Proceduralno generiranje scena u Unity-u

Hudelist, Marin

Undergraduate thesis / Završni rad

2023

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Rijeka / Sveučilište u Rijeci**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:195:103411

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-09-13**

Repository / Repozitorij:

Repository of the University of Rijeka, Faculty of Informatics and Digital Technologies - INFORI Repository

Sveučilište u Rijeci, Fakultet informatike i digitalnih tehnologija

Sveučilišni prijediplomski studij Informatika

Marin Hudelist

Primjena proceduralnog generiranja scena u Unityju

Završni rad

Mentor: Doc. dr. sc. Miran Pobar

Rijeka, rujan 2023.g

Sadržaj

1. Uvod 1

2. Proceduralno generiranje: razvoj, prednosti i izazovi 2

2.1. Kratka povijest proceduralnog generiranja 2

2.2. Prednosti proceduralnog generiranja 3

2.3. Izazovi proceduralnog generiranja 4

3. Tehnike i algoritmi proceduralnog generiranja 5

3.1. Osnovne tehnike i algoritmi 5

3.2. Integracija sa novim tehnologijama 9

4. Implementacija proceduralnog generiranja u igri „Deepsea Drifter“ 10

4.1. Kratki opis neprijatelja 11

4.2. Algoritam i tehnika za proceduralno generiranje 11

5. Implementacija algoritma 16

5.1. Razmotrene tehnike za proceduralno generiranje tamnice 16

5.1.1. Generiranje tamnica koristeći Cellular Automata 16

5.1.2. Generiranje tamnica koristeći nasumično postavljanje soba 17

5.2. Objašnjenje koda 18

5.2.1. Pronalaženje glavnih soba 18

5.2.2. Generacija soba 19

5.2.3. Izvođenje Delaunay triangulacije 20

5.2.4. Izrada minimalnog razapinjućeg stabla (MST) 22

5.2.5. Odvajanje soba i pozivanje ostalih metoda 23

5.2.6. Generacija koridora 23

5.2.7. Ostale metode 24

6. Testiranje proceduralno generiranog sadržaja 27

6.1. Provedeno testiranje 28

Zaključak 31

Popis korištenih assetova 32

Popis slika 33

Popis kratica 34

Popis kodova 35

Literatura 36

Prilog 1 Kod metode SeparateRooms 37

Prilog 2 Kod za generiranje hodnika 40

Metoda CreateCorridors 40

Metoda CalculateEdgeCenter 41

Metoda DetermineTurningPoint 42

Metoda IsIntersectingRoomBuffer 43

Metoda DrawCorridor 43

1. Uvod

U svijetu razvoja igara koji se stalno razvija, potraga za stvaranjem ekspanzivnih i nepredvidljivih svijetova dovela je do porasta uporabe proceduralnog generiranja,tehnike koja koristi algoritme za automatsko generiranje sadržaja. Proceduralno generiranje počiva na algoritmima koji mogu proizvesti ogromne količine sadržaja, osiguravajući da je iskustvo svakog igrača jedinstveno, ali koherentno.

Povijesno gledano, proceduralno generiranje vuče korijene iz rane računalne grafike i simulacija. Međutim, njegova primjena u igrama promijenila je način na koji programeri pristupaju dizajnu igara, posebno u igrama gdje bi samu veličinu svijeta igre bilo nepraktično dizajnirati ručno. Od beskonačnih terena "Minecrafta" do prostranih galaksija "No Man's Sky“ proceduralno generiranje pokazalo se neprocjenjivim alatom u arsenalu programera igara (*engl. Game Developera*).

Ipak, pravi potencijal proceduralnog generiranja nije samo za svrhu korištenja u AAA igrama, već i u inovativnim projektima koje poduzimaju pojedinačni programeri i entuzijasti. Jedan primjer toga bi bio projekt koji sam izradio u sklopu ovog završnog rada— generator tamnica(*engl. Dungeon*) gdje pomoću grafova proceduralno generiramo *dungeone* koje igrač može istraživati. Ovaj projekt, koji će biti detaljno opisan, služi kao dokaz svestranosti i dubine proceduralnog generiranja.

U ovom završnom radu proći ćemo kroz zamršenost proceduralnog generiranja, njegove prednosti i izazove, značajne primjere u industriji igara i duboko zaroniti u proceduralno generiranje grafova primijenjeno u mom projektu generatora tamnica.

2. Proceduralno generiranje: razvoj, prednosti i izazovi

U golemom krajoliku razvoja igara malo je tehnika koje su privukle toliko pažnje i uzrokovale toliko rasprava kao proceduralno generiranje. U svojoj srži, proceduralno generiranje je algoritamsko stvaranje sadržaja, metoda koja je u suprotnosti s tradicionalnim ručno izrađenim dizajnom (Hendrikx, 2013).Ovaj dinamični pristup stvaranju sadržaja promijenio je način na koji programeri zamišljaju i konstruiraju virtualne svjetove, likove, priče i više. Od svojih skromnih početaka u ranim danima računarstva do široko rasprostranjene primjene u današnjim igrama, proceduralno generiranje kontinuirano se razvijalo, donoseći sa sobom mnoštvo prednosti i izazova. Ovaj odjeljak zaranja u bogatu povijest proceduralnog generiranja, razjašnjava njegove brojne prednosti i suočava se sa složenošću i dilemama koje postavlja programerima i dizajnerima igara

2.1. Kratka povijest proceduralnog generiranja

Koncept proceduralnog generiranja, iako je danas duboko ukorijenjen u modernim igrama, ima korijene koji sežu dalje nego što bi se u početku moglo pretpostaviti. Prije primjene u igrama, proceduralno generiranje izvorno je započelo u području računalne grafike tijekom 1960-ih i 1970-ih (Hendrikx, 2013). Pioniri u ovoj domeni iskoristili su algoritme za izradu raznoraznih kompleksnih uzoraka, oblika i animacija. Ovo razdoblje postavilo je temeljna načela algoritamskog generiranja sadržaja.Proceduralno generiranje u igrama se prvi put primjećuje 1980-ih godina koje su označile značajan pomak jer su programeri igara počeli uviđati potencijal proceduralnih tehnika. Budući da hardverska ograničenja i ograničenja pohrane predstavljale značajn izazov, proceduralno generiranje pojavilo se kao rješenje. Rane igre poput "Elite" zadivile su igrače nudeći goleme zvjezdane sustave, svi generirani proceduralno unutar ograničenja memorije (Hendrikx, 2013). Kako je tehnologija napredovala, tako su napredovale i mogućnosti proceduralnog generiranja. Kasnih 1990-ih i 2000-ih svjedočimo porastu njegova prihvaćanja, osobito u igrama i simulacijama otvorenog svijeta (Hendrikx, 2013). Naslovi kao što su "Minecraft" postali su pioniri, demonstrirajući kako se ekspanzivni i nepredvidivi svjetovi igara mogu izraditi pomoću proceduralnih algoritama.

Danas se opseg proceduralnog generiranja proteže daleko izvan samih svjetova igara. Koristi se u samom dizajnu likova, priča, generiranju glazbe i slično Tehnologija je sazrela, omogućujući programerima da proizvedu detaljan i složen sadržaj, obogaćujući iskustva igrača kao nikada prije.

2.2. Prednosti proceduralnog generiranja

Jedna od najuvjerljivijih prednosti proceduralnog generiranja je njegova sposobnost da proizvede beskrajan niz sadržaja (Hendrikx, 2013). Igre koje koriste ovu tehniku mogu ponuditi igračima goleme krajolike koji se stalno mijenjaju, osiguravajući da svako igranje nudi svježe i jedinstveno iskustvo. Ova nepredvidivost dodaje sloj mogućnosti ponovnog igranja, jer igrači mogu kontinuirano istraživati nove mape, izazove i priče. Još jedna poprilično uvjerljiva prednost bi bilo učinkovito iskorištavanje resursa koje proceduralno generiranje nudi. Tradicionalni dizajn igara često zahtijeva ogromne količine prostora za pohranu ručno izrađenih sredstava. Proceduralno generiranje, s druge strane, koristi algoritme za generiranje sadržaja u hodu, značajno smanjujući potrebu za pohranom. Ova učinkovitost posebno je korisna za programere koji rade s ograničenim resursima ili žele stvoriti resursno neintenzivne igre. Osim što nudi resursnu učinkovitost proceduralno generiranje također pruža troškovnu i vremensku učinkovitost (Hendrikx, 2013). Ručna izrada detaljnih svjetova igre dugotrajan je i radno intenzivan proces. Proceduralno generiranje može automatizirati veliki dio toga, omogućujući programerima da proizvedu ekspanzivan sadržaj u kratkom vremenskom roku. Ta vremenska učinkovitost se prevodi u smanjene troškove razvoja, što čini proceduralno generiranje atraktivnom opcijom, posebno za indie programere. Proceduralno generiranje može se također koristiti za prilagođavanje iskustava u igri pojedinačnim igračima. Analizirajući ponašanje igrača, igre se mogu prilagoditi i stvoriti sadržaj koji je usklađen s preferencijama igrača, osiguravajući zanimljivije i osobnije iskustvo. Proceduralno generiranje uvelike olakšava posao indie programera ili malih timova, što sam i sam zamjetio tijekom razvoja igre, umjesto da sam ručno dizajnirao i smišljao tamnicae proceduralno generiranje mi je omogućilo stvaranje bezbroj kombinacija mapa. Uz ograničene timove i proračune, ručno stvaranje golemih svjetova igara može biti vremenski zahtjevno. Proceduralne tehnike osnažuju programere igara po pitanju stvaranja ekspanzivnih i okruženja za igre donekle izjednačavajući uvjete s većim studijima.

2.3. Izazovi proceduralnog generiranja

Iako proceduralno generiranje nudi mnoštvo prednosti, nije bez izazova. Njegova učinkovita implementacija zahtijeva duboko razumijevanje tehnika proceduralnog programiranja i potencijalnih zamki u koje programeri mogu upasti. Evo nekih od primarnih izazova povezanih s proceduralnim generiranjem:

- **Kvaliteta sadržaja**: Jedan od najznačajnijih izazova s proceduralno generiranim sadržajem je osiguranje dosljedne kvalitete. Za razliku od ručno izrađenog sadržaja, gdje dizajneri imaju izravnu kontrolu nad svakim detaljem, proceduralni sadržaj ponekad može proizvesti neočekivane ili neželjene rezultate. Osigurati da svaki proizvedeni komad zadovoljava željene standarde kvalitete može biti zahtjevan zadatak.
- **Balansiranje nasumičnosti i igrivosti**: Uspostavljanje prave ravnoteže između nasumičnosti i koherentnog iskustva igranja je ključno. Dok nepredvidljivost može dodati uzbuđenje, bitno je da generirani sadržaj ostane igriv i logičan.
- **Složenost algoritama**: Dizajniranje algoritama koji proizvode raznolik, ali koherentan sadržaj može biti komplicirano. Fino podešavanje ovih algoritama za postizanje željenih rezultata zahtijeva duboku stručnost i sam proces može biti poprilično dugotrajan.
- **Gubitak osobnog dodira**: Neki kritičari tvrde da proceduralno generiranom sadržaju može nedostajati dubina koja dolazi s ručno izrađenim dizajnom. Osigurati da igra zadrži svoju recimo „dušu“ uz korištenje proceduralnog generiranja može biti izazovno.
- **Zabrinutost u pogledu performansi**: Generiranje sadržaja u hodu, osobito u stvarnom vremenu, može zahtijevati mnogo resursa. Osiguravanje glatke izvedbe bez ugrožavanja bogatstva sadržaja zahtijeva optimizaciju i učinkovitu praksu kodiranja.
- **Preopterećenost igrača**: Beskonačni krajolici i beskrajni sadržaj ponekad mogu dovesti do preopterećenosti igrača. Ključno je osmisliti igru koja vodi i zaokuplja igrača, a da se ne osjeća izgubljeno u beskrajnom prostranstvu.

Iako ti izazovi mogu predstavljati prepreke, oni također nude prilike za inovacije. Prevladavanje ovih izazova zahtijeva spoj tehničke stručnosti, kreativnosti i dubokog razumijevanja očekivanja i preferencija igrača.

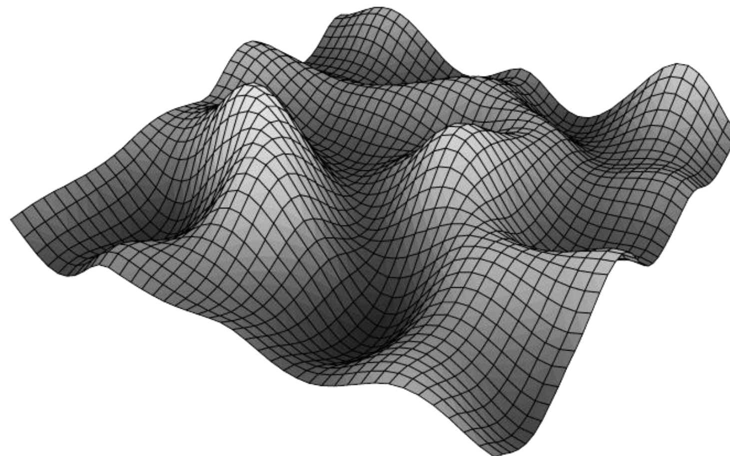
3. Tehnike i algoritmi proceduralnog generiranja

Proceduralno generiranje ima niz tehnika i algoritama koji se koriste već dugi niz godina te kako se tehnologija nastavlja razvijati, integracija ovih metoda s drugim tehnologijama dodatno pojačava potencijal proceduralnog generiranja u igrama.

3.1. Osnovne tehnike i algoritmi

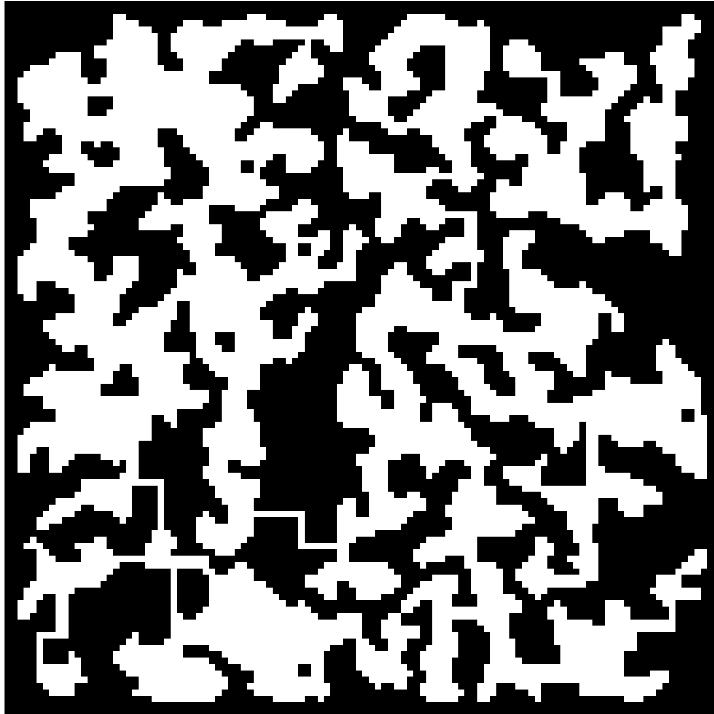
Proceduralno generiranje se temelji na bogatom nizu tehnika i algoritama koji se i dan danas razvijaju te od kojih je svaki prilagođen specifičnim vrstama stvaranja sadržaja. Te ćemo mi objasniti par popularnih tehnika (Togelius J., 2016)

- **Perlin Noise**: Koju je predstavio Ken Perlin 1980-ih se koristi za stvaranje organskih uzoraka prirodnog izgleda. Radi interpolacijom između niza slučajnih gradijentnih vektora u mreži. Rezultat je glatka, kontinuirana funkcija koja je idealna za stvaranje terena, vodenih površina i formacija oblaka. Njegova koherentnost i kontinuitet čine ga omiljenim za simulaciju prirodnih pojava u igrama.



Slika 3.1 Primjer terena generiranog pomoću Perlin Noise-a (Izvor: https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/perlin-noise-part-2/perlin-noise-terrain-mesh.html)

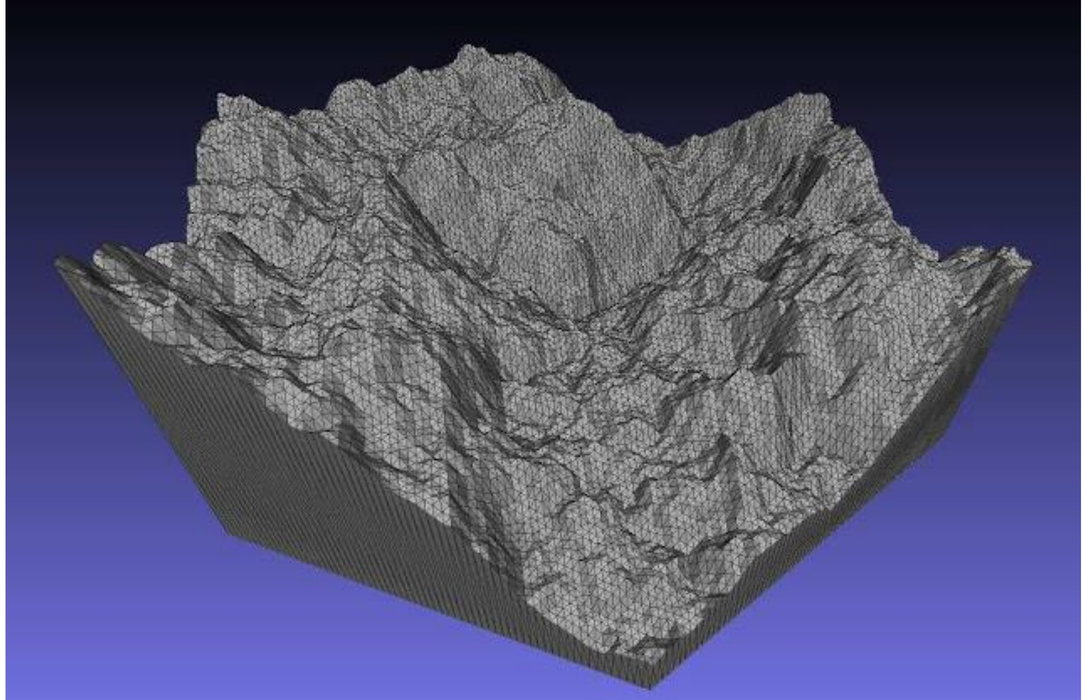
- **Cellular Automata**: Ovo je tehnika gjde se skup ćelija razvija kroz vrijeme prateći neki skup pravila. Jedan od najpoznatijih primjera je Conwayeva Igra života. U kontekstu razvoja igrica, cellular automata mogu se koristiti za simulaciju dinamičkih sustava poput širenja požara, toka vode ili za generiranje struktura nalik špiljama razvijanjem početne nasumične distribucije stanica.



Slika 3.2 Primjer špilje/tamnice generiran pomoću Cellular Automata (Izvor: https://www.gridsagegames.com/blog/2014/06/mapgen-cellular-automata/(Pristupljeno 23.9.2023))

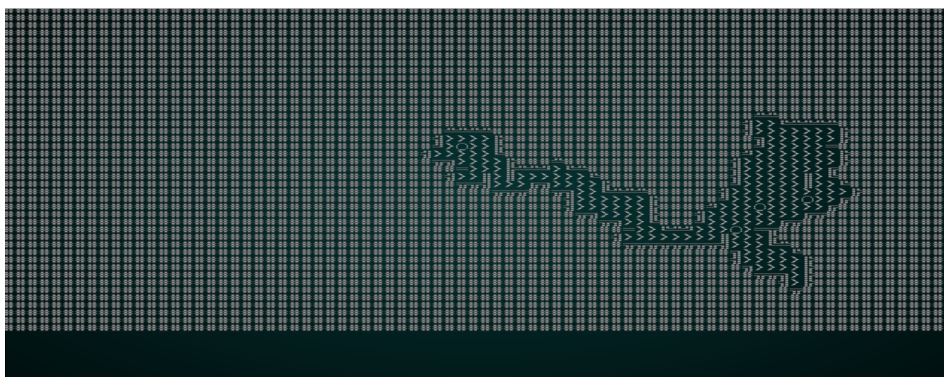
- **Voronoi dijagrami**: Ova tehnika dijeli neku danu ravninu na regije na temelju udaljenosti do skupa početnih točaka. Svaka točka u regiji bliža je svojoj početnoj točci nego bilo koja druga. U igricama, Voronoi dijagrami se mogu koristiti za definiranje teritorija, stvaranje mapa strateških igara ili generiranje terena koji izgledaju organski.

- **Generiranje fraktala**: Fraktali su strukture koje izgledaju slično na bilo kojoj razini povećanja. Ovo svojstvo samosličnosti iskorištava se u igrama za stvaranje obala, planina i drugih prirodnih struktura. Mandelbrotov skup i Julijin skup klasični su primjeri fraktala.



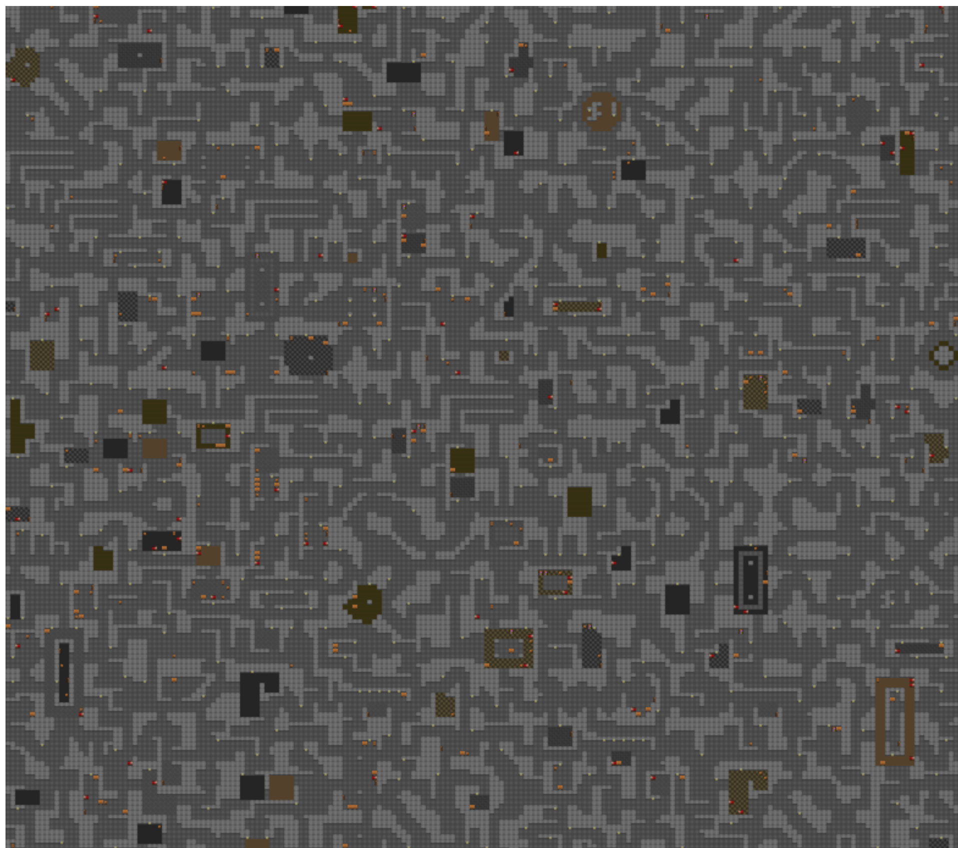
Slika 3.3 Primjer generiranih planina pomoću proceduralne generacije temeljene na fraktalima (Izvor:http://alexanderpruss.blogspot.com/2017/03/super-simple-fractal-terrain-generator.html (Pristupljeno 23.9.2023)

- **Algoritmi za labirint i tamnicu**: Stvaranje kompleksnih labirinata i tamnica sastavni je dio mnogih igara. Algoritmi kao što su Primov, Kruskalov i Drunkard Walk mogu generirati mape s različitim razinama složenosti i dizajna, osiguravajući da se igrači svaki put suoče s jedinstvenim izazovima.



Slika 3.4 Primjer procesa proceduralnog generiranja tamnice pomoću Random Walk algoritma (Izvor: https://bfnightly.bracketproductions.com/chapter_28.html (Pristupljeno 23.9.2023))

- **Noise funkcije**: osim Perlin Noisea, postoje i druge funkcije buke koje programeri koriste. Simplex Noise, nasljednik Perlin Noisea, nudi bržu alternativu koja je manje sklona nekim neželjenim rezultatima. Worley Noise, još jedan primjer Noise funkcije se, s druge strane, temelji na udaljenosti do najbliže točke u skupu nasumičnih točaka, stvarajući stanične uzorke.
- **Tile-ing i Wang tileovi**: Tile-ing uključuje ispunjavanje prostora uzorcima bez očitih ponavljanja. Wang tileovi su kvadratni tileovi s obojenim rubovima. Kada se postavljaju odgovarajuće se boje moraju dodirivati, što dovodi do uzoraka koji se ne ponavljaju te su idealne za generiranje teksture tla ili rasporeda nekog grada.



Slika 3.5 Proceduralno generirana mapa koristeći Wang Tile-ove (Izvor: https://nothings.org/gamedev/herringbone/ (Pristupljeno 23.9.2023)

- **Generiranje rijeka i cesta**: Ovi algoritmi simuliraju prirodne procese. Za rijeke, tipičan pristup može uključivati simulaciju protoka vode od visokih prema niskim područjima, urezivanje riječnih korita. Za ceste, algoritmi za traženje putanje poput A* mogu odrediti najučinkovitiju rutu između točaka, uzimajući u obzir teren i druge prepreke.

Svaka od ovih tehnika i algoritama nudi programerima igara alat za izradu bogatih, dinamičnih i impresivnih svjetova. Njihova primjena može varirati od makro, kao što je izgradnja svijeta, do mikro, kao što je detaljisanje tekstura ili lišća, naglašavajući njihovu svestranost i važnost u području razvoja igara.

3.2. Integracija sa novim tehnologijama

Područje razvoja igara u stalnom je stanju evolucije, s novim tehnologijama koje neprestano preoblikuju krajolik. Proceduralna generacija, sa svojom dinamičnom i prilagodljivom prirodom besprijekorno se integrira s ovim tehnologijama kako bi se stvorila kvalitetnija iskustva igranja. Neke od tehnologija gdje proceduralno generiranje može imati primjenu su (Russel, 2016):

- Virtualna stvarnost (VR) *(engl. Virtual Reality):* Proceduralno generiranje u VR-u može stvoriti beskonačne, imerzivne svjetove, omogućujući igračima da istražuju goleme krajolike ili okruženja koja reagiraju i razvijaju se u stvarnom vremenu. Još jedna primjena koju vrijedi spomenuti bi bile dinamičke interakcije. Dok se igrači nalaze u VR okruženju, proceduralni algoritmi mogu prilagoditi svijet oko sebe, osiguravajući jedinstveno iskustvo sa svakim igranjem.
- Proširena stvarnost (AR*) (engl. Augmented Reality)*: Za AR igre, proceduralno generiranje može postaviti dinamičke elemente igre na stvarni svijet. To može uključivati stvorenja koja, blaga ili izazove na temelju fizičkog okruženja igrača. Jedan popularni primjer AR igre bi bio Pokemon Go.
- Umjetna inteligencija (AI) *(engl. Artificial Intelligence)*: Spoj umjetne inteligencije i proceduralnog generiranja može dovesti do pametnijih algoritama koji prilagođavaju sadržaj na temelju ponašanja igrača, preferencija i povratnih informacija. To rezultira visoko personaliziranim i evoluirajućim iskustvima igre. Proceduralno generirani likovi koji nisu igrači (NPC-ovi *(engl. Non Playable Character)*) pokretani AI-jem mogu pokazivati složenija ponašanja, reakcije i interakcije, čineći svijet igre življim.
- Neuronske mreže: Neuronske mreže mogu se trenirati na ručno izrađenom sadržaju igre, što im omogućuje stvaranje novog sadržaja koji zadržava osjećaj i kvalitetu dizajna koje su izradili ljudi.

4. Implementacija proceduralnog generiranja u igri „Deepsea Drifter“

"Deepsea Drifter" stavlja igrače u ulogu istraživača koji istražuje misteriozne dubine podvodnog svijeta. Igrač igru započinje na ručno izrađenoj mapi u kojoj se stvori par skupova neprijatelja, igraču je cilj tu mapu istraživati te pronaći na mapi napuštene ruševine te nakon ulaska u ruševinu igrač izlazi iz svog vozila te ulazi u proceduralno generirani dungeon gdje mora proći sve sobe i poraziti neprijatelje. Ovi dungeoni, različiti po svom dizajnu i izazovima, generiraju se proceduralno, osiguravajući da je svaki dungeon jedinstveni test vještine i strategije. Sam algoritam korišten za proceduralno generiranje će se detaljnije objasniti. Po pitanju napretka koji igrači mogu očekivati kroz igranje igre, zamišljeno je da igrači sakupljaju dvije vrste valuta: privremenu valutu i trajnu valutu Poražavanje neprijatelja izvan tamnica daje igraču privremenu valutu, koja se može potrošiti tijekom tijekom igre za kupnju nadogradnji za igrača ili njegovo vozilo. Ove nadogradnje pomažu igračima u njihovom trenutnom istraživanju. Nasuprot tome, nagrade dobivene osvajanjem proceduralno generiranih tamnica dolaze u obliku trajne valute. Ovaj vrijedan resurs rezerviran je za kraj svake prolaska igre, omogućujući igračima da ulažu u trajne nadogradnje. Ova poboljšanja vrijede tijekom cijele igre, simbolizirajući rast i napredovanje igrača tijekom njegovih avantura.

Slika 4.1 Prikaz izgleda neprijatelja, vozila igrača i UI-a

4.1. Kratki opis neprijatelja

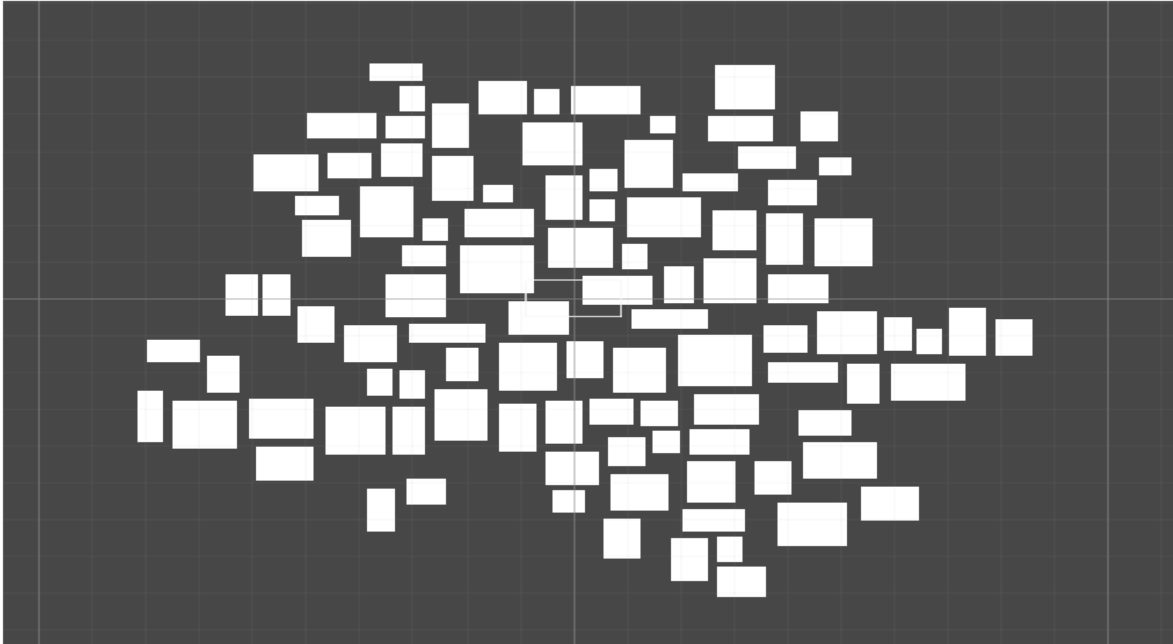
U igri trenutno postoje dva neprijatelja koja su implementirana za vanjski svijet odnosno za mapu izvan tamnica. Neprijatelj koji ima blisko dometne napade i neprijatelj koji ima dalekodometne napade. Neprijatelj sa blisko dometnim napadama je zamišljen kao neki oblik agresivne meduze te mu je cilj približiti se igraču i odraditi napad koji zahvaća područje u radijusu oko neprijatelja. Dalekodometni neprijatelj je zamišljen kao agresivni ježinac koji drži udaljenost od igrača te puca svoje šiljke na njega. Neprijatelji u tamnicama su trenutno konceptualni te nisu implementirani te iako su specifičnosti neprijatelja iz tamnica još u fazi dizajna, zamišljene su dvije vrste neprijatelja. Neprijatelj koji luta hodnicima tamnica. Brz i agresivan jurišat će na igrače čim ih vidi, tjerajući ih da iskoriste svoje okruženje kako bi se obranili od njegovih napada. Te druga vrsta neprijatelja koja je zamišljena kao statični neprijatelj koji puca na igrača. Stacioniran na strateškim točkama, neprijatelj će ciljati i pucati na igrače, zahtijevajući brze reflekse i strategije temeljene na zaklonu za sigurno kretanje kroz tamnicu. Ovi protivnici, kako u otvorenom svijetu tako i unutar tamnica, osiguravaju da su igrači uvijek u stanju pripravnosti.

4.2. Algoritam i tehnika za proceduralno generiranje

U "Deepsea Drifteru", proceduralno generiranje tamnica ključna je značajka, osiguravajući da svaki prolaz igre nudi jedinstveno iskustvo. Algoritam koji se koristi je mješavina računalne geometrije, teorije grafova i klasičnih tehnika razvoja igara. Samo generiranje se provodi kroz par koraka.

1. Generacija soba:

Proces počinje stvaranjem prostorija. Određeni broj soba (numberOfRooms) stvara se s nasumičnim položajima i veličinama unutar zadanih granica (minRoomSize i maxRoomSize). Svaka je soba predstavljena kao prefab (roomPrefab) i instancirana u svijetu igre.



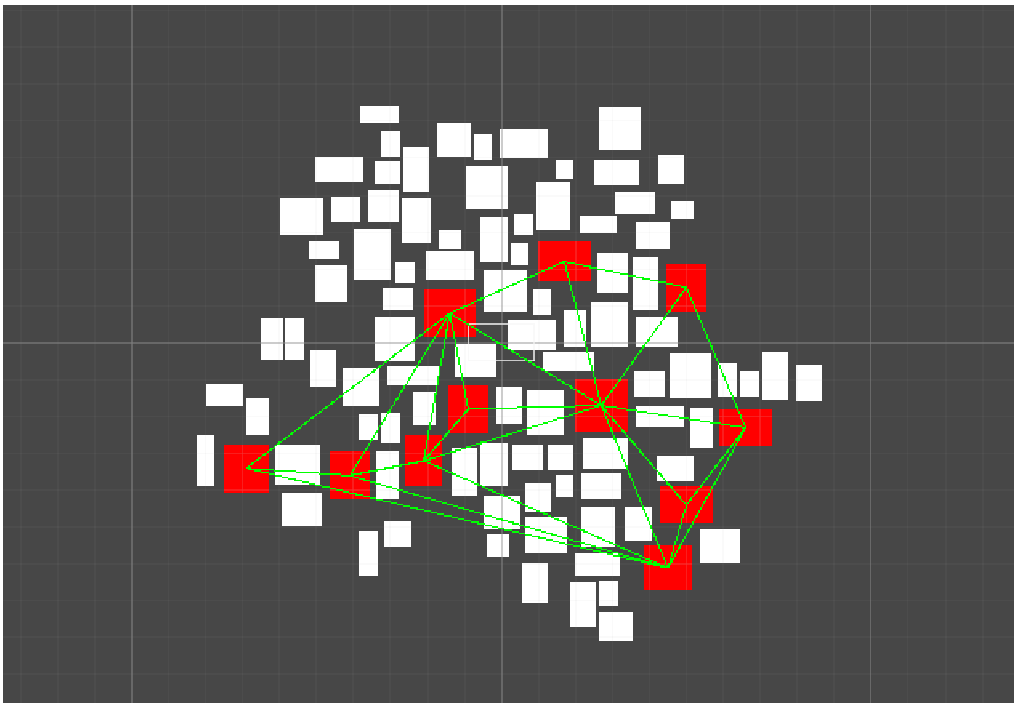
Slika 4.2 Visualizacija generiranih prostorija pomoću 2D spriteova.

2. Odvajanje soba:

Kako bi se spriječilo preklapanje soba, primjenjuje se jednostavni algoritam odvajanja. Ovaj iterativni proces odvaja sobe sve dok ne prestanu preklapanja. Razdvajanje osigurava da se sobe ne preklapaju te daje mjesta za spajanje prostorija. Nakon što su sobe odvojene bira se 10 do 15 glavnih soba koje se označuju na način da se oboje u crveno.

3. Delaunay triangulacija:

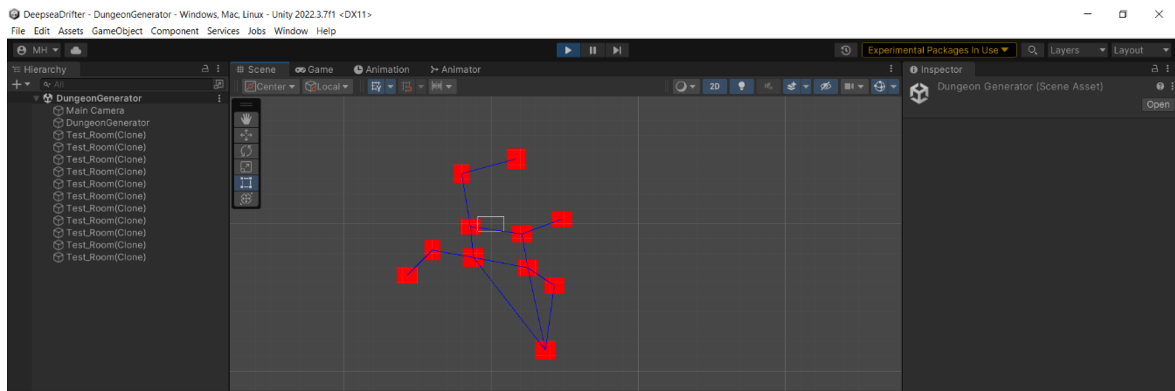
Za povezivanje soba koristi se Delaunayeva triangulacija pomoću bibiloteke Habrador Computational Geometry dostupne na Unity-u. Delaunayeva triangulacija, nazvana po ruskom matematičaru Borisu Delaunayu, tehnika je geometrijskog dijeljenja koja se koristi za rastavljanje skupa točaka u ravnini na skup trokuta koji se ne preklapaju (de Berg, 2008).Ključno svojstvo Delaunayeve triangulacije je da maksimizira minimalni kut svih kutova trokuta. Drugim riječima, stvara trokute koji su što je moguće bliži jednakostraničnim, što često rezultira "pravilnijim" trokutima. Jedna od primarnih karakteristika Delaunayeve triangulacije je da nijedna točka iz izvornog skupa točaka ne smije pasti unutar opisanog kruga (kružnice koja prolazi kroz sva tri vrha) bilo kojeg trokuta u triangulaciji (de Berg, 2008). Ovo svojstvo osigurava da su trokuti „pravilni" i da nemaju dugačke, tanke oblike. Cilj Delaunayeve triangulacije je kao što smo već spomenuli maksimiziranje minimalnog kuta svih trokuta. To znači da će čak i u nepravilnim raspodjelama točaka rezultirajući trokuti biti što je moguće bliži jednakostraničnim. Ovo je svojstvo posebno vrijedno u primjenama kao što su analiza konačnih elemenata i generiranje mreže, gdje dobro oblikovani trokuti dovode do preciznijih rezultata. Te u našem slučaju ova tehnika osigurava da su sobe povezane na način koji smanjuje ukupnu duljinu hodnika. Triangulacija stvara graf gdje su sobe čvorovi, a rubovi predstavljaju potencijalne hodnike.



Slika 4.3 Visualizacija Delaunay triangulacije

4. Minimalno razapinjuće stablo (MST)*(engl. Minimum Spanning Tree):*

Iz Delaunayeva grafa generira se minimalno razapinjuće stablo (MST). Minimalno razapinjuće stablo (MST) ključan je koncept u teoriji grafova i računalnoj znanosti, koristi se za povezivanje svih čvorova u povezanom, neusmjerenom grafu uz minimiziranje zbroja težina bridova. MST je stablo, što znači da je povezan graf bez ikakvih ciklusa. Povezuje sve čvorove (vrhove) izvornog grafa. Primarni cilj MST-a je kao što je ranije spomenuto minimizirati ukupnu težinu (ili cijenu) bridova, a istovremeno osigurati da su svi čvorovi povezani. Svaki rub u MST-u povezan je s težinom ili cijenom, a zbroj tih težina je minimiziran. Minimalna razapinjuća stabla nalaze primjenu u raznim domenama no u našem slučaju uporabili smo ga u razvoju igara. Nekoliko algoritama može pronaći MST danog grafa. Najpoznatiji algoritmi uključuju Kruskalov, Primov i Boruvkin algoritam. Mi smo koristili Kruskalov algoritam koji počinje s praznim grafom i iterativno dodaje rubove MST-u izbjegavajući cikluse. Razvrstava rubove po težini i dodaje ih jednog po jednog ako ne stvaraju ciklus. Ovo osigurava da su sve sobe povezane bez ikakvih ciklusa, pružajući temelj za raspored tamnice i hodnika. Nakon što se MST generira u njega se dodaju natrag neki ciklusi minimalno 2 maksimalno 5 ciklusa kako bi raspored hodnika bio zanimljiviji i da se izbjegne linearna tamnica



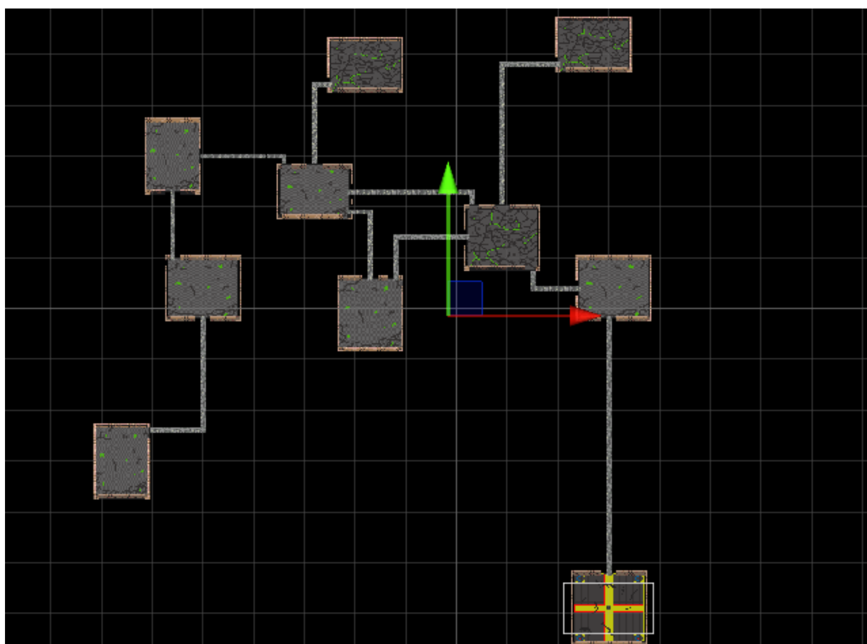
Slika 4.4 Visualizacija MST-a sa uklonjenim preostalim sobama i vraćenim ciklusima

5. Stvaranje hodnika:

Pomoću rubova MST-a nacrtani su hodnici koji povezuju sobe. Ovisno o međusobnom položaju povezanih prostorija, hodnici mogu biti ravni ili u obliku slova L. Hodnici su predstavljeni pomoću prefaba (corridorPrefab) i instancirani u svijetu igre. Ovo je kratki pregled i koraci kroz koje algoritam prolazi. Sljedeći odjeljak zaronit će dublje u kod, razjašnjavajući funkcionalost svake funkcije i njezinu ulogu u procesu generiranja.

6. Konačni rezultat

Na kraju cijelog procesa nakon što dodamo sve uređene prefabove dobijemo rezultat prikazan na priloženim slikama.



Slika 4.5 Prikaz cijele generirane tamnice

Slika 4.6 Prikaz sobe u kojoj se igrač instancira i instance igrača vidljive na sredini sobe

5. Implementacija algoritma

Sam proces implementacije algoritma je bio dugotrajan te su testirane razne tehnike i algoritme prije nego što je postignut zadovoljavajuć izgled generiranih tamnica.

5.1. Razmotrene tehnike za proceduralno generiranje tamnice

Proces proceduralnog generiranja često je iterativan, s različitim tehnikama koje se testiraju i usavršavaju kako bi se postigao željeni ishod. Za "Deepsea Drifter", putovanje do konačnog algoritma za generiranje tamnica započelo je eksperimentiranjem s nekoliko metoda dok nije postignut željeni rezultat.

5.1.1. Generiranje tamnica koristeći Cellular Automata

Cellular automata je tehnika u proceduralnom generiranju koja je ranije ugrubo objašnjena no ovdje ću proći malo detaljnije kroz tu metodu. Sastoji se od mreže ćelija, koje mogu biti u jednom od konačnog broja stanja. Mreža može biti u bilo kojem dimenzionalnom prostoru, ali za generiranje tamnica, 2D mreža je najčešća. Stanje svake ćelije razvija se kroz vrijeme, a prijelazi stanja određeni su njezinim trenutnim stanjem i stanjima susjednih ćelija, na temelju unaprijed definiranih pravila. Započinjemo s mrežom gdje je svakoj ćeliji nasumično dodijeljeno stanje: zid (puno) ili prostor (prazno). Tijekom nekoliko iteracija, primijenjujemo pravila na svaku ćeliju istovremeno. Korištena pravila su poprilično standardna za generiranje tamnica:

- Birth pravilo: Ako prazna ćelija ima točno X zidova kao susjeda, postaje zid.
- Survival pravilo: Ako zid ima Y ili Z zidove kao susjede, on ostaje zid. U suprotnom, postaje prazan.
- Death pravilo: Ako zid ima manje od W zidova kao susjeda, postaje prazan.

Mreža se razvija tijekom određenog broja ponavljanja, postupno se stabilizirajući u različite obrasce te se broj ponavljanja može prilagoditi kako bi se kontrolirala složenost i povezanost rezultirajuće strukture. Cellular automata mogu proizvesti kompleksne, međusobno povezane strukture koje se doimaju organski i prirodno te je sama metoda vrlo prilagodljiva, s tim da na ishod značajno utječu početni uvjeti i skupovi pravila. Razlog zašto sam odustao

od korištenja Cellular Automata metode za generiranje tamnica je bio problem sa rasporedom soba. Rezultirajući rasporedi cellular automata često nalikuju prirodnim špiljama, koje, iako zanimljive, nisu bile usklađene sa željenim strukturama, te je bio problem sa samom navigacijom tamnicama odnosno, bez dodatne naknadne obrade, generiranim tamnicama možda nedostaju jasni putevi što ih čini manje prikladnima za igranje. Dok pristup staničnih automata nudi jedinstvenu metodu za stvaranje struktura nalik špiljama, nedostatak kontrole nad postavljanjem prostorija i formiranjem hodnika doveo je do istraživanja drugih tehnika. Želja za strukturiranijim tamnicama s čistim sobama i hodnicima potaknula je prelazak na metodu nasumičnog postavljanja soba u kombinaciji s naprednim tehnikama generiranja hodnika.

5.1.2. Generiranje tamnica koristeći nasumično postavljanje soba

Pristup nasumičnog postavljanja soba s generiranjem koridora je tehnika koja se koristi za dizajniranje tamnicama koje uspostavljaju ravnotežu između slučajnosti i strukturiranog dizajna. Prvi korak uključuje označavanje određene mreže ili ograničenog prostora unutar kojeg će se instancirati sobe. Unutar tog unaprijed definiranog prostora, sobe se generiraju na temelju nasumičnih vrijednosti koje određuju njihov položaj, širinu i visinu. Ključni aspekt ovog koraka je osigurati da se prostorije ne preklapaju. Ako dođe do preklapanja, pokreću se mehanizmi za prostorno odvajanje prostorija, čuvajući cjelovitost granica svake sobe. Dimenzionalna ograničenja: Kako bi se unijela varijabilnost u izgled tamnice i kako bi se osiguralo da su sobe pogodne za igranje, ograničenja su nametnuta njihovim dimenzijama. To uključuje postavljanje minimalnih i maksimalnih pragova za veličine prostorija. Izvorni pristup mi je bio nasumično postavljanje soba na toj mreži, no odlučio sam pokušati različitu metodu gdje se sve sobe instanciraju u uskom prostoru te dodajući im 2D Collider u Unity-u da „eksplodiraju" od te točke generacije te da algoritam za odvajanje soba osigura da se ne preklapaju. Po pitanju generacije koridora izrazito je bitna Delaunayeva triangulacija. Ova tehnika je ključna u osiguravanju da su sobe međusobno povezane. Triangulacijom soba uspostavlja se mreža koja osigurava sveobuhvatnu dostupnost kroz sobe. Nakon triangulacije, kao što je ranije objašnjenjo izrađuje se MST sa ciljem pojednostavljenja veza između soba. To osigurava izravan, nesmetan put između prostorija, eliminirajući suvišne veze. Početni pristup za generiranje koridora bilo je korištenje algoritma A*, poznate tehnike pronalaženja puta. Međutim, njegova primjena nije odgovarala željenom rasporedu tamnice pošto su sami hodnici bili predirektni odnosno

većina hodnika je bila ravna ili kosa crta od sobe u sobu što nije odgovaralo mom dizajnu. To je dovelo do istraživanja i konačnog usvajanja Delaunayeve triangulacije u kombinaciji s MST-om, što je rezultiralo tamnicama koje su bile međusobno povezane i logično strukturirane. Ovaj pristup generiranju tamnica spaja strukturirani dizajn s elementima nasumičnosti, osiguravajući da je svaka iteracija tamnice jedinstvena, a ipak da je navigacija kroz tamnice jasna i zanimljiva. Jasno razgraničenje soba i hodnika poboljšava navigaciju igrača. Trenutni algoritam nudi visok stupanj prilagodbe, dopuštajući varijacije u složenosti tamnice, no ima i ovaj pristup svojih problema. Pristup zahtijeva detaljnu kalibraciju kako bi se osigurala usklađenost i koherentnost u rasporedu tamnice te ponekad se mogu dobiti tamnice gdje više izgledaju kao labirinti nego kao same tamnice.

5.2. Objašnjenje koda

Algoritam koji će biti objašnjen u ovom odjeljku strukturiran je oko nekoliko ključnih funkcija, od kojih svaka pridonosi formiranju koherentnog i zanimljivog rasporeda tamnice. Ove funkcije obuhvaćaju identifikaciju glavnih prostorija, stvaranje različitih soba, primjenu Delaunayeve triangulacije i konstrukciju minimalnog razapinjućeg stabla (MST). One zajedno rade sa generacijom koridora da bi se kreirala tamnica. Ovaj odjeljak ima za cilj razjasniti algoritam sa ciljem postizanja boljeg razumijevanja samog proceduralnog generiranja. Objašnjenja će biti za važnije metode koje se koriste.

5.2.1. Pronalaženje glavnih soba

Metoda IdentifyMainRooms ključna je u razlikovanju primarnih prostorija unutar tamnice, koje su sastavni dio rasporeda strukture. Radi na način da bira prostorije sa najvećom površinom.Funkcija sortira popis soba na temelju površine, raspoređujući ih silaznim redoslijedom. Ovo sortiranje je ključno za jednostavno prepoznavanje najvećih prostorija. Nasumično odabire broj glavnih soba, osiguravajući da se broj soba nalazi unutar danog minimalnog i maksimalnog raspona. Ova varijabilnost uvodi element nasumičnosti, povećavajući raznolikost u rasporedu tamnice. Identificiranje glavnih soba temeljno je u stvaranju tamnice. Omogućuje određivanje prioriteta značajnih područja, koja mogu poslužiti kao centralne točke tamnice, čime utječu na cjelokupni izgled i navigaciju korisnika.

```
Metoda IdentificirajGlavneProstorije(minBrojGlavnihProstorija,
maxBrojGlavnihProstorija) -> Lista GlavnihProstorija:
    // Sortiraj prostorije po površini u silaznom redoslijedu
    Sortiraj prostorije po (širina * visina) u silaznom redoslijedu
    // Nasumično odaberi broj glavnih prostorija unutar zadanog raspona
    brojGlavnihProstorija = Nasumični broj između minBrojGlavnihProstorija i
maxBrojGlavnihProstorija
    // Inicijaliziraj praznu listu za spremanje glavnih prostorija
    Inicijaliziraj praznu listu glavneProstorije
    // Iteriraj kroz sortiranu listu prostorija i dodaj prvih N prostorija u
listu glavneProstorije
    za i od 0 do min(brojGlavnihProstorija, broj prostorija) - 1:
        Dodaj prostorije[i] u glavneProstorije
    // Vrati listu glavnih prostorija
    Vrati glavneProstorije
```

Kod 5.2.1 Pseudokod metode IdentifyMainRooms

5.2.2. Generacija soba

Metoda GenerateRooms odgovorna je za stvaranje raznolikog skupa soba, od kojih svaku karakteriziraju nasumični atributi kao što su položaj, veličina i vrsta (prefab). Za svaku prostoriju koja se generira, metoda izračunava nasumični položaj unutar unaprijed definiranog raspona i nasumičnu veličinu unutar navedenih ograničenja. Kod u ovoj metodi se izvodi kroz par koraka.

1. Odredite položaj sobe: Za svaku sobu određuje se slučajni položaj unutar pravokutnog područja od (-50, -50) do (50, 50).
2. Određivanje veličine sobe: Veličina sobe se nasumično određuje unutar raspona minRoomSize i maxRoomSize.
3. Odabir prefaba sobe: Prefab sobe nasumično se odabire s popisa dostupnih prefabova
4. Instancirajte sobni objekt: Odabrana montažna soba instancirana je na određenoj poziciji s određenom veličinom.
5. Stvori instancu sobe: kreira se nova instanca sobe s postavljenim svojstvima i dodaje se na popis soba.

Ovi koraci se mogu bolje razumjeti u priloženom pseudokodu

```
Metoda GenerirajSobe:
    Za i od 0 do brojSoba - 1 učini:
        // Odredi slučajnu poziciju za sobu unutar pravokutnog područja od (-50, -
50) do (50, 50)
        pozicijaSobe = SlucajnaPozicija(-50, 50, -50, 50)
        // Odredi slučajnu veličinu za sobu unutar raspona minVelicinaSobe i
maxVelicinaSobe
        sirinaSobe = SlucajniRaspon(minVelicinaSobe.x, maxVelicinaSobe.x)
        visinaSobe = SlucajniRaspon(minVelicinaSobe.y, maxVelicinaSobe.y)
        // Odaberi slučajni prefab sobe iz liste dostupnih prefabrikata soba
        odabraniPrefabrikatSobe = SlucajniOdabir(prefabrikatiSoba)
        // Instanciraj odabrani prefab sobe na određenoj poziciji i veličini
        objektSobe = Instanciraj(odabraniPrefabrikatSobe, pozicijaSobe,
sirinaSobe, visinaSobe)
        // Stvori novu instancu Sobe s određenim svojstvima i dodaj je na popis
soba
        novaSoba = nova Soba(pozicijaSobe, sirinaSobe, visinaSobe, objektSobe)
        Dodaj novaSoba na popis soba
    Kraj Za
Kraj Procedura
```

Kod 5.1.2 Pseudokod metode GenerateRooms

Ova slučajnost osigurava varijabilnost u smještaju i dimenzijama prostorija. Prefab sobe nasumično se odabire s popisa i instancira s izračunatim atributima. Na kraju svaka instancirana soba je dodana na popis soba za daljnju obradu. Generiranje raznolikih soba ključan je korak u stvaranju privlačnih i dinamičnih okruženja te uvodi raznolikost i složenost.

5.2.3. Izvođenje Delaunay triangulacije

Metoda PerformDelaunayTriangulation primjenjuje Delaunayjevu triangulaciju na skup točaka koje predstavljaju položaje prostorija, vraćajući strukturu podataka koja predstavlja triangulaciju. Sami proces je odrađen pomoću biblioteke dostupne na GitHub-u „Computational-Geometry“ (Habrador, 2020).

```
Funkcija IzvediDelaunayTriangulaciju(Lista<Soba> sobe) vraća HalfEdgeData2
    // Kreiraj listu za pohranu točaka za triangulaciju
    Lista<MyVector2> listaTočaka = nova Lista<MyVector2>()
    // Popuni listu točaka s pozicijama soba
    Za svaku sobu u sobe
        Dodaj novu MyVector2(soba.pozicija.x, soba.pozicija.y) u listaTočaka
    // Kreiraj instancu Normalizer2
    Normalizer2 normalizator = novi Normalizer2(listaTočaka)
    // Normaliziraj točke
    Skup<MyVector2> normaliziraneTočke = normalizator.Normaliziraj(novi
Skup<MyVector2>(listaTočaka))
    // Izvedi Delaunay triangulaciju na normaliziranim točkama
    HalfEdgeData2 normaliziraniDelaunayPodaci =
_Delaunay.TočkaPoTočka(normaliziraneTočke, novi HalfEdgeData2())
    // Denormaliziraj podatke triangulacije
    HalfEdgeData2 delaunayPodaci =
normalizator.Denormaliziraj(normaliziraniDelaunayPodaci)
    // Vrati podatke triangulacije
    Vrati delaunayPodaci
Kraj Funkcije
```

Kod 5.2.3 Pseudokod metode PerformDelaunayTriangulation

Postupak se izvađa kroz tri glavna koraka:

1. Pretvorba točaka: Položaji soba pretvaraju se u popis točaka pogodnih za triangulaciju.
2. Normalizacija i triangulacija: Točke se normaliziraju na zajedničko mjerilo, a primjenjuje se Delaunayeva triangulacija. Normalizacija je ključna za osiguravanje točnosti i dosljednosti u procesu triangulacije.
3. Nenormalizacija: Podaci triangulacije su nenormalizirani kako bi se točke vratile na njihovo izvorno mjerilo, čuvajući cjelovitost položaja prostorija.

Delaunayeva triangulacija je temeljni geometrijski algoritam. Koristan je u raznim grafičkim primjenama, uključujući generiranje terena i mreže te optimalnom podjelom poligona na trokute koji se ne preklapaju. U našem kodu koristimo Delaunay triangulacija kako bi osigurali da su sve sobe povezane.

5.2.4. Izrada minimalnog razapinjućeg stabla (MST)

```
Metoda GenerirajMST(Lista<Soba> sobe, Lista<Brid> delaunayBridovi) ->
Lista<Brid>:
    // Sortiraj Delaunay bridove po težini
    Sortiraj delaunayBridovi po težini od najmanje do najveće
    // Inicijaliziraj strukturu UnionFind
    uf = Nova UnionFind(broj soba)
    // Inicijaliziraj listu bridova MST-a
    mstBridovi = Nova Lista<Brid>
    // Iteriraj kroz Delaunay bridove
    Za svaki brid u delaunayBridovi:
        // Dohvati indekse soba koje brid povezuje
        indeksSobeA = Indeks brid.sobaA u listi soba
        indeksSobeB = Indeks brid.sobaB u listi soba
        // Pokušaj unificirati podskupove koji sadrže sobe
        Ako uf.Union(indeksSobeA, indeksSobeB):
            // Dodaj brid u listu mstBridovi ako su sobe bile u različitim
podskupovima
            Dodaj brid u mstBridovi
    // Vrati listu bridova MST-a
    Vrati mstBridovi
```

Kod 5.3 Pseudokod metode GenerateMST

Minimalno razapinjuće stablo (MST) je temeljni koncept u teoriji grafova, koji se koristi za povezivanje svih vrhova u grafu s najmanjom mogućom ukupnom težinom bridova. U kontekstu ovog proceduralnog algoritma za generiranje tamnice, MST je ključan u osiguravanju da su sve sobe (vrhovi) u tamnici dostupne i međusobno povezane, dok se izbjegavaju suvišne veze. Za to se koristi metoda GenerateMST. Započinje razvrstavanjem bridova dobivenih iz Delaunay triangulacije po težini, osiguravajući da algoritam prvo uzme u obzir najkraće veze. Funkcija zatim inicijalizira podatkovnu strukturu Union-Find, koja je ključna za praćenje povezanih komponenti (podstabala) i izbjegavanje ciklusa tijekom izgradnje MST-a. Dok funkcija iterira kroz sortirane Delaunayeve bridove, koristi metodu Union strukture Union-Find kako bi utvrdila povezuje li trenutni brid dvije prethodno nepovezane komponente. Ako je tako, brid se dodaje MST-u. Ovaj proces se nastavlja sve dok se sve prostorije međusobno ne povežu najkraćim mogućim vezama, što rezultira koherentnim rasporedom tamnice. MST služi kao okosnica tamnice, osiguravajući raznolikost u putanji i rasporedu dok održava strukturalni integritet i igrivost generiranog okruženja.

5.2.5. Odvajanje soba i pozivanje ostalih metoda

Korutinska metoda SeparateRooms(Prilog 1) odgovorna je za odvajanje soba koje se preklapaju, identificiranje glavnih soba, izvođenje Delaunayeve triangulacije, generiranje grafa, stvaranje minimalnog razapinjućeg stabla (MST) s dodanim petljama i instanciranje igrača u odabranoj početnoj sobi. Metoda prvo iterativno provjerava preklapajuće sobe i odvaja ih prilagođavanjem njihovih položaja sve dok se preklapanja ne pronađu ili dok se ne postigne maksimalan broj ponavljanja. Iako sve sobe imaju 2D Collider koji pomaže odvajanju soba ova metoda dodatno osigurava da ne dođe do preklapanja. Nakon odvajanja, metoda identificira podskup glavnih prostorija pomoću IdentifyMainRooms metode i uklanja suvišne sobe sa scene. Nakon toga poziva se metoda za Delaunayjevu triangulaciju na glavnim sobama i generira se graf koji predstavlja veze između soba. MST se generira iz grafa, a dodaju se dodatne petlje za stvaranje alternativnih staza. Nakon toga se stvaraju koridori između povezanih soba, a zidovi se dodaju oko svake sobe. I na kraju bira se nasumično soba u kojoj će se igrač instancirati. Ova metoda poziva i spaja preostale metode u koherentnu cjelinu.

5.2.6. Generacija koridora

Sobe se spajaju koridorima na temelju izrađenog MST-a kojemu je dodano par ciklusa, te se koristi par metoda za osiguravanje pravilne generacije koridora.

Metoda CreateCorridors

Ova metoda(Prilog 2 Metoda CreateCorridors) iterira kroz rubove u MST-u i stvara koridore između povezanih soba. HashSetconnectedRooms inicijalizira se kako bi se pratili parovi soba koji su već povezani. Za svaki brid u mstEdges, metoda provjerava jesu li sobe već povezane. Ako jesu, preskače se na sljedeću iteraciju. Početna i krajnja točka koridora izračunavaju se pomoću metode CalculateEdgeCenter Ako su sobe poredane vodoravno ili okomito, pomoću metode DrawCorridor crta se ravni hodnik. U suprotnom, nacrtan je hodnik u obliku slova L. Metodom DetermineTurningPoint određuje se optimalna točka okretišta, a od početka do točke okretišta i od točke okretišta do kraja iscrtavaju se dva ravna koridora. Povezani par prostorija dodaje se u ConnectedRooms HashSet.

Metoda CalculateEdgeCenter (Prilog 2 Metoda CalculateEdgeCenter) izračunava središnju točku ruba koja je najbliža drugoj prostoriji. Izračunava središnje točke četiri potencijalna ruba (gornji, donji, lijevi, desni) prostorije. Zatim izračunava udaljenost od svake od tih točaka do središta druge prostorije. Metoda vraća središnju točku ruba s minimalnom udaljenošću od druge prostorije.

Metoda DetermineTurningPoint (Prilog 2 Metoda DetermineTurningPoint)određuje optimalnu točku skretanja za hodnik u obliku slova L. Izračunava dvije potencijalne okretišne točke: jednu za L-oblik koji je prvo vertikalan i jedan za L-oblik koji je horizontalni. Provjerava je li svaka točka skretanja važeća (tj. ne siječe se ni s jednom prostorijom) pomoću metode IsIntersectingRoomBuffer. Ako su obje točke skretanja valjane, odabire se ona koja rezultira kraćom ukupnom duljinom koridora. Ako vrijedi samo jedna točka skretanja, odabire tu. Ako nijedna točka skretanja nije valjana, zadana je prva.

Metoda IsIntersectingRoomBuffe(Prilog 2 Metoda IsIntersectinRoomBuffer) provjerava da li se točka skretanja ili njeni produžeci sijeku s nekom prostorijom. Iterira kroz sve sobe osim onih koje su povezane. Za svaku sobu izračunava granični okvir s međuspremnikom oko sobe. Provjerava sijeku li se točka skretanja ili njeni produžeci s graničnim okvirom bilo koje prostorije. Vraća true ako postoji presjek, a false u suprotnom.

Metoda DrawCorridors (Prilog 2 Metoda DrawCorridors) instancira segmente koridora između dvije točke. Određuje je li hodnik vodoravan ili okomit. Izračunava vrijednost koraka na temelju relativnih položaja početne i krajnje točke. Iterira od početne do krajnje točke, instancirajući segment koridora na svakom koraku. Ovaj kod učinkovito generira hodnike između prostorija. Korištenje različitih metoda za svaki zadatak čini kod modularnim i lakšim za razumijevanje, što je bitno za održavanje i proširenje funkcionalnosti u budućnosti.

5.2.7. Ostale metode

Postoji još par metoda koje iako ne igraju tako ključnu ulogu kao prijašnje objašnjene metode kod ne bi funkcionirao kako funkcionira bez njih. Neke bitnije od tih metoda bile bi metode GenerateGraphFromDelaunay i metoda AddLoopsToMST.

Metoda GenerateGraphFromDelaunay generira graf iz podataka Delaunayeve triangulacije, koji predstavlja veze između soba. Metoda pretvara trokute dobivene iz Delaunayeve

triangulacije u bridove koji predstavljaju veze između prostorija. Svaki brid povezuje dvije sobe, a rezultirajući popis bridova tvori graf.

Metoda AddLoopsToMST dodaje dodatne cikluse minimalnom razapinjućem stablu (MST) kako bi se stvorile alternativne staze između prostorija. Metoda izračunava broj ciklusa koje treba dodati na temelju broja soba i određenog omjera. Identificira potencijalne bridove ciklusa koji su dio Delaunayeve triangulacije, ali nisu u MST-u i dodaje ih .

```
Metoda DodajCikluseUMST(Lista<Soba> sobe, Lista<Brid> mstBridovi, Lista<Brid>
delaunayBridovi, int ciklusiPoSobi = 3) vraća Lista<Brid>
    // Izračunaj broj ciklusa za dodavanje
    int brojCiklusa = broj soba / ciklusiPoSobi

    // Dohvati bridove koji su u delaunayBridovi, ali nisu u mstBridovi
    Lista<Brid> potencijalniCiklusBridovi =
delaunayBridovi.Oduzmi(mstBridovi).NaListu()

    // Promiješaj potencijalniCiklusBridovi za nasumični odabir
    System.Random rng = novi System.Random()
    int n = broj potencijalniCiklusBridovi
    Dok je n > 1
        n--
        int k = rng.Sljedeći(n + 1)
        Brid vrijednost = potencijalniCiklusBridovi[k]
        potencijalniCiklusBridovi[k] = potencijalniCiklusBridovi[n]
        potencijalniCiklusBridovi[n] = vrijednost

    // Dodaj prvih 'brojCiklusa' bridova iz promiješane liste u MST
    Za i od 0 do manje od brojCiklusa i broj potencijalniCiklusBridovi
        Dodaj potencijalniCiklusBridovi[i] u mstBridovi

    Vrati mstBridovi
Kraj Funkcije
```

Kod 5.4 Pseudokod metode AddLoopsToMST

```
Metoda GenerirajGrafIzDelaunay(Lista<Soba> sobe, HalfEdgeData2 delaunayPodaci)
vraća Lista<Brid>
    Lista<Brid> bridovi = nova Lista<Brid>()
    Skup<Trokut2> trokuti =
_TransformacijaIzmeđuStrukturaPodataka.HalfEdge2NaTrokut2(delaunayPodaci)

    // Pretvori trokute u bridove
    Za svaki trokut u trokuti
        Soba sobaA = prva soba u sobe gdje je soba.pozicija jednaka novom
Vector2(trokut.p1.x, trokut.p1.y)
        Soba sobaB = prva soba u sobe gdje je soba.pozicija jednaka novom
Vector2(trokut.p2.x, trokut.p2.y)
        Soba sobaC = prva soba u sobe gdje je soba.pozicija jednaka novom
Vector2(trokut.p3.x, trokut.p3.y)

        Ako sobaA nije null i sobaB nije null, dodaj novi Brid(sobaA, sobaB) u
bridovi
        Ako sobaB nije null i sobaC nije null, dodaj novi Brid(sobaB, sobaC) u
bridovi
        Ako sobaA nije null i sobaC nije null, dodaj novi Brid(sobaA, sobaC) u
bridovi

    Vrati bridovi
Kraj Funkcije
```

Kod 5.5 Pseudokod metode GenerateGraphFromDelaunay

6. Testiranje proceduralno generiranog sadržaja

Testiranje proceduralno generiranog sadržaja (PGS) jedinstveno je izazovno zbog njegove dinamične i nepredvidive prirode. Primarni izazovi uključuju (Shaker et al., 2016):

- Varijabilnost: PGS inherentno pokazuje veliku varijabilnost, pri čemu je svaka instanca generiranog sadržaja potencijalno značajno drugačija. Ova varijabilnost postavlja izazove u uspostavljanju dosljedne osnove za testiranje.
- Ponovljivost: Raznolika priroda PGS-a čini reproduciranje specifičnih scenarija za daljnje testiranje ili otklanjanje pogrešaka složenim zadatkom.
- Pokrivenost: postizanje sveobuhvatne pokrivenosti testom je zamršeno, jer dinamičko generiranje sadržaja implicira da se netestirani scenariji mogu neprestano pojavljivati.
- Evaluacija: Procjena kvalitete i primjerenosti generiranog sadržaja često uključuje subjektivnu prosudbu, zahtijevajući ravnotežu između objektivnog automatiziranog testiranja i subjektivne ljudske procjene.

Za prevladavanje ovih izazova obično se koristi nekoliko metoda testiranja (Shaker et al., 2016):

- Automatizirano testiranje: Ovaj pristup uključuje razvoj skripti za provjeru valjanosti različitih aspekata kao što su strukturni integritet, povezanost i drugi objektivni kriteriji u različitim instancama generiranog sadržaja.
- Heuristička evaluacija: Heuristička evaluacija metoda je inspekcije upotrebljivosti pri kojoj evaluatori ispituju sučelje i procjenjuju njegovu usklađenost s priznatim načelima upotrebljivosti
- Testiranje korisnika: Prikupljanje povratnih informacija od krajnjih korisnika ključno je za dobivanje uvida u igrivost, angažman i sveukupno zadovoljstvo korisnika, što je ključno za uspjeh razvoja igre.

6.1. Provedeno testiranje

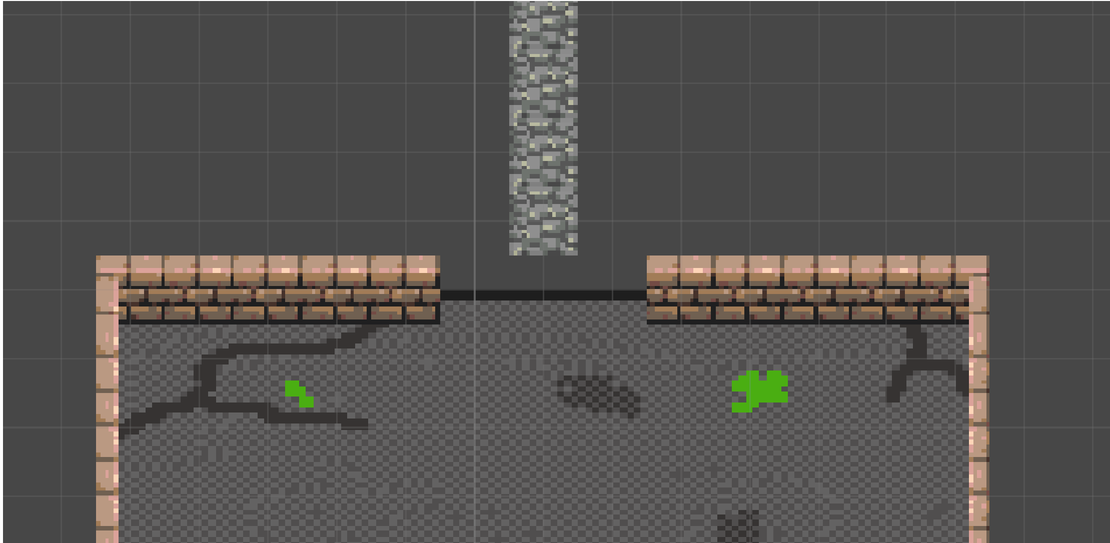
Kako bi se procijenila učinkovitost i kvaliteta proceduralno generiranih tamnica, provedena je heuristička evaluacija. Ova metoda omogućava sustavno pregledavanje korisničkog sučeljea generatora tamnice i generirani sadržaj, osiguravajući da se pridržava utvrđenih načela upotrebljivosti i normi dizajna igre. Istraženo je više instanci generiranih tamnica, sa fokuom na aspekte kao što su:

- Mogućnost igranja: Procijenjena je igrivost tamnica, osiguravajući da se njima može upravljati, da su izazovne i privlačne.
- Ravnoteža: Ispitana je ravnoteža između različitih elemenata unutar tamnica, kao što su neprijatelji, nagrade i prepreke, kako bih osigurao pravedno i ugodno iskustvo igranja.
- Estetika: Procijenjena je vizualna privlačnost i tematska dosljednost tamnica kako bih se procijenilo cjelokupno estetsko iskustvo.
- Raznolikost: Procijenjena je raznolikost i jedinstvenost generiranih tamnica kako bi se odredila sposobnost generatora da proizvodi raznolik sadržaj.

Slika 6.1 In-game screenshot igrača u sobi.

Heuristička procjena dala je uvide u prednosti i područja poboljšanja za generator tamnica. Tamnice su pokazale visoku razinu raznolikosti, nudeći jedinstveno iskustvo u svakom slučaju. Ravnoteža između različitih elemenata igre općenito je dobro održavana, pridonoseći privlačnom iskustvu igranja. Estetski elementi tamnica bili su dosljedni i iako ne toliko vizaulno privlačni pošto sam ručno crtao dizajn soba sa profesionalnim dizajnom bili bi estetski privlačni. Tijekom heurističke evaluacije uočena su dva problema. Prvi

problem povezan je sa samom navigacijom tamnica, odnosno sa generiranje hodnika, problem je riješen na način da je kod za generaciju hodnika usavršen. Problem nije potpuno uklonjen no minimiziran je njegov učinak na igru do razine gdje predstavlja zanemariv problem za igrivost. U priloženoj slici se može vidjeti jedan primjer loše generiranog hodnika, hodnikom se i dalje može ući u sobu no vizualno nije spojen sa sobom.



Slika 6.2 Primjer generirane tamnice gdje hodnik nije vizualno spojen sa sobom

Drugi uočeni problem je bio povezan sa generacijom zidova oko soba. U rijetkim slučajevima u jednoj ili dvije sobe jedan „tile“ zida nije pravilno instanciran. Po pitanju igrivosti problem nije bio značajn pošto su 2D Collider komponente bile dovoljno velike da se igrač ne može provući kroz rupu te zbog svog rijetkog pojavljivanja problem je zanemaren.

Slika 6.3 Primjer generirane tamnice gdje dva zida nisu instancirana (označeni sa crvenim)

Provođenje heurističke evaluacije bilo je ključno u identificiranju i rješavanju područja poboljšanja u generatoru tamnice. Prilagodbe napravljene kao rezultat evaluacije pridonijele su poboljšanju ukupne kvalitete i igrivosti proceduralno generiranih tamnica.

Zaključak

Tijekom ovog rada objašnjeno je ugrubo područje proceduralnog generiranja sadržaja (PGS), s posebnim fokusom na njegovu primjenu u razvoju igara. Početni odjeljci postavili su temelje za razumijevanje PGS-a, zalazeći u njegovu evoluciju, prednosti koje donosi digitalnom području i izazove koje inherentno predstavlja. Značajan dio ovog rada bio je posvećen praktičnoj primjeni ovih tehnika u igrici "Deepsea Drifter". Pružen je uvid u različite elemente igre, uključujući prirodu protivnika i algoritme korištene za implementaciju PGS-a. Nadalje, predstavljena je dubinska analiza procesa implementacije i temeljnog koda za generiranje tamnice. Razmatrani su i ocijenjeni različiti pristupi i tehnike. Zaključno, ovaj je rad ponudio temeljit pregled proceduralnog generiranja sadržaja, od njegovih teorijskih temelja do praktičnih primjena i metodologija testiranja.

Popis korištenih assetova

Cave Tile Set od korisnika FrozenTaurus

https://frozentaurus.itch.io/frozentauruss-cave-tileset

Underwater Fantasy od korisnika Ansimuz

https://assetstore.unity.com/packages/2d/textures-materials/water/underwater-fantasy-87457

Steampunk UI od tvrtke Gentleland

https://assetstore.unity.com/packages/2d/gui/icons/steampunkui-238976

Popis slika

Slika 3.1 Primjer terena generiranog pomoću Perlin Noise-a (Izvor: https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/perlin-noise-part-2/perlin-noise-terrain-mesh.html) 5

Slika 3.2 Primjer špilje/tamnice generiran pomoću Cellular Automata (Izvor: https://www.gridsagegames.com/blog/2014/06/mapgen-cellular-automata/(Pristupljeno 23.9.2023)).......... 6

Slika 3.3 Primjer generiranih planina pomoću proceduralne generacije temeljene na fraktalima (Izvor:http://alexanderpruss.blogspot.com/2017/03/super-simple-fractal-terrain-generator.html (Pristupljeno 23.9.2023)........ 7

Slika 3.4 Primjer procesa proceduralnog generiranja tamnice pomoću Random Walk algoritma (Izvor: https://bfnightly.bracketproductions.com/chapter_28.html (Pristupljeno 23.9.2023)) 7

Slika 3.5 Proceduralno generirana mapa koristeći Wang Tile-ove (Izvor: https://nothings.org/gamedev/herringbone/ (Pristupljeno 23.9.2023) 8

Slika 4.1 Prikaz izgleda neprijatelja, vozila igrača i UI-a 10

Slika 4.2 Visualizacija generiranih prostorija pomoću 2D spriteova. 12

Slika 4.3 Visualizacija Delaunay triangulacije 13

Slika 4.4 Visualizacija MST-a sa uklonjenim preostalim sobama i vraćenim ciklusima 14

Slika 4.5 Prikaz cijele generirane tamnice 15

Slika 4.6 Prikaz sobe u kojoj se igrač instancira i instance igrača vidljive na sredini sobe 15

Slika 6.1 In-game screenshot igrača u sobi. 28

Slika 6.2 Primjer generirane tamnice gdje hodnik nije vizualno spojen sa sobom 29

Slika 6.3 Primjer generirane tamnice gdje dva zida nisu instancirana (označeni sa crvenim) 29

Popis kratica

MST	*Minimum Spanning Tree*	Minimalno razapinjuće stablo
PGS	Proceduralno generiran sadržaj	
VR	*Virtual Reality*	Virtualna stvanors
AR	*Augmented Reality*	Proširena stvarnost
AI	*Artificial Intelligence*	Umjetna inteligencija

Popis kodova

Kod 5.1.2 Pseudokod metode GenerateRooms 20

Kod 5.2.3 Pseudokod metode PerformDelaunayTriangulation 21

Kod 5.3 Pseudokod metode GenerateMST 22

Kod 5.4 Pseudokod metode AddLoopsToMST 25

Kod 5.5 Pseudokod metode GenerateGraphFromDelaunay 26

Literatura

de Berg, M. C. (2008). Computational Geometry: Algorithms and Applications. Springer.

Habrador. (29. Prosinac 2020). *Computational-geometry*. Dohvaćeno iz Github: https://github.com/Habrador/Computational-geometry

Hendrikx, M. M. (19. Veljača 2013). Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications*, str. 1-22.

Russel, S. J. (2016). *Artificial intelligence: a modern approach.* Pearson Education Limited.

Togelius J., S. N. (2016). *Procedural Content Generation in Games: A Textbook and an Overview of Current Research.* Springer.

Prilog 1 Kod metode SeparateRooms

```
 private IEnumerator SeparateRooms()
    {
        bool roomsOverlapping = true;
        int maxIterations = 1500;  // Da se izbjegnu
beskonacne petlje
        int currentIteration = 0;

        while (roomsOverlapping && currentIteration <
maxIterations)
        {
            roomsOverlapping = false;
            foreach (Room roomA in rooms)
            {
                foreach (Room roomB in rooms)
                {
                    if (roomA != roomB &&
AreRoomsOverlapping(roomA, roomB))
                    {
                        roomsOverlapping = true;
                        Vector2 separationDirection =
(roomA.position - roomB.position).normalized;
                        roomA.position += separationDirection
* 1f;
                        roomB.position -= separationDirection
* 1f;
                    }
                }
            }
            currentIteration++;
            yield return null;  // Cekanje jedan frame
        }
        yield return new WaitForSeconds(10f);
        foreach (Room room in rooms)
        {
            room.position = new
Vector2(Mathf.Round(room.roomGameObject.transform.position.x)
, Mathf.Round(room.roomGameObject.transform.position.y));
```

```
            room.roomGameObject.transform.position =
room.position;
            // Zakljucavanje soba na mjestu
            Rigidbody2D rb =
room.roomGameObject.GetComponent<Rigidbody2D>();
            if (rb)
            {
                rb.constraints =
RigidbodyConstraints2D.FreezeAll;
            }
        }

        yield return new WaitForSeconds(2f);

        // Azuriranje lokacija soba
        foreach (Room room in rooms)
        {
            room.position =
room.roomGameObject.transform.position;
        }

        // Poziv metode za odabir glavnih soba
        List<Room> mainRooms = IdentifyMainRooms(10, 15);  //
Biranje 10-15 glavnih soba

        // Ukljanjanje neodabranih soba
        foreach (Room room in rooms)
        {
            if (!mainRooms.Contains(room))
            {
                Destroy(room.roomGameObject);
            }
        }

        rooms = mainRooms;


        // Delaunay triangulacija
        HalfEdgeData2 delaunayData =
PerformDelaunayTriangulation(mainRooms);
```

```
        //Visualizacija DelaunayTriangulacije. Sluzi za
vizualno debugiranje Delaunay triangulacije)
        //dungeonVisualization.VisualizeTriangulation(delauna
yData);
        List<Edge> graphEdges =
GenerateGraphFromDelaunay(mainRooms, delaunayData);
        List<Edge> mstEdges = GenerateMST(mainRooms,
graphEdges);
        List<Edge> mstEdgesWithLoops =
AddLoopsToMST(mainRooms, mstEdges, graphEdges);
        //Visualizacija MST-a, služi za vizualno debugiranje
generiranog MST-a
        //dungeonVisualization.VisualizeMST(mstEdgesWithLoops
);
        yield return new WaitForSeconds(1f);
        CreateCorridors(mstEdgesWithLoops);
        yield return new WaitForSeconds(2f);
        foreach (Room room in rooms)
        {
            AddWallsAroundRoom(room);
        }
        // 1. Izaberi sobu
        Room spawnRoom = rooms[Random.Range(0, rooms.Count)];

        // 2. Promijeni prefab odabrane sobe
        GameObject spawnRoomObj =
Instantiate(spawnRoomPrefab, spawnRoom.position,
Quaternion.identity);
        spawnRoomObj.transform.localScale = new
Vector3(spawnRoom.width, spawnRoom.height, 1);
        Destroy(spawnRoom.roomGameObject);
        spawnRoom.roomGameObject = spawnRoomObj;

        // 3. Instanciraj igraca u odabranoj sobi
        Vector3 playerPosition = new
Vector3(spawnRoom.position.x, spawnRoom.position.y, -1);
        Instantiate(playerPrefab, playerPosition,
Quaternion.identity);
    }
```

Prilog 2 Kod za generiranje hodnika

Metoda CreateCorridors

```
private void CreateCorridors(List<Edge> mstEdges)
 {
     HashSet<(Room, Room)> connectedRooms = new
HashSet<(Room, Room)>();
     foreach (Edge edge in mstEdges)
     {
         Room roomA = edge.roomA;
         Room roomB = edge.roomB;

         if (connectedRooms.Contains((roomA, roomB)) ||
connectedRooms.Contains((roomB, roomA)))
         {
             // Spojene sobe se preskacu
             continue;
         }
         Vector2 start = CalculateEdgeCenter(roomA,
roomB);
         Vector2 end = CalculateEdgeCenter(roomB, roomA);

         // Ako su sobe u ravnini horizontalno ili
vertikalno instanciraj ravni koridor
         if (start.x == end.x || start.y == end.y)
         {
             DrawCorridor(start, end);
         }
         else
         {
             // Odredivanje optimalnog skretanja
             Vector2 turnPoint =
DetermineTurningPoint(start, end, roomA, roomB);
             DrawCorridor(start, turnPoint);
             DrawCorridor(turnPoint, end);
         }
         connectedRooms.Add((roomA, roomB));
     }
 }
```

Metoda CalculateEdgeCenter

```
    private Vector2 CalculateEdgeCenter(Room room, Room 
otherRoom)
    {
        float buffer = 1f;

        Vector2 topCenter = new Vector2(room.position.x, 
room.position.y + room.height / 2 - buffer);
        Vector2 bottomCenter = new Vector2(room.position.x, 
room.position.y - room.height / 2 + buffer);
        Vector2 leftCenter = new Vector2(room.position.x - 
room.width / 2 + buffer, room.position.y);
        Vector2 rightCenter = new Vector2(room.position.x + 
room.width / 2 - buffer, room.position.y);



        // Izracun udaljenosti centra druge sobe do centra 
rubova sobe
        float topDistance = Vector2.Distance(topCenter, 
otherRoom.position);
        float bottomDistance = Vector2.Distance(bottomCenter, 
otherRoom.position);
        float leftDistance = Vector2.Distance(leftCenter, 
otherRoom.position);
        float rightDistance = Vector2.Distance(rightCenter, 
otherRoom.position);

        // Centar brida do kojeg treba najmanja udaljenost
        float minDistance = Mathf.Min(topDistance, 
bottomDistance, leftDistance, rightDistance);

        if (minDistance == topDistance) return topCenter;
        if (minDistance == bottomDistance) return 
bottomCenter;
        if (minDistance == leftDistance) return leftCenter;
        return rightCenter;
    }
```

Metoda DetermineTurningPoint

```
private Vector2 DetermineTurningPoint(Vector2 start, Vector2
end, Room roomA, Room roomB)
    {
        float buffer = 1f; // Odmak za drzanje koridora dalje
od soba

        Vector2 turnPoint1 = new Vector2(start.x, end.y);
//Prvo vertikalni
        Vector2 turnPoint2 = new Vector2(end.x, start.y);
//Prvo horizontalni

        bool isTurnPoint1Valid =
!IsIntersectingRoomBuffer(turnPoint1, start, end, roomA,
roomB, buffer);
        bool isTurnPoint2Valid =
!IsIntersectingRoomBuffer(turnPoint2, start, end, roomA,
roomB, buffer);

        if (isTurnPoint1Valid && isTurnPoint2Valid)
        {
            // Biranje tocke skretanja koja daje najkraci put
            return Vector2.Distance(start, turnPoint1) +
Vector2.Distance(turnPoint1, end) < Vector2.Distance(start,
turnPoint2) + Vector2.Distance(turnPoint2, end) ? turnPoint1
: turnPoint2;
        }
        else if (isTurnPoint1Valid)
        {
            return turnPoint1;
        }
        else if (isTurnPoint2Valid)
        {
            return turnPoint2;
        }
        else
        {
            return turnPoint1;
        }
    }
```

Metoda IsIntersectingRoomBuffer

```
    private bool IsIntersectingRoomBuffer(Vector2 turnPoint,
Vector2 start, Vector2 end, Room roomA, Room roomB, float
buffer)
    {
        foreach (Room room in rooms)
        {
            if (room != roomA && room != roomB)
            {
                Rect boundingBox = new Rect(room.position.x -
room.width / 2 - buffer, room.position.y - room.height / 2 -
buffer, room.width + 2 * buffer, room.height + 2 * buffer);
                if (boundingBox.Contains(turnPoint) ||
boundingBox.Contains(new Vector2(turnPoint.x, start.y)) ||
boundingBox.Contains(new Vector2(end.x, turnPoint.y)))
                {
                    return true;
                }
            }
        }
        return false;
    }
```

Metoda DrawCorridor

```
    private void DrawCorridor(Vector2 start, Vector2 end)
    {
        bool isHorizontal = start.y == end.y;
        int step = isHorizontal ? (end.x > start.x ? 1 : -1)
: (end.y > start.y ? 1 : -1);


        if (isHorizontal)
        {
            for (int x = (int)start.x; x != (int)end.x; x +=
step)
            {
                GameObject corridorInstance =
Instantiate(corridorPrefab, new Vector3(x, start.y, 0),
Quaternion.identity);
```

```
                corridorPrefabs.Add(corridorInstance);
            }
        }
        else
        {
            for (int y = (int)start.y; y != (int)end.y; y +=
step)
            {
                GameObject corridorInstance =
Instantiate(corridorPrefab, new Vector3(start.x, y, 0),
Quaternion.identity);
                corridorPrefabs.Add(corridorInstance);
            }
        }

        // Instanciranje zadnjeg "tile-a"
        GameObject endCorridorInstance =
Instantiate(corridorPrefab, new Vector3(end.x, end.y, 0),
Quaternion.identity);
        endCorridorInstance.GetComponent<SpriteRenderer>().co
lor = Color.red; // Bojanje zadnjeg "tile-a" u crveno
        corridorPrefabs.Add(endCorridorInstance);
    }
```