SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Sveučilišni preddiplomski studij

Pronalaženje puta u proceduralno generiranim labirintima

Završni rad

Matej Kuprešak

Osijek, 2023.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac Z1P - Obrazac za ocjenu završnog rada na preddiplomskom sveučilišnom studiju

Osijek, 10.07.2023.

Odboru za završne i diplomske ispite

Prijedlog ocjene završnog rada na preddiplomskom sveučilišnom studiju

Ime i prezime Pristupnika:	Matej Kuprešak
Studij, smjer:	Programsko inženjerstvo
Mat. br. Pristupnika, godina upisa:	R 4376, 11.10.2021.
OIB Pristupnika:	79231023445
Mentor:	izv. prof. dr. sc. Časlav Livada
Sumentor:	,
Sumentor iz tvrtke:	
Naslov završnog rada:	Pronalaženje puta u proceduralno generiranim labirintima
Znanstvena grana rada:	**Umjetna inteligencija (zn. polje računarstvo)**
Zadatak završnog rad:	U radu je potrebno opisati mehanizme proceduralnog generiranja labirinta te izazove koje ima umjetna inteligencija prilikom automatskog pronalaženja puta. Eksperiment je potrebno napraviti u Unityu. Tema rezervirana za: Matej Kuprešak
Prijedlog ocjene završnog rada:	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 2 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene od strane mentora:	10.07.2023.
Datum potvrde ocjene od strane Odbora:	24.07.2023.

Potvrda mentora o predaji konačne verzije rada:	*Mentor elektronički potpisao predaju konačne verzije.*
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O ORIGINALNOSTI RADA

Osijek, 24.07.2023.

Ime i prezime studenta:	Matej Kuprešak
Studij:	Programsko inženjerstvo
Mat. br. studenta, godina upisa:	R 4376, 11.10.2021.
Turnitin podudaranje [%]:	4

Ovom izjavom izjavljujem da je rad pod nazivom**: Pronalaženje puta u proceduralno generiranim labirintima**

izrađen pod vodstvom mentora izv. prof. dr. sc. Časlav Livada

i sumentora ,

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SADRŽAJ

1. UVOD 1

1.1. Zadatak završnog rada 1

2. PREGLED PODRUČJA TEME RADA 2

2.1. Proceduralna generacija i pronalazak puta 2

3. KORIŠTENA PROGRAMSKA OKRUŽENJA 3

3.1. Unity Game Engine 3

3.2. Visual studio 4

4. PROGRAMIRANJE I DIZAJN SIMULATORA 6

4.1. Proceduralno generiranje labirinta 6

4.2. Stvaranje igrača i izlaza iz labirinta 9

4.3. Algoritmi za pronalazak puta 11

4.3.1. Breadth-first search – pretraživanje po širini 11

4.3.2. Dijkstra 13

4.4. Kontroliranje simulatora 14

5. REZULTATI ALGORITAMA 16

5.1. Rezultati korištenja BFS algoritma 16

5.2. Rezultati korištenja Dijkstrinog algoritma 17

5.3. Usporedba rezultata 18

6. ZAKLJUČAK 20

LITERATURA 21

ŽIVOTOPIS 22

SAŽETAK 23

ABSTRACT 24

1. UVOD

Tema ovog završnog rada bit će izraditi 3D računalnu igru primjenom znanja stečenih o *Unity 3D* razvojnom okruženju. Igra se odvija iz ptičje perspektive (engl. *3D top-down view*) te je cilj same igre pronaći izlaz iz svijeta. Ovaj rad prikazat će jedan od načina proceduralne generacije labirinta, odnosno samog svijeta u kojem će se igra odvijati. Osim proceduralne generacije, cilj ovog rada bit će implementirati pronalazak puta (engl. *pathfinding*) koristeći dva od nekolicine algoritama za pronalazak puta. U ovom radu konkretno, prikazat će se rezultati pronalaska puta korištenjem algoritma pretraživanja po širini (engl. *Breadth-first search*) te tako i Dijkstrinog algoritma.

Projekt je izrađen u *Unity 3D* razvojnom okruženju koristeći verziju 2020.3.20f1. *Unity* omogućuje korisnicima sve najvažnije značajke za stvaranje igara kako u 2D-u tako i u 3D-u. Omogućuje značajke poput fizike, 3D prikaza, detekcije sudara i slično. Programski jezik korišten u ovom projektu je C#, kojim se razvojno okruženje koristi kako bi rukovalo logikom i funkcijama same igre. *Unity* pruža podršku za razvoj igara za veliki broj platformi, neke od njih su: *Mac*, *Linux*, *PlayStation*, *Xbox* te naravno *Windows*.

Interes za ovu temu proizlazi iz igranja igara koje koriste slične metode i načine stvaranja svojih svjetova. Igre kao što su *Minecraft* i *Valheim*[1] koriste se proceduralnom generacijom sličnom onoj koja će biti prikazana unutar ovog rada. Također, protivnici unutar same igre koriste algoritme pronalaska puta, kako bi pratili igrača i pronašli najbrži put do istog.

1.1. Zadatak završnog rada

U radu je potrebno prikazati načine implementacije već poznatih algoritama kao što su pretraživanje po širini, te Dijkstrin algoritam. Također, potrebno je objasniti načela korištena za generiranje samog labirinta. Na kraju, bit će prikazana izvedba praktičnog prikaza toga svega u 3D obliku te prikladnog korisničkoj sučelja (engl. *user interface*).

[1] S obzirom na spomen ovih igara, za više informacija o njima posjetiti njihove stranice: https://www.minecraft.net/en-us, https://www.valheimgame.com/

2. PREGLED PODRUČJA TEME RADA

2.1. Proceduralna generacija i pronalazak puta

Proceduralna generacija sadržaja unutar igara odnosi se na stvaranje dijelova igre s ograničenim ili neizravnim unosom korisnika. Metode proceduralne generacije nastale su iz nekolicine različitih razloga, kao što su ušteda vremena i troškova razvoja i mogućnost ponovnog igranja. Ovakav način stvaranja igara, konkretno svjetova unutar igara daje pomoć dizajnerima u smislu smanjenja vremena potrebnog za stvaranje konkretnog svijeta te im to donosi više mogućnosti za smišljanje i dodavanje raznih detalja unutar samih svjetova. Idealno, ovakvi algoritmi trebali bi biti brzi i pouzdani, što većinom i jesu. Naravno, s druge strane donose i neke negativne posljedice, počevši od teže implementacije samih algoritama, ali dolazeći i do činjenice da su takvi algoritmi zahtjevni za sklopovlje računala [1]. Kako bi generacija samog terena bila što realnija, u pravilu se koristi nekakav vid šuma (engl. *noise*). U ovom radu koristit će se Perlinov šum (engl. *Perlin noise*). Perlinov šum je pseudo-slučajni uzorak vrijednosti koje se generiraju u 2D ravnini. Iako se ova tehnika inače koristi za stvaranje 3D terena, *Unity* to ne nudi kao gotovu opciju. Pseudo-slučajni uzorak govori da šum neće dati kompletno nasumične vrijednosti, nego vrijednosti koje se sastoje od „valova“ čije se vrijednosti postupno povećavaju i smanjuju. Ovako generirani šum kasnije se koristi za stvaranje npr., karti visina [2].

Pronalazak puta unutar igara obavlja se raznim algoritmima. Strategije samog pronalaska puta imaju zadatak pronaći put između bilo koje dvije dane točke. Ovakvi algoritmi kao izlaz daju više različitih točaka, koje kada bi se spojilo, dobio bi se konkretan put od točke A do točke B. S obzirom da računalo često u tim situacijama mora obraditi velike količine informacija, algoritmi trebaju biti vrlo dobro optimizirani kako ne bi narušavali performanse, pogotovo ako se dogodi situacija da put između dvije točke ne postoji [3]. A*, Dijkstrin, te algoritam pretraživanja po širini su jedni od najbolji algoritama za pronalazak puta. Algoritmi za pronalaženje puta imaju zadatak odrediti najbrži, odnosno najkraći put između dvije točke. Svaki od ovih algoritama ima određene prednosti i nedostatke. U ovom radu će se usporediti rad Dijkstrinog i algoritma pretraživanja po širini, ne bi li se donijela odluka koji bi bio prikladniji za korištenje u ovom slučaju. Dijkstrin algoritam je algoritam pretraživanja grafa koji će dati najkraći put između dva čvora u težinskom grafu. U ovom radu će se primijeniti na graf koji predstavlja točke po kojima se može hodati u proceduralnom 3D prostoru. S druge strane, algoritam pretraživanja po širini će također dati najkraći put između dvije točke, ali na način da neće koristiti težinski graf, nego graf u kojem bridovi nemaju pridružene težine niti vrijednosti.

3. KORIŠTENA PROGRAMSKA OKRUŽENJA

3.1. Unity Game Engine

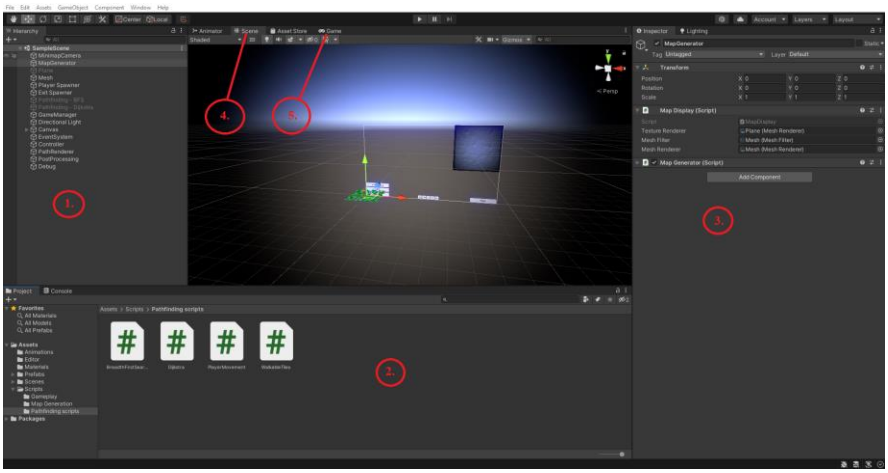
Unity je višeplatformski softver za razvoj igara koji razvija *Unity Technologies*, prvi put najavljen i izdan u lipnju 2005. godine. S vremenom je *Unity* razvio podršku za razne platforme, uključujući *desktop* platforme, mobilne, konzole ali u zadnje vrijeme i na područje virtualne stvarnosti. Kako se *Unity* smatra relativno jednostavnim i pristupačnim za početnike, vrlo je popularan u izradi igara za iOS i *Android* uređaje, te za razvoj raznih nezavisnih (engl. *indie*) igara.

Ovo razvojno okruženje daje priliku stvoriti igre u 3D svijetu kao i u 2D svijetu. Naravno, *Unity* se može koristiti za stvaranje igara u bilo kojem žanru, ali jednako tako i za stvaranje simulacija koje mogu biti korištene u nekim drugim sferama života.

Službeni programski jezik koji se koristi za razvoj unutar *Unitya* je C#. Za stvaranje igara, kao i programiranje ponašanja korištenih u programima i igrama, koristi se skriptiranje (engl. *scripting*). Skriptiranje govorima objektima unutar projekta kako će se ponašati, odnosno kako će komunicirati sa ostalim članovima same igre. Skriptiranje u *Unity-u* razlikuje se od običnog programiranja, u tome što se ovdje ne mora kreirati sami kod koji pokreće aplikaciju, nego se fokus prebacuje samo na elemente koju čime samu igru igrivom, ili simulacijske aspekte unutar nekakvog drugačijeg projekta. *Unity* pokretanje aplikacijskog koda izvodi za korisnika, dok se korisnik samo treba potruditi da kod bude optimiziran i spreman za izvođenje u što više sličica po sekundi (engl. *FPS – Frames Per Second*) [4].

Sučelje samog programa sastoji se od nekoliko glavnih elemenata. Kao što je vidljivo na slici 3.1., pod brojem 1 nalazi se hijerarhija (engl. *hierarchy*) unutar koje se može vidjeti sve objekte koji se trenutno nalaze u sceni. Desnim klikom unutar hijerarhije također mogu se dodavati novi elementi u samu scenu. Pod brojem 2 vidi se projektni prozor. On daje uvid u sve datoteke koje se nalaze u projektu na disku. Sadržava apsolutno sve potrebno i korišteno, od tekstura, modela, pa sve do skripti i audio zapisa. Broj 3 označava inspektor. Inspektor u *Unity-u* označava prozor koji daje mogućnost upravljanja nad objektom, upravljanja u smislu određivanje njegove pozicije, rotacije, razmjera (*Transform* komponenta), upravljanje ostalim skriptama koje su dodane na njega i slično. Za kraj, prikazani su prozori pod brojevima 4 i 5. Prozor pod brojem 4 prikazuje zapravo prostor kojim se može upravljati i koji je dan na korištenje za stvaranje samog svijeta za sami projekt. Unutar njega postavljaju se pozicije, rotacije i svi ostali funkcionalni i nefunkcionalni elementi potrebni za rad igre. Nakon što je sve uspješno posloženo, rad se može

pregledati unutar prozora broj 5 pod imenom „Game“ koji zapravo prikazuje realnu sliku kako će se sve prikazivati prilikom pokretanja samog projekta, kako u samom sučelju, tako i u stvarnom izrađenom projektu.



Sl. 3.1. Prikaz Unity 3D sučelja

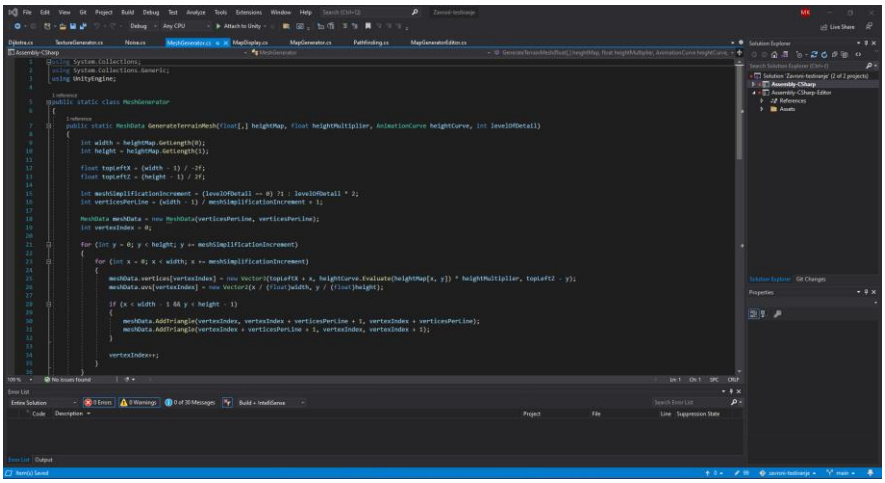
3.2. Visual studio

Visual studio je integrirano razvojno okruženje (engl. *IDE – Integrated development environment*) koje se koristi za razvoj različitih računalnih programa kao što su web stranice, web aplikacije, web servisi a ujedno i mobilnih aplikacija. U ovom okruženju može se koristiti nekoliko drugačijih razvojnih platformi, npr. *Windows API, Windows Forms, Windows Presentation Foundation* i druge. *Studio* uključuje i editor programskog koda koji podržava *IntelliSense*, te također i refaktoriranje koda. *IntelliSense* je komponenta koja omogućava završavanje koda, u smislu dopune koda koji korisnik započne pisati.

Postoji nekoliko različitih verzija *Visual Studia*, to su *Community, Professional* te *Enterprise*. *Community* verzija je besplatna, što zapravo čini ovaj editor vrlo raširenim među programerima. Slogan iza *Community* verzije je „Free, fully-featured IDE for students, open-source and individual developers“, što u prijevodu kaže: „Besplatno potpuno opremljeno razvojno okruženje za studente, *open-source* i individualne programere.“ [5].

Visual studio podržava 36 različitih programskih jezika te dozvoljava editoru da podrži i skoro pa svaki programski jezik. Neki od unaprijed ugrađenih jezika su C, C++, C++/CLI, Visual Basic, C# i drugi. Od spomenutih jezika, dolazimo i do jezika korištenog unutar ovog rada koji je C#.

Na slici 3.2. vidi se prikaz samog sučelja unutar jedne skripte i primjer pisanja koda unutar samog C# programskog jezika. C# je objektno orijentiran programski jezik, što je jedno od mogućih pristupa programiranju. Unutar objektno orijentiranog programiranja, naglasak je na stvaranju aplikacija kao skupa objekata (na objektima se zasniva cijeli pristup), za razliku od načina kojima je težište na radnjama koje se vrše na podatkovnim strukturama. Osnovni koncepti objektno orijentiranog programiranja su: korištenje objekata (objekti su osnovna jedinica objektno orijentiranog programirana), enkapsulacija (odnosi se na skrivanje unutarnjih detalja objekta od ostatka programa – privatna svojstva i atributi), apstrakcija (definiraju se apstrakcije ili sučelja koja definiraju zajedničke karakteristike i funkcionalnosti objekata), nasljeđivanje (kreiranje novih klasa temeljenih na već postojećim), te na kraju polimorfizam (predstavlja sposobnost objekta da se ponaša na različite načine ovisno o kontekstu u kojem se koristi).



Sl. 3.2. Prikaz Visual Studio sučelja

4. PROGRAMIRANJE I DIZAJN SIMULATORA

Unutar ovog poglavlja bit će opisane teorijske osnove samih korištenih tehnika, te će se prikazati način na koji su te iste tehnike implementirane unutar samog rada za izvedbu zadatka rada koji je zadan.

4.1. Proceduralno generiranje labirinta

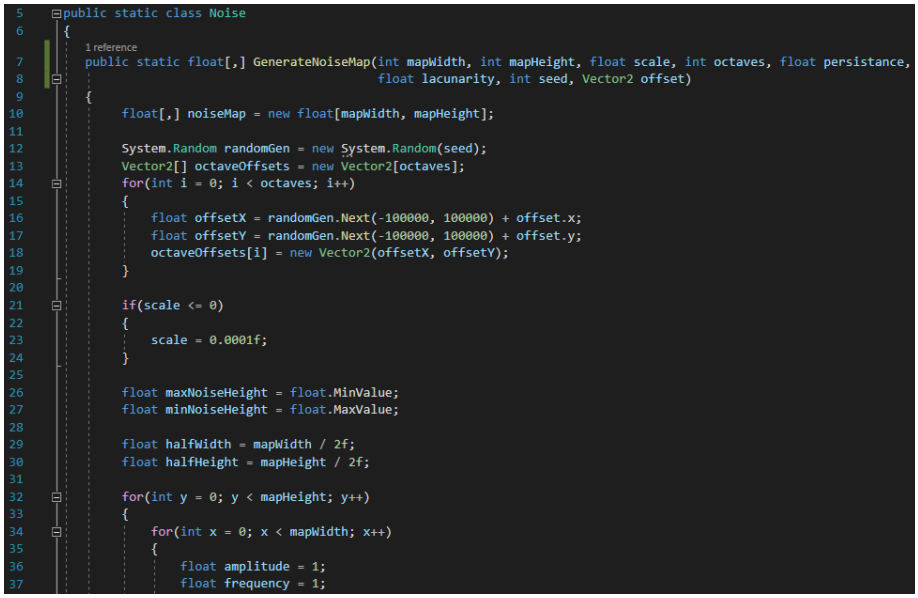
Proceduralno generiranje labirinta, odnosno samog terena koji će se koristiti za pronalazak puta temeljit će se na ovim principima:

- Perlinov šum (*engl. Perlin noise*)
- Visinska mapa (*engl. Height map*)
- Karta boja (*engl. Color map*)
- Razina detalja (*engl. Level of detail – LOD*)

Ideju iza Perlinovog šuma imao je Ken Perlin, koji je htio pronaći način generacije trodimenzionalnih tekstura na proizvoljne objekte bez projiciranja tih objekata na dvodimenzionalne površine. [6] Perlinov šum može se definirati za bilo koji broj dimenzija. Implementacija najčešće uključuje sljedeća tri koraka:

- Definiranje mreže vektora
- Izračunavanje skalarnog produkta vektora
- Interpolacija između vrijednosti

Naime, Perlinov šum unutar ovog rada implementiran je na sljedeći način. Kao što je vidljivo na slici 4.1., unutar klase *Noise*, postoji metoda *GenerateNoiseMap*. Klasa *Noise* je statična jer neće biti potrebno više instanci same klase. Metoda *GenerateNoiseMap* nakon svog izvođenja daje 2D polje vrijednosti između 0 i 1, koje će se iskoristiti za generiranje samog terena. Nakon postavljanja pomaka za oktave (*engl. octave offset*), ono omogućuje da promjenom vrijednosti samog vektora pomaka, se „prolazi“ kroz taj generirani šum (ograničen je na vrijednosti do -10000, do 10000 jer je testiranjem utvrđeno da se na većim vrijednostima krene čudno ponašati). Razmjeru (*engl. scale*) se naravno ne smije dopustiti da bude 0, te iz tog razloga ako dođe do pokušaja postavljanja u 0, postavlja se na neku minimalnu vrijednost kako ne bi dovelo do grešaka, npr. 0.0001 u ovom slučaju.



```
public static class Noise
{
    1 reference
    public static float[,] GenerateNoiseMap(int mapWidth, int mapHeight, float scale, int octaves, float persistance,
                                            float lacunarity, int seed, Vector2 offset)
    {
        float[,] noiseMap = new float[mapWidth, mapHeight];

        System.Random randomGen = new System.Random(seed);
        Vector2[] octaveOffsets = new Vector2[octaves];
        for(int i = 0; i < octaves; i++)
        {
            float offsetX = randomGen.Next(-100000, 100000) + offset.x;
            float offsetY = randomGen.Next(-100000, 100000) + offset.y;
            octaveOffsets[i] = new Vector2(offsetX, offsetY);
        }

        if(scale <= 0)
        {
            scale = 0.0001f;
        }

        float maxNoiseHeight = float.MinValue;
        float minNoiseHeight = float.MaxValue;

        float halfWidth = mapWidth / 2f;
        float halfHeight = mapHeight / 2f;

        for(int y = 0; y < mapHeight; y++)
        {
            for(int x = 0; x < mapWidth; x++)
            {
                float amplitude = 1;
                float frequency = 1;
```

Sl. 4.1. Prvi dio klase Noise

Prolaskom kroz cijelu kartu (*engl. map*) određuju se vrijednosti Perlinovog šuma za svaku točku unutar karte. Vrijednost Perlinovog šuma množi se sa 2 te mu se oduzima 1, kako bi se dobile vrijednosti između -1 i 1 (ranije navedena „Interpolacija između vrijednosti“). To je potrebno kako bi se moglo dobiti i smanjenje i povišenje vrijednosti visine unutar samog terena kasnije.

Nakon cijelog procesa (ostatak vidljiv na slici 4.2.), potrebno je samo normalizirati vrijednosti šuma kako bi se vrijednosti vratile na originalne vrijednosti koje daje funkcija *Mathf.PerlinNoise*, odnosno kako bi vrijednosti bile unutar skupa [0,1]. Važno je napomenuti da se amplituda množi sa postojanosti kako bi se smanjivale njene vrijednosti za svaku oktavu, dok se frekvencija množi sa lakunarnosti (kontraudubljenje terena) kako bi se povećavala frekvencija. Ukratko, taj postupak simulira učinak dodavanja slojeva šuma više frekvencije i niže amplitude, te laički rečeno, omogućuje da se generiraju tereni koji su ugodniji oku promatrača, te realniji stvarnom okolišu. Ranije navedeno izračunavanje skalarnog produkta vektora obavlja sama funkcija *Mathf.PerlinNoise.*

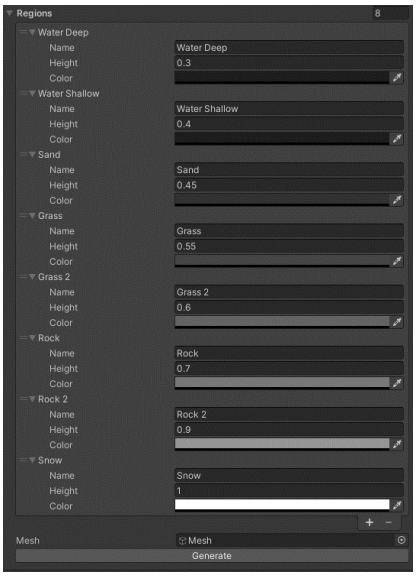
```
                float frequency = 1;
                float noiseHeight = 0;

                for(int i = 0; i <octaves; i++)
                {
                    float sampleX = (x - halfWidth) / scale * frequency + octaveOffsets[i].x;
                    float sampleY = (y - halfHeight) / scale * frequency + octaveOffsets[i].y;

                    //da bude od -1 do 1
                    float perlinValue = Mathf.PerlinNoise(sampleX, sampleY) * 2 - 1;

                    noiseHeight += perlinValue * amplitude;

                    amplitude *= persistance;
                    frequency *= lacunarity;
                }

                if(noiseHeight > maxNoiseHeight)
                {
                    maxNoiseHeight = noiseHeight;
                }
                else if(noiseHeight < minNoiseHeight)
                {
                    minNoiseHeight = noiseHeight;
                }
                noiseMap[x, y] = noiseHeight;
            }
        }

        for (int y = 0; y < mapHeight; y++)
        {
            for (int x = 0; x < mapWidth; x++)
            {
                noiseMap[x, y] = Mathf.InverseLerp(minNoiseHeight, maxNoiseHeight, noiseMap[x, y]);
            }
        }

        return noiseMap;
    }
```

Sl. 4.2. Drugi dio klase Noise

Visinska mapa u računalnoj grafici je zapravo rasterska slika (*engl. raster image*) koja se pretežno koristi kao „mreža“ za stvaranje, odnosno modeliranje visina, u ovom slučaju, konkretno pri stvaranju labirinta. Ovakav način generiranja visina temelji se zapravo na Perlinovom šumu definiranom ranije. Prema bojama generiranim samim šumom teren se izdiže odnosno spušta. Šum daje polje vrijednosti [0,1] na temelju kojega generiramo samu visinu te nakon toga i boje samog terena, odnosno regije. Za samu simulaciju visina, unutar koda koriste se petlje kako bi stvorili „vrh“ za svaku točku unutar visinske mape. Te točke se množe sa multiplikatorom visine (*engl. Height multiplier*) te se evaluiraju koristeći visinsku krivulju koja je također definirana unutar *Unity Editora*.

U ovome radu kartu boja dobiva se zapravo iz visinske mape. Iz unaprijed određenih regija samog labirinta, odnosno određenih visina unutar kojih se nalazi neka od regija, generator teksture dobit će 2D polje boja iz kojeg će generirati same teksture potrebne za teren. Detaljnije, boje se dobiju tako što se iterira kroz svaku točku iz dobivenog 2D polja i uspoređuje vrijednost visine te točke sa vrijednostima visina definiranim u polju *TerrainType* (definira regije za korištenje). Ovisno o visini, dodjeljuje se odgovarajuća boja. Za primjer određivanja regija i samim time boja

pogledati sliku 4.3. Ručno se dodaju regije po vlastitom odabiru. U ovom primjeru dodano je 8 regija te su podijeljene smisleno.



Sl. 4.3. Prikaz određivanja regija

Visine moraju biti postavljen u intervalu [0,1], te su i boje sukladno tome postavljene kako bi sam teren imao što smisleniji prikaz. Struktura za regije zahtijeva za svaku regiju da joj se da ime, radi preglednosti, iza toga zahtijeva visinu terena te se na kraju može postaviti željena boja. Radi lakšeg korištenja dodana je na kraju tipka *Generate* (vidljivo također na slici 4.3.).

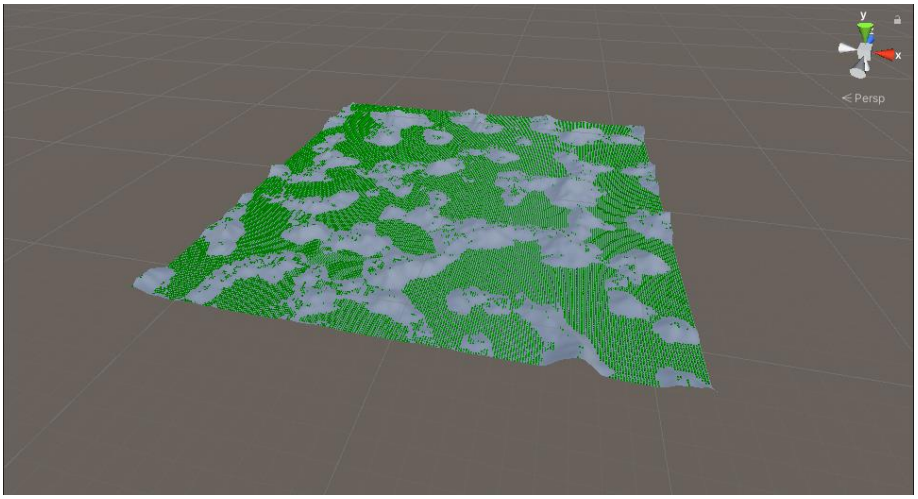
Razina detalja je tehnika koja smanjuje broj operacija koje *Unity* zahtjeva od GPU-a (*engl. Graphical Processing Unit*) da obavi za prikazivanje udaljenih dijelova labirinta. Ova tehnika omogućuje *Unity-u* da smanji broj „trokuta“ koje treba prikazivati za određene objekte unutar scene na temelju udaljenosti od kamere. [7]

4.2. Stvaranje igrača i izlaza iz labirinta

Stvaranje, odnosno postavljanje pozicije igrača i izlaza iz samog labirinta obavlja se na isti način, s razlikom u tome da se izlaz stvara nakon igrača te da se mora stvoriti na unaprijed određenoj minimalnoj udaljenosti od igrača. Biranje pozicija za stvaranje oba „objekta“ temelji se na skripti *WalkableTiles*. Uz pomoć skripte *WalkableTiles* dobiju se, kako samo ime kaže, svi

dijelovi terena na kojima igrač ima mogućnost hodati. U nastavku će detaljnije biti objašnjena implementacija gore navedenoga.

Za bolje razumijevanje kako se odvija stvaranje igrača i izlaza, uvid daje slika 4.4.



Sl. 4.4. Prikaz mjesta gdje igrač može hodati

Naime zeleno što se vidi na slici je samo grafički prikaz koji nestaje pokretanjem samog simulatora. Zelene kocke označavaju mjesta na kojima je nagib samog terena dovoljno mali kako bi igrač po tim samim mjestima mogao hodati, odnosno na kojima teren nema veliku strminu za „penjati“ se. Skripta *WalkableTiles* osim što će dati ovakav vizualni prikaz metodom *DrawWalkableTiles*, dat će također i samu listu vrijednosti metodom *GetWalkableTiles*. Lista vrijednosti predstavlja listu koja se sastoji od vektora koji predstavljaju pozicije na terenu po kojima igrač može hodati. Svaka od tih vrijednosti predstavlja svojevrsni vektor.

Nadalje, unutar skripti *PlayerSpawner* i *ExitSpawner* koristi se instanca skripte *WalkableTiles* za dohvaćanje podatka o pozicijama koje ona skladišti. Igrač će se stvoriti na nasumično odabranoj poziciji iz liste svih mogućih pozicija. S druge strane, pri stvaranju izlaza mora se uzeti u obzir činjenica da se izlaz i igrača želi stvoriti na određenoj minimalnoj udaljenosti jedno od drugoga. Takva funkcionalnost postiže se na način da se stvori još jedna lista koja sadrži samo vrijednosti pozicija koje su dovoljno udaljene od igrača (proizvoljno određena udaljenost). Nakon toga odvija se isti postupak kao i kod stvaranja igrača, samo što se sad uzima u obzir nova lista dovoljno udaljenih pozicija iz koje se nasumično uzima jedna za stvaranje izlaza.

4.3. Algoritmi za pronalazak puta

Pronalazak puta kao metoda, koristila se i koristi se u mnogim područjima, počevši od igara pa do autonomne vožnje. Cilj samog pronalaženja puta koristeći neke od najpoznatiji algoritama je zapravo dati najkraću i najučinkovitiju rutu za nekakav „objekt“ da dođe do svojeg odredišta. Unutar industrije igara, pronalazak puta se najčešće koristi za likove koji nisu igrivi (*engl. NPC – Non Playable Character*), ili unutar nekih strateških igara u stvarnom vremenu, za pomicanje entiteta od jedne točke do točke gdje je korisnik kliknuo mišem (pronalazak puta između trenutne pozicije i pozicije klika miša). [8] Neke od najpoznatijih metoda korištene danas su:

- A*
- Theta*
- Dijkstra
- Pretraživanje po širini (BFS)

Za potrebe ovog rada, implementirat će se i pokazati algoritam pretraživanja po širini te Dijkstrin algoritam, koristeći ponovno skriptu *WalkableTiles* kako bi se dobile sve točke koje se mogu uzeti u obzir pronalaska samog puta od igrača do izlaza.

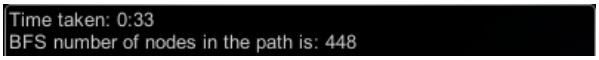
4.3.1. Breadth-first search – pretraživanje po širini

Ovaj algoritam zapravo radi na način da uzima početni čvor (u ovom slučaju poziciju igrača) te ispituje sve neposredne susjedne čvorove, zatim sve čvorove dva „koraka“ daleko, i tako dalje sve dok ne pronađe ciljni čvor (zapravo poziciju izlaza iz labirinta koju smo generirali ranije). [9] Bitno je naglasiti da pregledava baš sve čvorove na svakoj dubini stabla. Složenost algoritma pretraživanja po širini u veliko O notaciji (određuje vremensku složenost na jednom računalu, prednost je što će imati istu složenost na većini drugih računala što omogućava poopćavanje same složenosti) je:

$$O(|V| + |E|) \tag{4-1}$$

U nastavku će biti objašnjena implementacija samog algoritma unutar ovog projekta. Naime, algoritam je implementiran po gore navedenom objašnjenju. Koristeći skriptu *WalkableTiles* dobit će se sve točke koje treba uzeti u obzir sami algoritam. Algoritam će inicijalizirati red (u skripti „*queue“*) koji će sadržavati sve točke koje treba istražiti te uz to i *hash set* koji će čuvati vrijednosti točaka koje su već posjećene. Uz to, inicijalizirat će se i rječnik (*engl. dictionary*) koja će bilježiti prethodnu točku za svaku točku na putu. Izvođenje će započeti s prvom

točkom (pozicijom igrača), stavit će ju u red „*queue*“ te označiti kao posjećenu. Zatim se uzimaju u obzir sve točke u redu i provjerava se jesu li susjedi već posjećeni, ako nisu, dodaju se u red i označavaju kao posjećeni. Algoritam također bilježi za svakog susjeda da je došao iz trenutne točke (spomenuti rječnik vrijednosti). Po definiciji BFS algoritma prolazi se prvo kroz sve točke unutar jednog reda, odnosno na jednoj dubini stabla, pa se tek onda može ići dalje na sljedeći red, odnosno sljedeću dubinu. Ovaj algoritam ponavljat će se sve dok se ne provjere sve točke ili dok se ne pronađe krajnja točka, u ovom slučaju izlaz iz labirinta koji je prethodno generiran. Ukoliko algoritam pronađe izlaz, koristi metodu *GeneratePath* koja će konstruirati listu pozicija koja predstavlja najkraći put od igračeve pozicije do pozicije izlaza. U konzoli će to izgledati ovako, slika 4.5.



Sl. 4.5. Prikaz rezultata BFS algoritma na konkretnom slučaju

Uz to, unutar samog *Inspectora* u *Unityu*, na objektu na kojem se nalazi ova skripta, bit će dodane sve točke u listu te će im se moći vidjeti pozicije (vizualno prikazano, u smislu prikazivanja broja vektora koje će vratiti sami algoritam). Prikaz metode *FindPath* koja zapravo obavlja BFS algoritam nalazi se na slici 4.6.

```
81      private List<Vector3> FindPath(Vector3 start, Vector3 end)
82      {
83          cameFrom = new Dictionary<Vector3, Vector3>();
84          queue = new Queue<Vector3>();
85          visited = new HashSet<Vector3>();
86
87          queue.Enqueue(start);
88          visited.Add(start);
89
90          while (queue.Count > 0)
91          {
92              Vector3 current = queue.Dequeue();
93
94              if (current == end)
95              {
96                  return GeneratePath(start, end);
97              }
98
99              foreach (Vector3 neighbor in GetNeighbors(current))
100             {
101                 if (!visited.Contains(neighbor))
102                 {
103                     queue.Enqueue(neighbor);
104                     visited.Add(neighbor);
105                     cameFrom[neighbor] = current;
106                 }
107             }
108         }
109
110         return null;
111     }
```

Sl. 4.6. Metoda *FindPath* unutar BFS algoritma

4.3.2. Dijkstra

Dijkstrin algoritam također pronalazi najkraći put između dvije točke, pozicije na labirintu (pozicije igrača i pozicije izlaza) koristeći skriptu WalkableTiles. Ovaj algoritam oslanja se na „težine“ rubova, odnosno „težine“ puteva do ostalih čvorova unutar grafa. Algoritam pokušava pronaći put koji će imati najmanju ukupnu „težinu“, odnosno pokušat će dati najmanju ukupnu udaljenost između dvije točke koje su mu predane. Složenost Dijkstrinog algoritma u veliko O notaciji je:

$$O(V^2)$$

U nastavku će biti opisana programska implementacija samog algoritma unutar simulatora. Slika 4.7. prikazuje metodu *FindPath* ovoga puta implementiranu za potrebe Dijkstrinog algoritma. Kao i kod BFS algoritma, ova skripta prvo preuzima podatke koji su joj potrebni iz skripte *WalkableTiles* te pronalazi objekte igrača i izlaza te skladišti i njihove pozicije. Potrebne informacije traže se unutar metode *GetPathList.* Podaci iz *WalkableTiles* su naravno samo sve moguće pozicije po kojima igrač može hodati. Nakon toga pokreće se Dijkstrin algoritam odnosno poziva se metoda *FindPath* sa slike 4.7.

```
75    private List<Vector3> FindPath(Vector3 start, Vector3 end)
76    {
77        distances = new Dictionary<Vector3, float>();
78        cameFrom = new Dictionary<Vector3, Vector3>();
79        visited = new HashSet<Vector3>();
80
81        foreach (Vector3 tilePos in walkableTilePositions)
82        {
83            distances[tilePos] = Mathf.Infinity;
84        }
85
86        distances[start] = 0;
87        cameFrom[start] = start;
88
89        while (visited.Count < walkableTilePositions.Count)
90        {
91            Vector3 current = GetNextVertex();
92            visited.Add(current);
93
94            if (current == end)
95            {
96                return GeneratePath(start, end);
97            }
98
99            foreach (Vector3 neighbor in GetNeighbors(current))
100           {
101               float distanceToNeighbor = Vector3.Distance(current, neighbor);
102               float tentativeDistance = distances[current] + distanceToNeighbor;
103
104               if (tentativeDistance < distances[neighbor])
105               {
106                   distances[neighbor] = tentativeDistance;
107                   cameFrom[neighbor] = current;
108               }
109           }
110       }
111
112       return null;
113   }
```

Sl. 4.7. Metoda *FindPath* unutar Dijkstrinog algoritma

Zatim se inicijaliziraju potrebne varijable (*distances*, *cameFrom*, *visited*) i postavljaju se početne udaljenosti svih točaka na beskonačnost osim početne pozicije, naravno, njoj se postavlja udaljenost na 0. Algoritam će se ponavljati iterativno sve dok postoje točke koje nisu posjećene ili dok ne dođe do ciljne točke (pozicije izlaza iz labirinta). To će omogućiti metoda *GetNextVertex*, ona je metoda koja za zadatak ima pronaći sljedeću pločicu s najmanjom udaljenošću koja još nije posjećena, odnosno nije dodana u listu *visited*. Kada su sve pozicije posjećene i obrađene, metoda *GetPathList* provjerava je li put koji je generiran prazan, te ovisno o tome ispisuje poruku. Uz pripadajuću poruku, ova skripta ispisat će i vrijeme potrebno za njeno izvođenje u konzolu. U konzoli će to izgledati ovako, slika 4.8.

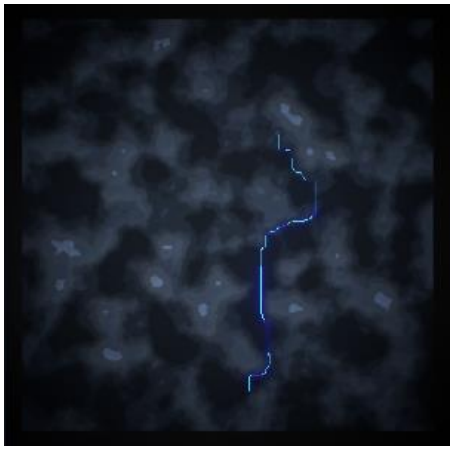
Sl. 4.8. Prikaz rezultata Dijkstrinog algoritma na konkretnom slučaju

4.4. Kontroliranje simulatora

Samo korisničko sučelje za upravljanje simulatorom je jednostavno. Na slici 4.9 najbolje je vidljiv raspored samih kontrola. Sučelje se sastoji sa lijeve strane od 3 tipke koje pokreću same algoritme, te pokreću kretanje igrača po točkama koje mu algoritmi daju. Također, u gornjem lijevom kutu nalazi se konzola koja se inače nalazi u samom sučelju *Unitya*, kao nekakav dodatak za lakše praćenje rezultata samih algoritama. Sa desne strane nalazi se tipka koja resetira cijelu scenu na početak, kako bi se moglo ponovno testirati. Te za kraj na sredini ekrana nalaze se tipke koje će ubrzati kretanje igrača kako čekanje da igrač dođe do izlaza ne bi bilo predugo, ili kako bi ga se moglo usporiti i bolje vidjeti kako se kreće.

Sl. 4.9. Prikaz korisničkog sučelja

Nakon pritiska na tipku *Start running* igrač će krenuti trčati prema točkama u pronađenom putu. Radi boljeg uvida u njegov put i njegovo kretanje implementiran je trag (*engl. trail*) koji prati igrača sve dok ne dođe do cilja. U gornjem desnom kutu nalazi se minimapa koja daje uvid u poziciju samog igrača i izlaza. Trag je također vidljiv na spomenutoj minimapi te to u izvedbi izgleda ovako, vidljivo na slici 4.10.



Sl. 4.10. Prikaz traga koji igrač ostavlja

5. REZULTATI ALGORITAMA

U ovom poglavlju bit će prikazani te uspoređeni rezultati koje daju ranije implementirani algoritmi. Iz prikazanih rezultata, moći će se zaključiti koji od ovih algoritama radi bolje u određenim situacijama i koji bi bilo prikladnije koristiti za kakve probleme. Testiranje će se provesti na velikim količinama podataka, radi uvida u to kako će se algoritmi ponašati, hoće li moći dati rezultate te kakvi će oni sami biti.

5.1. Rezultati korištenja BFS algoritma

Rezultati BFS algoritma prikazani ovdje, testirani su na osobnom računalu poprilične snage (procesor *AMD Ryzen 5 3600x*, grafička kartica *NVIDIA GeForce RTX 2060*, 32gb RAM memorije) te se iz tog razloga mogu uzeti u obzir pri biranju algoritama za implementaciju u osobne potrebe. U nastavku će se zaključiti uspješnost algoritma na temelju rezultata prikazanih u tablici 5.1.

Commented [ČL1]: Navesti specifikacije računala.

Tablica 5.1. Rezultati BFS algoritma

BFS algoritam						
Primjer	**Broj točaka**	***Seed***	**Pronađen put**	**Broj točaka puta**	**Vrijeme (min)**	**Strmina terena**
1	29606	400	Ne	0	0:16	15
2	32270	420	Da	228	0:23	15
3	28308	450	Ne	0	0:23	15
4	38902	475	Da	187	0:29	15
5	27993	1000	Ne	0	0:06	15
6	34053	825	Da	226	0:30	15
7	34128	950	Da	270	0:28	15
8	26269	225	Da	225	0:13	15
9	30720	100	Da	213	0:22	15
10	33869	567	Ne	0	0:36	15

Kroz različiti raspon točaka testirana je brzina pronalaska puta, ukoliko put u tim uvjetima postoji naravno. Neki slučajevi rezultirali su time da puta između igrača i izlaza nema, nekoliko čimbenika se tu moraju uzeti u obzir. Kao glavni čimbenik navodi se strmina terena. Naime strmina terena odlučuje koliko se igrač može „penjati“ u visinu te iz toga proizlazi, ukoliko je teren brdovit, postoji mogućnost da igrač neće moći doći do izlaza jer neće imati mogućnost prijeći „brda“. Testirane su različite nasumične vrijednosti *seeda* radi uvida u različite terene, ravninske, brdske i slično. Uz strminu, drugi najbitniji čimbenik je naravno pseudo-slučajna generacija pozicija

igrača i izlaza. Kako nad tim simulator nema kontrolu, ne može uvijek garantirati da će postojati put između igrača i izlaza. Kako je ranije navedeno, igrač i izlaz su na unaprijed određenoj minimalnoj udaljenosti, u ovom slučaju je to *150f*. U tablici je vidljivo da su vremena izvođenja algoritma mala (ne prelazi 36 sekundi) što su dobri rezultati. Naravno, rezultati su prihvatljivi u smislu samog simulatora, za korištenje u realnim primjerima trebalo bi iskoristiti neke od načina optimizacije radi još bržeg izvođenja algoritma.

5.2. Rezultati korištenja Dijkstrinog algoritma

Rezultati Dijkstrinog algoritma testirani su na istom, gore navedenom, računalu te će kasnije biti uspoređeni sa rezultatima drugog algoritma korištenog, algoritma BFS. U nastavku će se prikazati rezultati u tablici 5.2., na uzor prethodno pokazanih rezultata za BFS algoritam.

Tablica 5.2. Rezultati Dijkstrinog algoritma

Dijkstrin algoritam						
Primjer	**Broj točaka**	***Seed***	**Pronađen put**	**Broj točaka puta**	**Vrijeme (min)**	**Strmina terena**
1	28917	330	Da	265	0:37	15
2	35814	250	Da	219	1:16	15
3	32579	125	Da	209	0:58	15
4	37318	430	Da	234	1:32	15
5	26612	3011	X	0	X	15
6	27723	750	X	0	X	15
7	32462	756	Da	179	0:16	10
8	25598	100	Da	217	0:37	10
9	30012	2212	Da	237	0:32	10
10	25903	630	Da	185	0:25	10

Kao i kod BFS algoritma, Dijkstrin algoritam testiran je na velikom broju podataka, odnosno točaka koje su označene kao da pružaju mogućnost igraču da hoda preko njih (*WalkableTiles*). Rezultati koji su u tablici označeni sa X nisu dali rezultate, odnosno vrijeme čekanja je bilo predugačko te je dovelo do toga da se *Unity* smrzne te je iz tog razloga nakon njih strmina terena smanjena na 10 kako bi se smanjio pritisak na samo računalo te se dobili prihvatljivi rezultati. Kao i gore navedeno, rezultati pokazuju da su najbitniji čimbenici broj točaka te strmina terena. Većim brojem točaka vrijeme čekanja je u pravilu išlo prema gore. U nekim slučajevima unatoč većem broju točaka, algoritam je dao vrlo dobre rezultate, odnosno niska vremena potrebna

za pronalazak puta. Pozadina toga jeste ponovno pseudo-slučajna generacija pozicija igrača i izlaza. Minimalna udaljenost ručno postavljena nije se mijenjala te ona nije utjecala na rezultate. Rezultati primjene ovog algoritma dosta variraju, uzimajući u obzir pseudo-slučajne pozicije, ne može se predvidjeti kako će točno reagirati u određenim situacijama. U usporedbi sa BFS algoritmom, daje lošije rezultate, odnosno zahtijeva više vremena za pronalazak puta, a u par slučajeva i nije dao nikakav rezultat.

5.3. Usporedba rezultata

Kako bi se mogao izvući zaključak te doći do mogućnosti korištenja samih algoritama izvan ovog rada, za kraj usporedit će se rezultati oba algoritma. Sama usporedba dati će uvid u to koji algoritam će se bolje ponašati sa određenom količinom podataka te različitim situacijama u kojima će se naći s obzirom na pozicije igrača i izlaza. Usporedba je vidljiva unutar tablice 5.3.

Tablica 5.3. Usporedba rezultata algoritama

Usporedba algoritama						
Primjer	**Broj točaka**	***Seed***	**Pronađen put**	**Broj točaka puta**	**Vrijeme (min)**	**Strmina terena**
1 - BFS	30720	100	Da	227	0:25	15
1 - D	30720	100	Da	227	1:06	15
2 - BFS	28890	700	Da	257	0:13	15
2 - D	28890	700	Da	257	0:27	15
3 - BFS	36780	867	Da	219	0:36	15
3 - D	36780	867	Da	219	1:41	15
4 - BFS	34925	211	Da	243	0:34	15
4 - D	34925	211	Da	243	1:34	15
5 - BFS	27893	1007	Da	591	0:25	15
5 - D	27893	1007	Da	591	1:05	15

Primjeri pod istim rednim brojem testirani su u potpuno istim okolnostima. Pod istim okolnostima podrazumijeva se:

- Ista pozicija igrača i izlaza
- Jednak broj točaka
- Jednak *seed*
- Jednaka strmina terena

Nasumičnim odabirom vrijednosti *seeda* slučajno su došli rezultati koji svi daju pozitivan ishod, odnosno svi su uspjeli pronaći put. Zbog istih početnih situacija u kojima su se našli algoritmi, odnosno igrač i izlaz, broj točaka je jednak u svim ishodima algoritama. Kao što je vidljivo u tablici 5.3. svaki rezultat Dijkstrinog algoritma zahtijevao je barem duplo više vremena za svoju izvedbu. Razlog tome može biti udaljenost početne i krajnje pozicije, jer ako su pozicija igrača i izlaza blizu jedna drugoj, BFS algoritam će se obaviti u manje koraka jer Dijkstra može „proširiti" pretragu i na pozicije koje su znatno udaljene od cilja. Iz tog istog razloga, Dijkstrin algoritam zapravo ima više čvorova za obraditi u usporedbi sa BFS-om. S obzirom kako je Dijsktrin algoritam implementiran (koristeći težinski graf) on bi trebao uvijek vratiti najkraći put između dvije točke koje mu se predaju kao argumenti. S druge strane, BFS nije implementiran na način da koristi težinski graf, te se iz tog razloga ne može garantirati da će dati najkraći put u svakom pokušaju testiranja. Za kraj, još jedan od ključnih utjecaja bio bi taj što BFS inače brže radi u grafovima koji su jednostavni i u kojima nema različitih težina bridova jer BFS algoritam ne mora pratiti i ažurirati težine bridova, za razliku od Dijkstre.

6. ZAKLJUČAK

Unutar ovoga rada kreiran je simulator koji pruža stvaranje različitih proceduralno generiranih labrinata te prilagodbu svojstva samog terena prema korisnikovim željama. Od regija, boja samih regija, pa sve do visine terena dano je korisniku na biranje i određivanje. Problemi na koje se nailazi u pokušaju implementacije ovakvog sustava su uspješno obrađeni.

Objašnjeni su principi nekih od poznatijih načina izvedbe ovakvog problema, za koje je predstavljeno i jednostavno korisničko sučelje kako bi se korisniku olakšalo testiranje samih algoritama korištenih. Napravljeni su i neki vizualni efekti radi lakšeg uvida u rezultate samih algoritama.

Na temelju rezultata dobivenih samim testiranjem algoritama, donosi se zaključak, da ako je u pitanju struktura grafa koja je jednostavna i ne sadrži različite težine bridova, prikladnije bi bilo koristiti BFS algoritam. Naravno, u stvarnoj implementaciji bilo kojeg od ova dva algoritma, u njihovu implementaciju bilo bi potrebno dodati neke od načina optimizacije kako bi se „brže“ dobilo rezultate te kako bi oni sami bili pouzdaniji. Moglo bi se npr. optimizirati skupove podataka koje je potrebno analizirati algoritmima (provesti neke od postupaka analize skupova podataka), koristiti heuristiku (pomaže algoritmu da pretražuje čvorove koji vjerojatno vode prema cilju, što može ubrzati pronalaženje puta).

Commented [ČL2]: Ovo prebaciti u poglavlje Zaključak I prilagoditi s već napisanim tekstom

Ovaj simulator, te njegov kod, odnosno implementacija algoritama mogu se koristiti pretežno u području game developmenta u smislu pronalaženja puta za objekte odnosno likove koji nemaju mogućnost kontrole od strane igrača, ili pronalaženja puta u strategijama stvarnog vremena.

LITERATURA

[1] Shaker, Noor, Julian Togelius, and Mark J. Nelson. "Procedural content generation in games." (2016): 978-3. [8.6.2023]

[2] https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html [8.6.2023]

[3] Graham, Ross, Hugh McCabe, and Stephen Sheridan. "Pathfinding in computer games." *The ITB Journal* 4.2 (2003): 6. [8.6.2023]

[4] https://unity.com/how-to/learning-c-sharp-unity-beginners#what-languages-can-you-use-unity--2 [8.6.2023]

[5] https://visualstudio.microsoft.com/ [8.6.2023]

[6] Tatarinov, Andrei. "Perlin noise in real-time computer graphics." *GraphiCon*. 2008. [25.6.2023]

[7] https://docs.unity3d.com/Manual/LevelOfDetail.html [26.6.2023]

[8] Pardede, Sara Lutami, et al. "A Review of Pathfinding in Game Development." *[CEPAT] Journal of Computer Engineering: Progress, Application and Technology* 1.01 (2022): 47-56. [26.6.2023]

[9] Stout, Bryan. "Smart moves: Intelligent pathfinding." *Game developer magazine* 10 (1996): 28-35. [26.6.2023.]

[10] Rafiq, Abdul, Tuty Asmawaty Abdul Kadir, and Siti Normaziah Ihsan. "Pathfinding Algorithms in game development." *IOP Conference Series: Materials Science and Engineering*. Vol. 769. No. 1. IOP Publishing, 2020. [26.6.2023]

ŽIVOTOPIS

Matej Kuprešak je rođen 10. srpnja 2000. godine u Slavonskom Brodu te je prve godine života proveo u Županji. Završio je Osnovnu školu „Mate Lovraka“ u Županji te nakon toga upisao Prirodoslovno-matematičku gimnaziju također u Županji. Nakon gimnazije, upisuje preddiplomski studij računarstva na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku.

SAŽETAK

Naslov: **Pronalazak puta u proceduralno generiranim labirintima**

U ovome radu napravljen je simulator za prikaz jednog od načina proceduralne generacije labirinta. Uz proceduralnu generaciju napravljen je i sustav za testiranje algoritama za pronalazak puta. Proceduralna generacija implementirana je na temeljima Perlinovog šuma (*engl. Perlin noise*), sami šum generira „boje“ terena te se iz toga nastavlja na izdizanje samog terena, ali se dodaju i konkretne boje kako bi teren bio što realniji. Kontroliranje ovih svojstava ponuđeno je unutar *Unity* sučelja. Za pronalaženje puta korišteni su Dijkstrin algoritam te algoritam pretraživanja po širini. Implementirani su na način da koriste podatke dobivene iz druge klase pod imenom *WalkableTiles* te koristeći znanje dobiveno o pozicijama po kojima se može kretati, ispisuju put od jedne točke do druge ukoliko takav put postoji. Cijeli sustav ukomponiran je u jednostavno sučelje za korištenje pomoću kojega možemo testirati algoritme te vidjeti njihove rezultate (broj točaka, vrijeme trajanja). Uz to postoji i grafički prikaz radi lakšeg praćenja puta kojeg će igrač prijeći. Rezultati prikazani mogu ukazati na to koji bi algoritam bilo prikladnije koristiti u određenim situacijama.

Ključne riječi: algoritam pretraživanja po širini, Dijkstrin algoritam, Perlinov šum, proceduralna generacija, pronalazak puta

ABSTRACT

Title: **Pathfinding in procedurally generated mazes**

This graduate paper shows the development of a simulator which presents the implementation of procedural generation. A system meant for testing some of the pathfinding algorithms was made, along with the procedural generation. Procedural generation is implemented on the basis of Perlin noise, where the noise itself generates the „colors“ of the terrain, and from that the system continues to raise up the terrain and gives it proper colors so that the terrain looks more realistic. The control of these properties is offered within the Unity interface. The two algorithms used for pathfinding were Dijkstra's and breadth-first search algorithm. They are implemented in a way so that they use the data obtained from another class titled WalkableTiles. By using the given information about positions they are able to find and show the path from one point to another in case the path exists. The whole system is integrated in a simple user interface with which the algorithms may be tested and their results can be displayed (number of nodes in the path, pathfinding duration). In addition, there is also a graphic display for easier monitoring of the path that the player will move along. The results shown can indicate which algorithm would be more appropriate to use in certain situations.

Key words: breadth-first search algorithm, Dijkstra's algorithm, Perlin noise, procedural generation, pathfinding.