potrebnog za izradu i pokretanje igara. Popularni i lako dostupni softverski paketi za izradu igara (poput Unity-a) znatno olakšavaju i ubrzavaju cijeli proces, te nude veliki izbor različitih alata za izradu svih vrsta komponenti i sadržaja potrebnog jednoj igri. Većina alata poput Unity-a ima i vlastitu elektroničku tržnicu na kojoj je moguće besplatno preuzeti ili kupiti gotove komponente, uključujući i radove profesionalnih dizajnera, umjetnika i programera, te ih je vrlo lako uklopiti u svoje projekte. Zbog svega ovog, proceduralno generiranje sadržaja u računalnim igrama je s vremenom izgubilo svoju originalnu namjenu i ulogu, ali se je i dalje vrlo prisutno u sferi izrade igara.

Budući da više nema potrebe za korištenjem proceduralnog generiranja sadržaja kao načina izbjegavanja drastičnih memorijskih ograničenja, ono je postalo prvenstveno dio stilske i konceptualne ideje igre. Programeri igara se danas često odlučuju za ovaj pristup zato jer misle da se savršeno uklapa u njihovu viziju gotove igre, i sve koncepte i ideje koje žele prenijeti igračima. Ako se želi napraviti jednostavna igra čiji je glavni ili jedini fokus na istraživanje ili akciju (kao što je primjer igre napravljene u sklopu ovog rada), proceduralno generiranje se može pokazati kao idealan izbor. S druge strane, ako je prvenstveni naglasak igre na radnji i priči, različitim likovima i njihovim interakcijama, ili pak rješavanju kompleksnijih problema i prepreka (poput logičkih i *puzzle* igara), proceduralno generiranje je vrlo teško ili gotovo nemoguće uklopiti u taj koncept, te je ručna izrada svega daleko bolji pristup.

Proceduralno generiranje također ima daleko više smisla koristiti u manjim projektima na kojima sudjeluje jedna ili vrlo mali broj osoba. Visokobudžetne igre su često vrlo kompleksne ispod površine, te sadrže velik broj različitih komponenata koje bi bilo vrlo teško povezati i uklopiti unutar bilo kakvih algoritama za kreiranje sadržaja. Igrači i kupci takvih igara također očekuju jako veliku razinu detaljnosti unutar igre (poput dobro razrađene priče i likova), što se ne uklapa u koncept proceduralnog generiranja. Još jedna prednost je i produljenje „životnog vijeka“ igre. Čak i vrlo skromne igre se mogu činiti jako velikima ili praktički beskonačnim zahvaljujući tome što je svaka razina igre drugačija, te je u teoriji moguće stvoriti gotovo beskonačnu količinu razina.

Ovakav koncept igrače potiče na stalno igranje iznova, te im stalno nudi nove prethnodno neviđene izazove, što je vrlo teško postići ako je igra izrađena na „tradicionalni“ način, sa predefiniranim sadržajem.

Iako je prvenstvena namjena Unity-a i sličnih softverskih alata olakšavanje izrade gotovih predefiniranih komponenti, poput razina unutar igre, ovime je pokazano da je itekako moguće kreirati sav željeni sadržaj u Unity-u pomoću algoritama. Čak i vrlo jednostavni algoritmi (npr. Board algoritam) mogu stvoriti zabavne igre, te su dovoljno fleksibilni da ih se može i prenamijeniti u druge svrhe. Kompleksniji algoritmi (npr. Tilemap algoritam) pokuzuju da je moguće povezati programski kod sa naprednijim Unity alatima i funkcionalnostima poput Tilemap-a, što donosi brojne prednosti kod performansi i fleksibilnosti. Također je moguće i povezati proceduralni pristup sa ručnom izradom sadržaja, te ručno uređivati proceduralno generirani sadržaj. Također se može zaključiti da postoji gotovo neograničen broj načina na koji se mogu osmisliti i realizirati algoritmi koji kreiraju elemente igre, te ne postoji strogo definirani dobar ili loš način, odnosno dobar ili loš algoritam. Sve opcije imaju svoje prednosti i mane, a o samoj osobi koja radi igru ovisi da odabere algoritam koji odgovara konceptu same igre, te njegovoj viziji konačnog projekta.

„Pod punom odgovornošću pismeno potvrđujem da je ovo moj autorski rad čiji niti jedan dio nije nastao kopiranjem ili plagiranjem tuđeg sadržaja. Prilikom izrade rada koristio sam tuđe materijale navedene u popisu literature ali nisam kopirao niti jedan njihov dio, osim citata za koje sam naveo autora i izvor te ih jasno označio znakovima navodnika. U slučaju da se u bilo kojem trenutku dokaže suprotno, spreman sam snositi sve posljedice uključivo i poništenje javne isprave stečene dijelom i na temelju ovoga rada“.

U Zagrebu, 26.08. 2018.

Popis kratica

RPG	*Role playing game*	igra s igranjem uloga
ASCII	*American Standard Code for Information Interchange*	Američki standardni kod za razmjenu informacija
IDE	*Integrated development environment*	integrirano razvojno okruženje
API	*Application programming interface*	programsko sučelje aplikacija

Popis slika

Popis kôdova

Literatura

[1] J. GIBSON; *Introduction to Game Design, Prototyping, and Development*; Addison-Wesley Professional; (2014); 978-0321933164

[2] R. NYSTROM; *Game Programming Patterns*; Genever Benning; (2014); 978-0990582908

[3] S. MADHAV; *Game Programming Algorithms and Techniques*; Addison-Wesley Professional; (2013); 978-0321940155

[4] M. SMITH, C. QUEIROZ; *Unity 5.x Cookbook*; Packt Publishing; (2015); 978-1784391362

[5] T. LINTRAMI; *Unity 2017 Game Development Essentials - Third Edition*; Packt Publishing; (2018); 978-1786469397

[6] T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, C. STEIN; *Introduction to Algorithms, 3rd Edition*; The MIT Press; (2009); 978-0262033848

[7] R. WATKINS; *Procedural Content Generation for Unity Game Development*; Packt Publishing; (2016); 978-1-78528-747-3