Proceduralno generiranje tamnica u računalnim igrama

Garmaz, Filip

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU

FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Sveučilišni studij

PROCEDURALNO GENERIRANJE TAMNICA U RAČUNALNIM IGRAMA

Završni rad

Filip Garmaz

Osijek, lipanj 2019.

Sadržaj

1. UVOD

Izrada video igra često je vrlo zahtjevan proces, kako vremenski tako i u smislu uloženog rada i vještine. Komponenta u izradi video igara koja ima važnu ulogu je dizajniranje razina *(engl. level design).* Dizajn razina zahtjeva ne samo matematičku ili fizičku teoretsku podlogu, nego i programerske i umjetničke vještine modeliranja i dizajniranja razina, tako da oni izgledaju lijepo, budu dojmljivi i privlačni budućim igračima *(engl. gamers)*, koji će u konačnici i kupiti igru. Drugim riječima sveukupni uspjeh igre dobrim dijelom ovisi o estetici, koja je većim dijelom sam dizajn razina.

Međutim, u većini slučajeva razine se stvaraju ručno i to zahtjeva puno rada, vještine i vremena. To predstavlja veliki izazov novim developerima igara jer oni su obično programeri ili inženjeri s malim budžetom, koji si ne mogu priuštiti profesionalne umjetnike i dizajnere razina *(engl. level designers)*.

Ipak, postoji pristup da programatski i nasumično, uz zadana pravila i uvjete računalo samo stvara razine. Takav pristup zahtjeva minimalan ručni rad sa modeliranjem razina i može poštedjeti developere rada i vremena. Glavna prednost ovakvog pristupa je nasumičnost, tj. svaki put kada algoritam stvara novu razinu, ta razina nikada neće biti identična prethodno stvorenim razinama. To omogućava veću dinamiku igre i veću vrijednost ponovnog igranja *(engl. replayability)*, jer svaki put kada se započne nova igra razine će biti potpuno nove i različite od prošle sesije igranja. Ovakvi algoritmi se najčešće koriste u tzv. Dungeon Crawler žanrovima igra. Cilj ovoga rada je napraviti algoritam koji će nasumično stvoriti razinu, odnosno tamnicu *(engl. dungeon)* te omogućiti razumijevanje u funkcionalnost algoritma. Tamnica se sastoji od različitih hodnika ili soba različitih oblika i veličina, za koje će se dalje koristiti naziv ćelija. ćelije predstavljaju jedinični dio tamnice i služe kao građevinski blokovi, čija kombinacija može proizvesti nebrojivo mnogo oblika tamnice.

1.1. Zadatak završnog rada

Zadatak završnog rada je osmisliti algoritam za automatsko i nasumično stvaranje tamnica. Za testiranje nasumično stvorenih tamnica koristi se Unity 2019, a za izradu algoritma koristi se programski jezik C#, koji je ugrađen u Unity, te može sa lakoćom koristiti Unityjeve API-je. Za izradu 3D modela ćelija koristi se Blender v2.79b.

2. IZRADA 3D MODELA ĆELIJA U BLENDERU

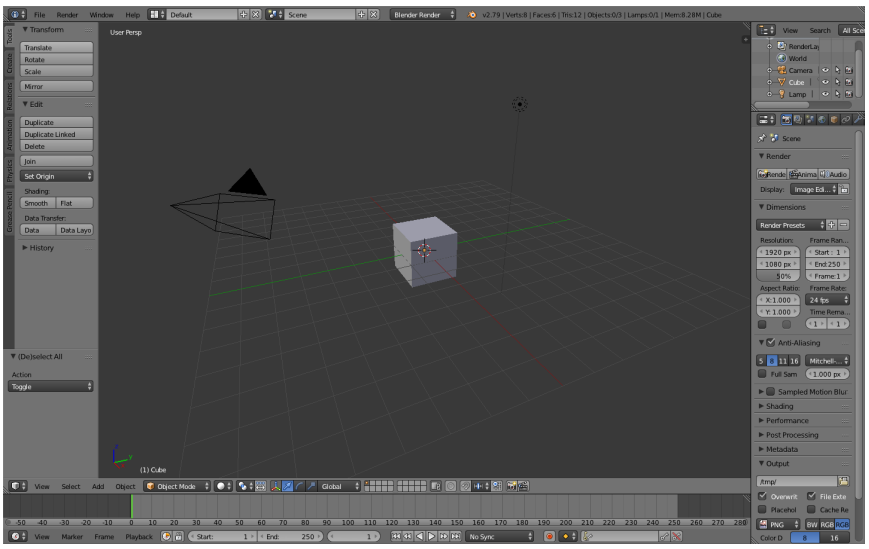
2.1. Uvod u Blender

Blender je besplatni i open source programski paket za izradu, simulaciju, vizualizaciju i modeliranje 3D objekata. Odlikuje se bogatstvom značajki, koje služe za modeliranje, manipulaciju, razne simulacije fizike, animiranje 3D objekata i još mnogo toga, pa čak može poslužiti i kao game engine za izradu računalnih igra, preko programskog jezika Python, koji je ugrađen u Blender. Ipak što se tiče izrade igra, Blender kao game engine se nije pokazao najboljim izborom, pogotovo ne uz Unity, koji je mnogo napredniji i jednostavniji za korištenje u odnosu na Blender. Blender se isključivo koristio za izradu vizualnih 3D modela ćelija, koje će nadalje poslužiti u izradi tamnice u Unity engineu.

2.2. Sučelje Blendera

Prilikom otvaranja Blendera, pojavit će se karakteristično Blender sučelje sa kockom, izvorom svjetlosti i kamerom, kao glavni 3D modeli u sceni. U većini slučajeva nisu potrebni, te se pomoću tipke a popraćene sa tipkom x lako obrišu.

Također je vrijedno napomenuti neke korisne značajke Blenderovog sučelja. Jedna od njih je opcija „Add”. Pomoću koje se mogu dodavati osnovni modeli, od kojih je nadalje moguće izraditi složenije 3D modele. Na primjer, odabirom opcije „Add” → „Mesh” → „Cube” stvorit će se 3D model kocke, kao na početku pokretanja Blendera. Tu kocku je moguće manipulirati i oblikovati tako da nastane bilo koji oblik, no tu je samo mašta granica. Manipulacija izgledom objekta se postiže korištenjem moda „Edit Mode”. Jednostavno desnim klikom se označi objekt kocke i pomoću tipke tab ulazi se u Edit Mode, također pomoću iste tipke se izlazi Edit Modea nazad u Object Mode.

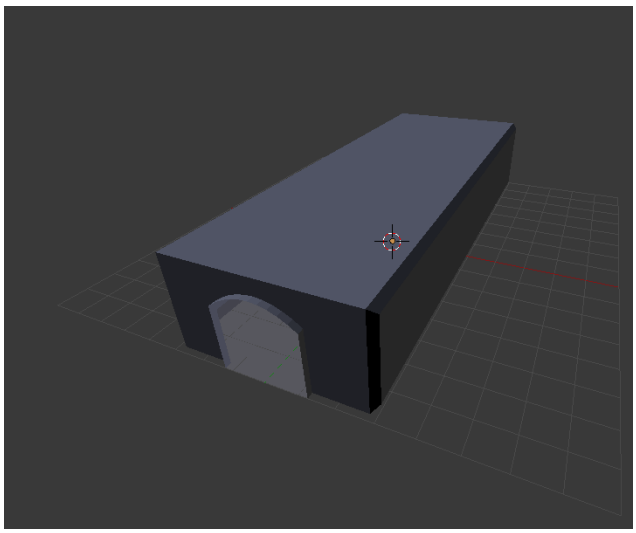
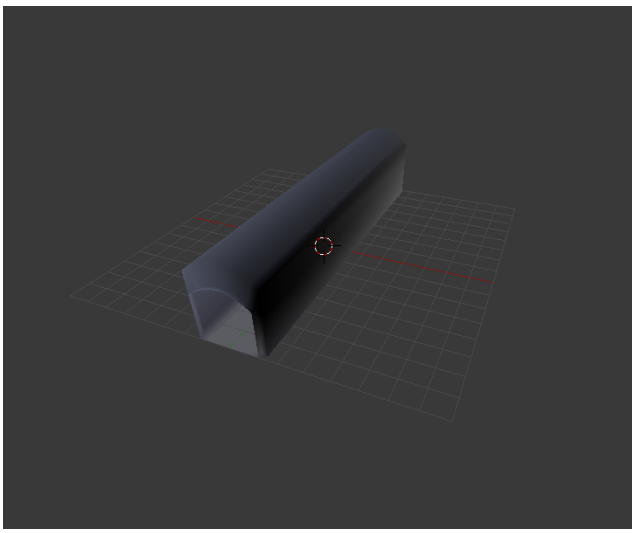


Sl. 2.1. Karakteristično sučelje Blendera.

2.3. Izrada modela ćelija

Prije same izrade modela ćelija, vrlo su važne dimenzije ćelija. Sve ćelije moraju imati svoje otvore na granicama svoje ograničavajuće kutije. Ograničavajuća kutija *(engl. Bounding box)* je zamišljeni prostor oko 3D modela i opisuje gdje se svaka točka 3D modela nalazi u globalnom prostoru u odnosu na tu ograničavajuću kutiju. Veličina ograničavajuće kutije može biti proizvoljna, no jednom kada se postavi sve ćelije moraju poštivati granice ograničavajuće kutije. Za primjer uzelo se za veličinu ograničavajuće kutije 16 m. Ovo je vrlo važno u algoritmu kod stvaranja tamnice, jer algoritam koristi tzv. snap units. Snap unit je najmanja moguća mjera ili pomak na koju se može postaviti ćelija, znači svakih 16 k metara $k \in \mathbb{Z}$, bilo u x ili y smjeru. Kako su otvori ćelija točno na granicama svog ograničavajuće kutije, to znači da će izlaz ćelije ujedno biti ulaz u susjednu ćeliju i tamnica će tada biti zatvorena. Više o tome će biti raspravljeno u sljedećim poglavljima.

Što se tiče visine i oblika ćelije, tu je poželjna umjetnička sloboda. Moguće je izraditi više različitih oblika ćelija sa različitim detaljima, te imati bogatu raznolikost varijanta ćelija kod stvaranja.

Sl. 2.2. a) Ravna ćelija, dimenzija 16m x 6m x 3m. Otvor ćelije je točno na granici ograničavajuće kutije. **b)** Druga varijanta ćelije, dimenzija 16m x 3m x 4m. Bez obzira koliko je oblik različit, otvor je na granici ograničavajuće kutije.

Sada kada su dimenzije i granice jasne, može se započeti izrađivati model. Ovdje će biti ukratko objašnjeno kako u Blenderu modelirati jednostavni 3D oblik.

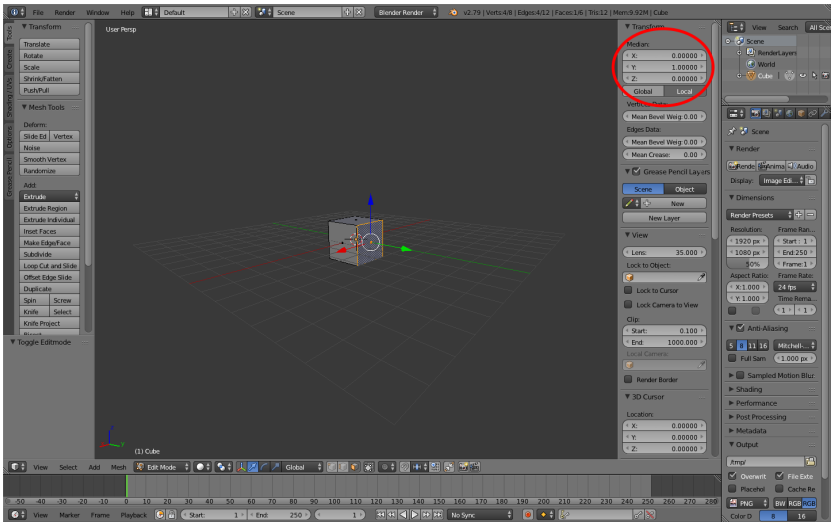
Najprije treba stvoriti novi objekt, kojem će se izmijeniti izgled, tako da izgleda kao ćelija. Pomoću opije „Add” → „Mesh” → „Cube” stvorit će se kocka u praznom prostoru.

Za manipuliranje izgledom objekta pomoću desnog klika označimo objekt te pritiskom tipke tab ulazimo u „Edit Mode”. U Edit Modeu moguće je desnim klikom označiti lica, bridove pa čak i točke, koje sačinjavaju taj objekt, odnosno njegov mesh.

Nakon toga potrebno je lijevim klikom najdesniji gumb:

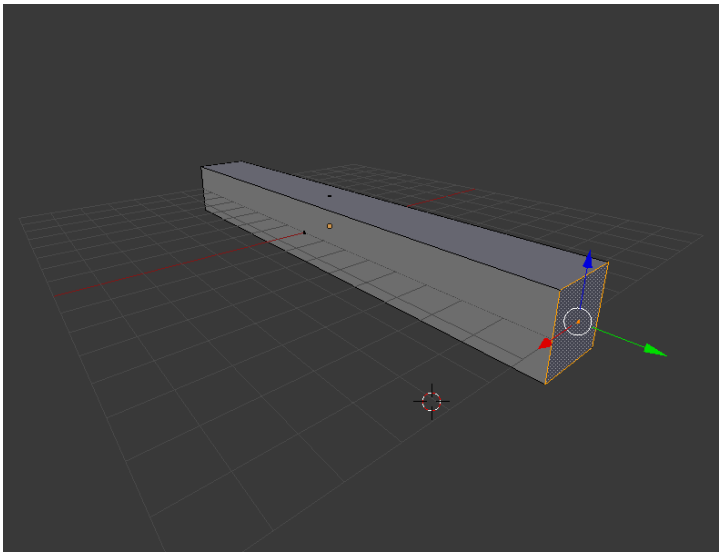
Sl. 2.3. Gumb za označavanje lica objekta u Edit Modeu.

Kako je y-os u Blenderu definirana kao smjer naprijed i nazad, označimo lice kocke na zelenoj y-osi. Pomoću tipke n moguće je prikazati ploču sa svojstvima ćelije sa koordinatama tog lica, na desnoj strani ekrana („Property Panel”).



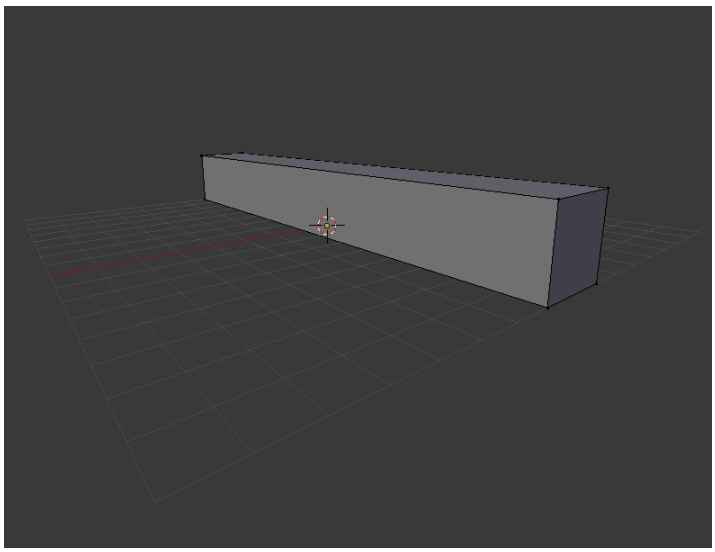
Sl. 2.4. Prikaz pozicija označenog lica na „Property Panelu”.

Stranu kocke okrenutu prema negativnom smjeru y-osi možemo produžiti i to do -8 m. To isto možemo učiniti i sa stranom okrenutom u pozitivnom smjeru y-osi, samo produžimo do +8 m.



Sl. 2.5. Izgled 3D modela nakon produženja stranica po y-osi.

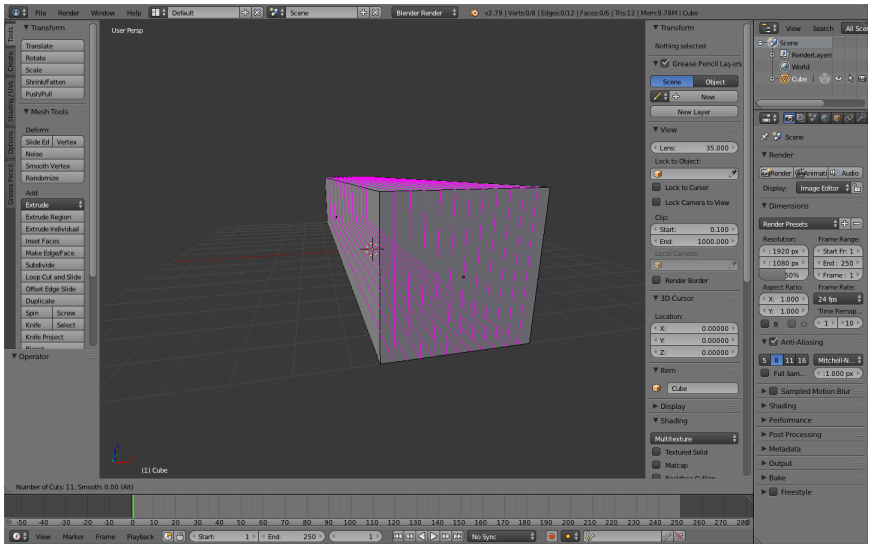
Sa slike se može primijetiti da je dno (najdonje lice) naše ćelije ispod poda ograničavajuće kutije, što nije poželjno. Treba premjestiti prema gore dok ne dodiruje pod. Da bi se to učinilo, potrebno je pomoću tipke a označiti sva lica ćelije, te pritisnuti tipku g za pomicanje te ćelije. Zatim treba pritisnuti tipku z tako da se ćelija pomiče samo po z-osi. Nakon toga možemo slobodno upisati broj koji će pomaknuti ćeliju za točno taj upisani broj. Budući da je ćelija bila kocka visine 2 m, pomaknuta -1m u svoje središte, znači da treba je pomaknuti za +1m tako da dira pod.



Sl. 2.6. 3D model ćelije pomaknut tako da mu dno dira pod.

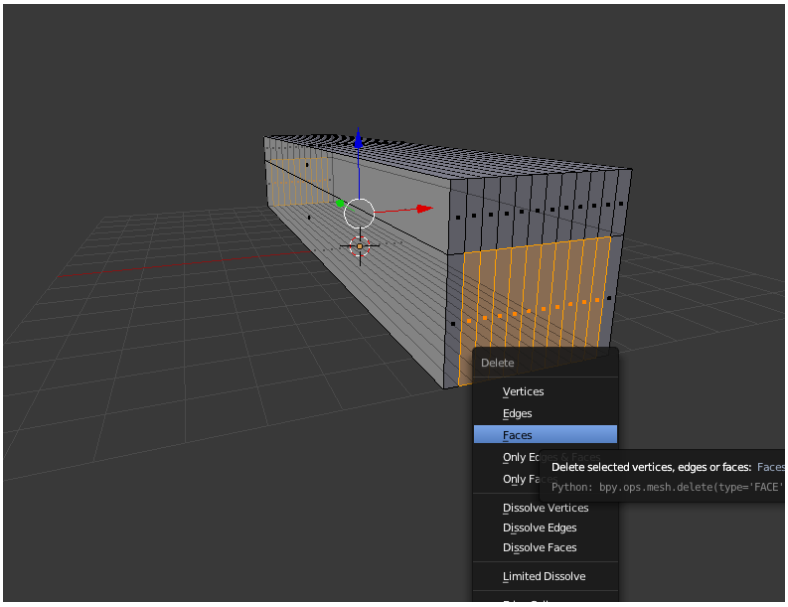
Također moguće je podesiti i širinu i visinu ćelije. Jednostavno sa desnim klikom označi se odgovarajuće lice te se povuče u odgovarajućem smjeru po odgovarajućoj osi. Na primjer, označimo najgornje lice te pomoću Property Panela izmijenimo poziciju lica na 2 na 3 metra.

Nakon što smo zadovoljni sa trenutnom veličinom ćelije, potrebno je napraviti otvore ćelija. To se može postići naredbom „Loop Cut", koju pozivamo kombinacijom tipki ctrl i r. Loop Cut će podijeliti lica ćelije na jednak broj proizvoljno mnogo djelova. Nakon pritiska kombinacija tipki za Loop Cut kotačićem miša možemo odabrati na koliko dijelova želimo podijeliti ćeliju. U projektu se podesio Loop Cut na 11, što znači da se ćeliju podijelilo na 11 djelova, odnosno to su dodatnih 12 lica.



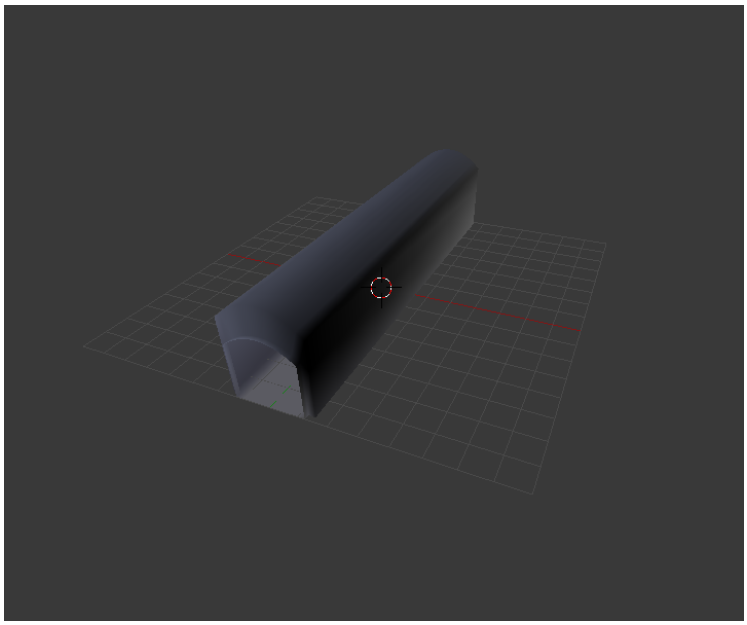
Sl. 2.7. Loop Cut u akciji.

Nakon izvršenja naredbe Loop Cut, modelu ćelije se može napraviti otvor. Potrebno je označiti novonastala lica sa prednje i stražnje strane, te pritiskom tipke x obrisati ta lica, kao na slici 2.8.



Sl. 2.8. Stvaranje otvora na ćeliji brisanjem lica prethodno stvorena naredbom Loop Cut.

Sada imamo funkcionalnu ćeliju. Ipak, ta ćelija izgleda poput kvadra, vrlo jednostavno, bez detalja i jednobojno. Ovdje vrlo veliku ulogu ima iskustvo sa radom u Blenderu i kreativnost. Pomicanjem točaka, lica ili bridova, ćeliji se može dati složeniji oblik, koji ljepše izgleda.



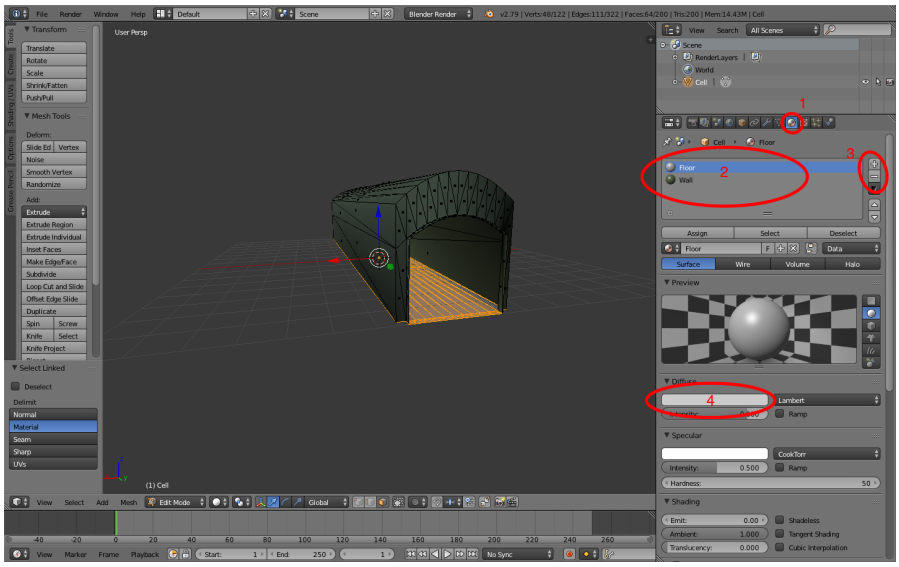
Sl. 2.9. Oblik ćelije nakon dorade na estetici.

2.4. Izrada materijala i tekstura ćelije

Sada ćelija ima lijepi oblik, međutim jednobojna je. Potrebno je dodati teksturu. Model će izgledati detaljnije i ljepše, što model ima više materijala i tekstura, te što je ta tekstura detaljnija.

Najprije je dobro dodati modelu materijale, materijali služe da se model podjeli na dijelove sa različitim svojstvima. Na primjer ćelija može imati drveni pod, a zidove od kamena, zato je dobra praksa da ta ćelija ima dva materijala. Materijali također mogu poslužiti da model ima više tekstura, tj. za svaki materijal možemo postaviti različitu teksturu.

Kako bi ćelija imala teksture, najprije je potrebno dodati materijale. Materijal se može dodati tako da se dođe na izbornik svojstava objekta, koji se nalazi skroz na desnoj strani sučelja, te klikom na ikonu sa materijalima. Nakon toga klikom na gumb + može se napraviti novi materijal i može mu se dodati boja, pomoću koje se razlikuju dijelovi ćelije. Bitno je naglasiti da tijekom dodavanja novog materijala potrebno je označiti sva ona lica ćelija na koji se želi primijeniti materijal. Materijal se može primijeniti klikom na gumb „Assign”.



Sl. 2.10. Primjena materijala na lica modela ćelije (1 – Izbornik za materijale, 2 – Lista svih materijala za trenutni model, 3 – Dodaj / ukloni materijal, 4 – Boja materijala).

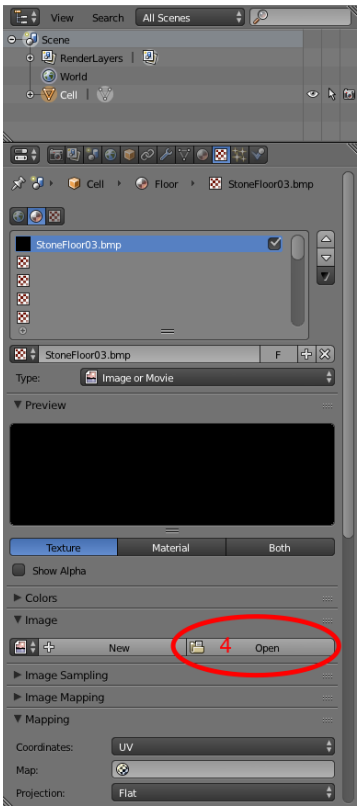
Da bi se dodala tekstura potrebna je slika teksture, te 2D projekcija lica ćelije na tu teksturu. Proces 2D projekcije lica modela na teksturu naziva se UV Unwrap.

Ovo su teksture koje su korištene u projektu:

Sl. 2.11. Teksture koje su korištene za izgled ćelija. Sve teksture su izvučene iz igre „The Elder Scrolls V: Skyrim" te pripadaju firmi Bethesda Softworks LLC, sva prava pridržana. Ove teksture su korištene isključivo za demonstraciju projekta i ne koriste se u nikakve komercijalne svrhe.

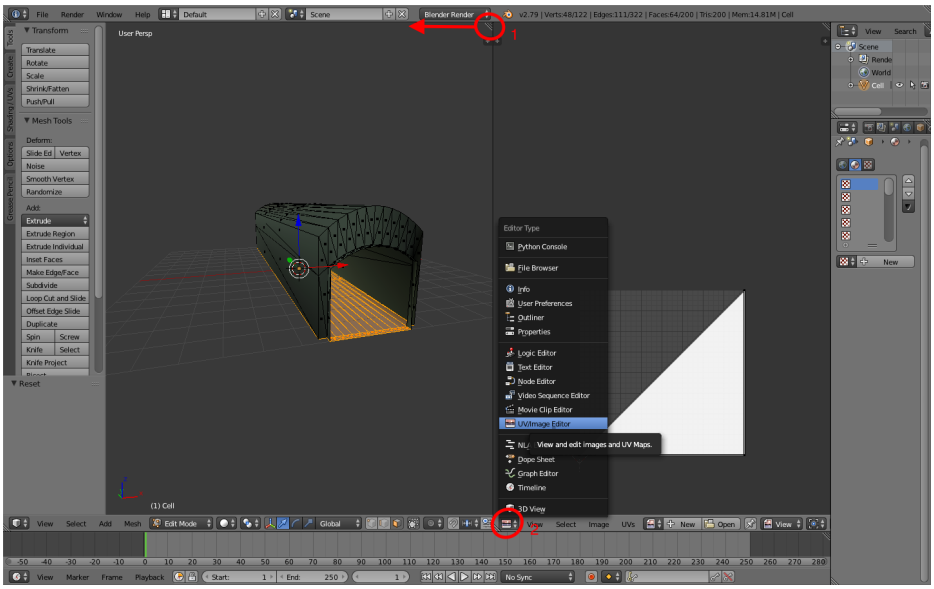
Kako bi blender učitao teksture u materijal, potrebno je na izborniku svojstava objekta otići na ikonu za teksture objekta. Nakon toga u izborniku za teksture je potrebno stvoriti novu teksturu klikom na gumb, te se može dodati naziv.

Sl. 2.12 Izrada nove prazne teksture (1 – Izbornik za teksture, 2 – Lista tekstura za trenutni materijal, 3 – Gumb za stvaranje nove teksture).

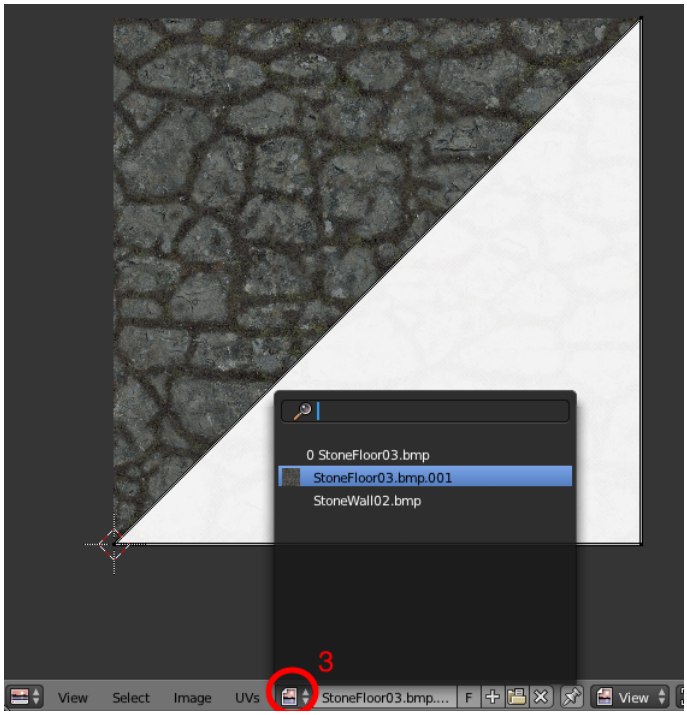


Sl. 2.13 Otvaranje datoteke teksture, nakon izrade nove teksture (4 – Gumb za učitavanje slike teksture, učitana StoneFloor03.bmp).

Sada model ima teksturu, ipak tekstura se još uvijek ne prikazuje na modelu. Razlog tome je što model nema 2D projekciju svojih lica na učitanu teksturu iz datoteke. Kako bi se izvršio UV Unwrap, potrebno je učitati novu karticu sa izbornikom „UV/Image Editor".

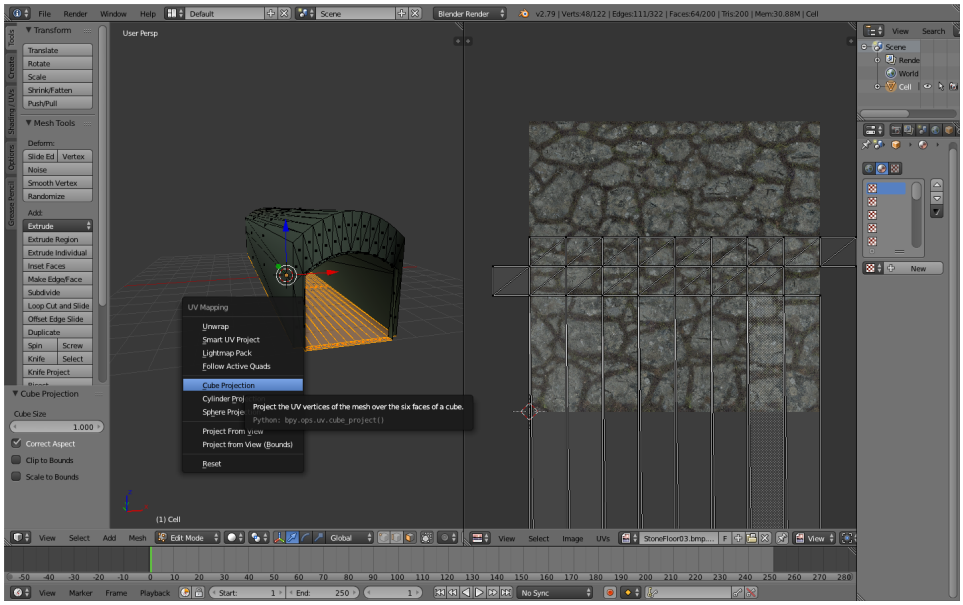


Sl. 2.14 Otvaranje UV/Image Editora (1 – Da bi se otvorila novu karticu, potrebno je držati lijevu tipku miša na iscrtkani trokutić na gornjoj desnoj strani i vući kurzor udesno; 2 – Nakon stvaranja nove kartice, potrebno je odabrati „UV/Image Editor" iz izbornika za uređivanje projekcije na teksturu).



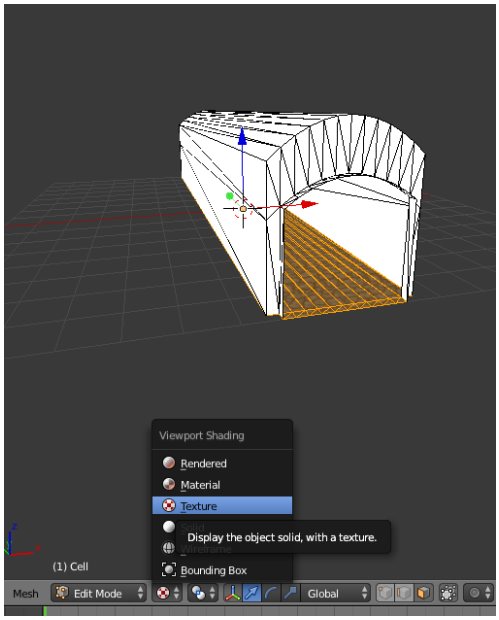
Sl. 2.15 Primjena teksture na 2D projekciju lica modela (3 – odabir teksture).

Konačno sada treba projicirati model na 2D teksturu. Postoji puno načina kako izvršiti UV Unwrap, i to je često najsloženiji dio kod izrade 3D modela. Najlakši i najbrži način kako izvršiti UV Unwrap je da se označe sva lica trenutnog materijala, te pritiskom tipke u odabere opcija „Cube Projection”.



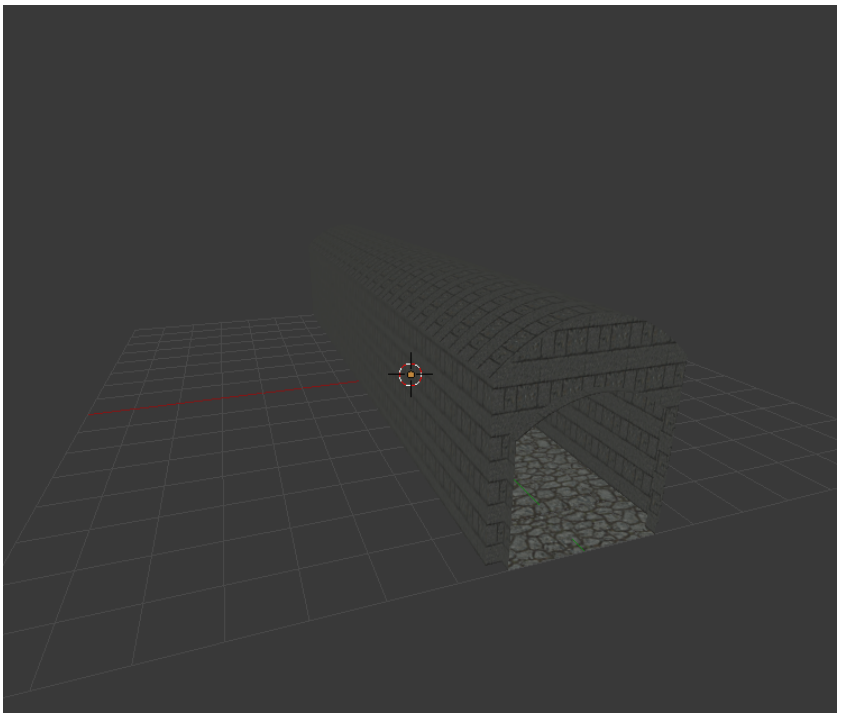
Sl. 2.16. Cube Projection u akciji.

Sada se mogu prikazati teksture 3D modela odabirom gumba načine prikaza 3D modela.



Sl. 2.17. Odabir pogleda na teksture. Može se primijetiti tekstura na podu ćelije, kod označenih lica.

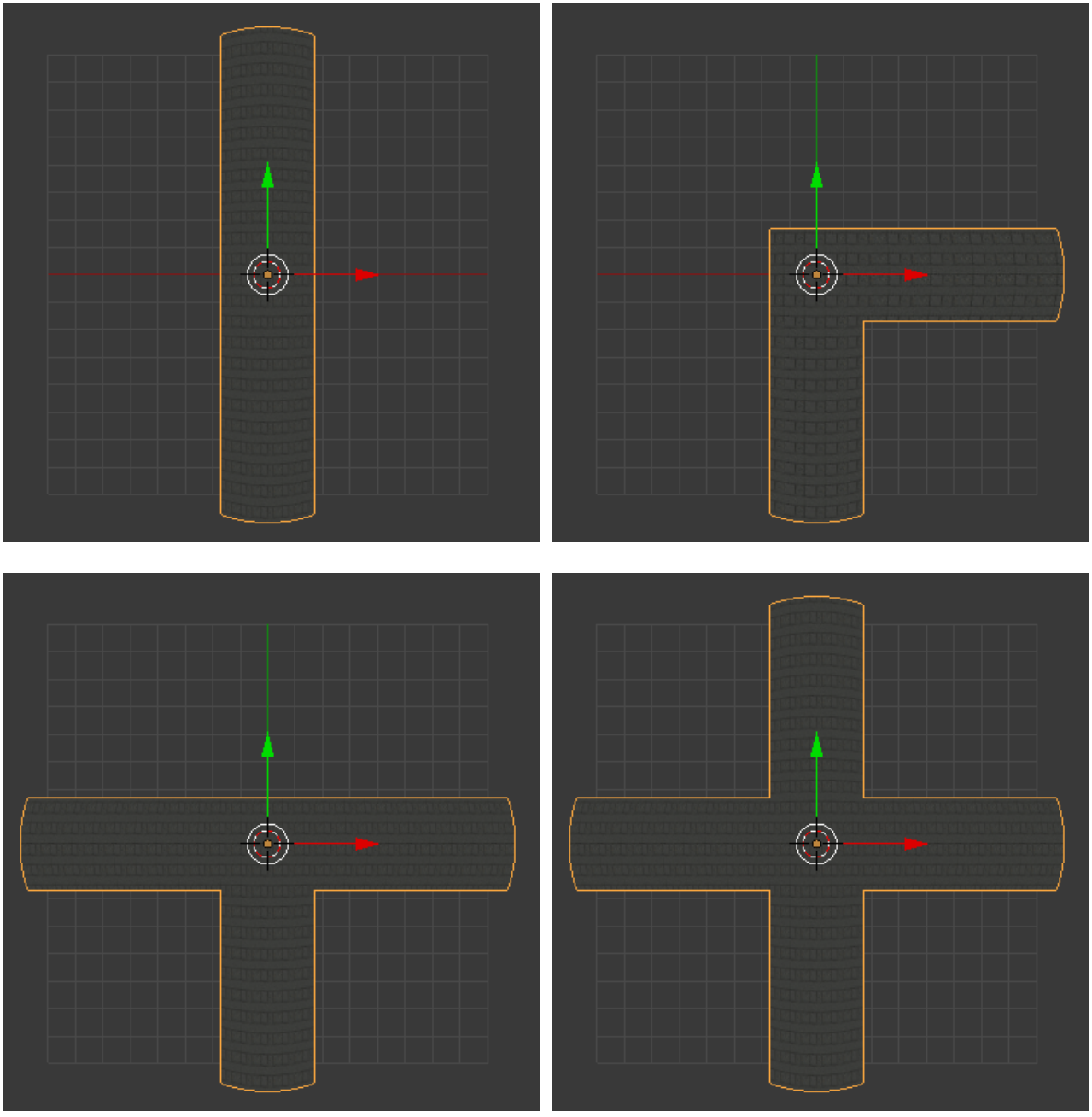
Ponavljanjem postupka UV Unwrapa za ostale materijale dobivamo konačan izgled ćelije.

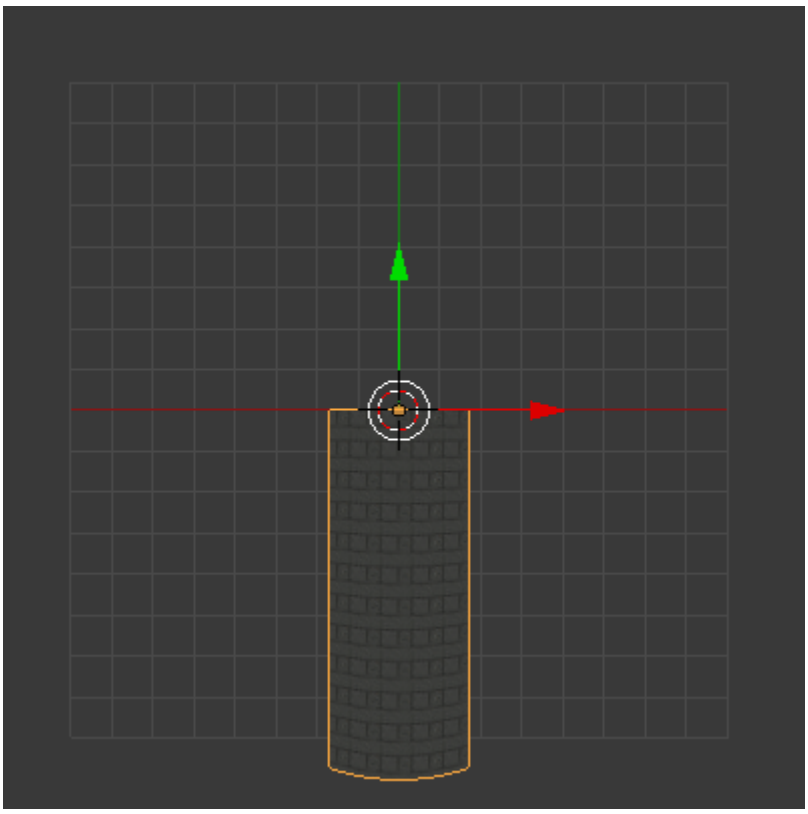


Sl. 2.18. Konačan izgled modela ćelije.

2.4. Tipovi ćelija

Generator tamnice razlikuje 4 tipa ćelija. Ćelija koja je bila izrađena u poglavljima 2.2. i 2.3. je ravna ćelija *(engl. Straight Cell)*. Međutim, kako bi generator pravilno izrađivao tamnicu, potrebna su još tri tipa ćelije: ćelija oblika L *(engl. L-Shaped Cell, Curved Cell)*, ćelija oblika T *(engl. T-Shaped Cell)*, ćelija križnog oblika *(engl. Crossed Cell)*, te slijepi hodnik *(engl. Dead End)*.





Sl. 2.19. Opisi raznih tipova ćelija: **a)** Ravna ćelija, ima dva otvora, jedan naprijed, drugi nazad. **b)** Ćelija oblika L, ima dva otvora, jedan nazad, drugi desno. **c)** Ćelija oblika T, ima tri otvora, jedan nazad, drugi desno i treći lijevo. **d)** Ćelija križnog oblika, ima otvore sa svih 4 strana. **e)** Ćelija slijepog hodnika, ima samo jedan otvor prema nazad.

Svi 3D modeli tipova ćelija su izrađeni na istom principu, kao u poglavljima 2.2. i 2.3.

3. GENERATOR TAMNICA (DUNGEON GENERATOR)

3.1. Uvod u generator tamnica

Dungeon Generator (ime datoteke, DungeonGenerator.cs, prilog P. 3.1.) je algoritam za automatizirano i nasumično stvaranje tamnice iz jednostavnih gradivnih blokova, ćelija. Algoritam je napisan u programskom jeziku C# od nule *(engl. from scratch)*, te nema nikakve veze sa postojećim algoritmima na Internetu (osim krajnjeg cilja, naravno). Nastanak ovog algoritma inspiriran je igrama kao što su: „Legend of Grimrock", te „SCP: Containment Breach".

Postojeći algoritmi stvaranja tamnica vrlo su složeni i nerazumljivi. Ovaj algoritam je osmišljen tako da na što jednostavniji i razumljiviji način nasumično stvori tamnicu. Zato uz bogato komentirani izvorni kod, iznosi svega oko 250 linija koda.

3.2. Izrada i logika algoritma

Algoritam se sastoji od 2 klase. Glavna klasa se zove DungeonGenerator i ona sadrži sve atribute i metode koji se koriste za stvaranje tamnica.ž

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

public class DungeonGenerator {
        // Constants
        public const ushort MAXCELLSNUM = 300;

        // Methods
        public List<Cell> generateDungeon() {
                /* (...) */
        }
}
```

Isječak koda 3.1. Klasa DungeonGenerator i njezini glavni atributi i metode.

Druga klasa zove se Cell i ona sadrži numeričke podatke pojedine ćelije (pozicija na x-osi, pozicija na y-osi, podaci da li ćelija ima susjede: odostraga, desno, sa preda, lijevo). Klasa Cell se koristi unutar klase DungeonGenerator, te metoda klase DungeonGenerator (generateDungeon()) vraća

listu klase Cell, koja zapravo predstavlja podatke cjelokupne tamnice i koristi se kao nacrt za postavljanje konkretnih 3D modela ćelija nakon generacije.

```
public class Cell {
        // Attributes
        public byte neighborsCode;
        public short positionX;
        public short positionY;

        // Default Constructor
        public Cell() {
                this.neighborsCode = 0;
                this.positionX = 0;
                this.positionY = 0;
        }

        // Paremeter Constructor
        public Cell(byte neighborsCode, short x, short y) {
                this.neighborsCode = neighborsCode;
                this.positionX = x;
                this.positionY = y;
        }

        // Methods
        public void setState(byte newNeighborsCode, short newX, short newY) {
                this.neighborsCode = newNeighborsCode;
                this.positionX = newX;
                this.positionY = newY;
        }

        public void clearState() {
                this.setState(0, 0, 0);
        }

        public bool checkIfCleared() {
                if(this.neighborsCode == 0) {
                        return true;
                }

                return false;
        }

        public string printCellContent() {
                return (
                        this.neighborsCode.ToString() + " "
                        + this.positionX.ToString() + " "
                        + this.positionY.ToString() + "\n"
                );
        }
}
```

Isječak koda 3.2. Kod klase Cell.

Kao što je već spomenuto, sama generacija tamnice odvija se unutar DungeonGenerator klase, specifično u njezinoj metodi generateDungeon(). Ideja generacije vrlo je jednostavna. Unutar metode postoje dvije liste:

```
List<Cell> cells = new List<Cell>();
```
Isječak koda 3.3. Lista cells.

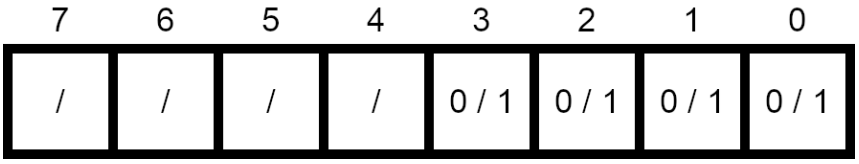
```
List<Cell> cellQueue = new List<Cell>();
```
Isječak koda 3.4. Lista cellQueue.

3.2.1. List<Cell> cells

Lista cells bilježi sve novo stvorene ćelije, tj. njihove pozicije po x-osi (cells[i].positionX), y-osi (cells[i].positionY), te podatak da li imaju susjedne ćelije na 4 moguće strane (cells[i].neighborsCode), $i \in [\ 0, MAXCELLSNUM - 1\ ]$. Te strane mogu biti prema nazad, odnosno južna strana (ubuduće označavat će se sa S), prema desno, odnosno istočna strana (oznaka E), prema naprijed, odnosno sjeverna strana (oznaka N) te prema lijevo, odnosno zapadna strana (oznaka W). Zapis susjednih ćelija omogućava relativno lako određivanje tipa trenutne ćelije, te njezinu orjentaciju.

Međutim, problem je kako zapisati strane na kojima se nalazi susjedna ćelija, te kako znati koja je to, od četiri moguće strane, ona na kojoj se nalazi susjedna ćelija. Rješenje je da se kodiraju podaci u binarnom zapisu. Jedan bit može biti 0 ili 1, odnosno ima susjednu ćeliju ili nema susjednu ćeliju. Također kako ima 4 raspoložive strane, dovoljna su 4 bita kako bi se saznala i strana na kojoj se nalazi susjedna ćelija. Kako varijabla neighborsCode unutar klase Cells zauzima 1 bajt, odnosno 8 bita (jer se ne može zauzeti manji dio u memoriji), imamo dodatna 4 bita viška, što ne predstavlja veliki problem. Jednostavno ta 4 bita zanemarimo; od viška glava ne boli.

Dakle, ovako je osmišljen model zapisa susjednih ćelija:



Sl. 3.1. Zapis susjednih ćelija u 1 bajt podatka.

Slika Sl 3.1. predstavlja strukturu memorijskog bloka od 1 bajta varijable neighborsCode. Može se primjetiti da bitovi na pozicijama 7, 6, 5 i 4 nisu korišteni jer su suvišni. Svaki bit na pozicijama 3, 2, 1 i 0 može biti 0 ili 1.

Bit na poziciji 0 govori ima li trenutna ćelija susjednu ćeliju na strani S.

Bit na poziciji 1 govori ima li trenutna ćelija susjednu ćeliju na strani E.

Bit na poziciji 2 govori ima li trenutna ćelija susjednu ćeliju na strani N.

Bit na poziciji 3 govori ima li trenutna ćelija susjednu ćeliju na strani W.

Može se proizvoljno definirati koji će bit određivati sa koje strane će biti susjedna ćelija. Međutim, zbog praktičnih razloga koristi se baš ovakav model. Više o tome će biti objašnjeno u metodi initCellLists.

3.2.2. List<Cell> cellQueue

Na sličan način je osmišljena varijabla cellQueue. Ipak, ona ima značajne razlike. Glavna razlika je ta što se varijabla cellQueue ponaša kao struktura podataka red *(engl. Queue)*, te je njezina svrha privremeno skladištiti nadolazeće ćelije koji će biti budući susjedi one ćelije pozivatelja, koristeći FIFO (First In First Out) paradigmu. Ćelija pozivatelj je ona ćelija koja, nakon što se stvori, nadoda se na listu cells, te nasumično odredi raspoloživa mjesta gdje će proširiti nove ćelije. Te nove ćelije biti će budući susjedi ćeliji pozivatelju i pohranjuju se upravo u cellQueue.

Kako bi se pravilno odredila buduća ćelija u varijablu positionX i positionY pohranjuje se pozicija ćelije pozivatelja. U varijablu neighborsCode pohranjuje se strana na koju je ćelija pozivatelj odredila da će biti nova ćelija smještena.

Ćelija pozivatelj može se proširit na više smjerova buduće ćelije, no buduća ćelija može imati samo jednog pozivatelja. Zato buduća ćelija može biti pozvana u točno jednu od četiri strana. Za to su

potrebna samo 2 bita (umjesto 4 bita kao što je to slučaj za listu cells), koja se zapisuju u varijablu neighborsCode.

Zapis u 2 bita može imati 4 moguće vrijednosti: 0, 1, 2 ili 3.

Ovako je osmišljen zapis:

0 – ćelija pozivatelj proširila se na stranu S

1 – ćelija pozivatelj proširila se na stranu E

2 – ćelija pozivatelj proširila se na stranu N

3 – ćelija pozivatelj proširila se na stranu W

Može se primijetiti da je zapis u red cellQueue drugačiji nego lista cells, iako je to zapravo ista klasa. Lista cells skladišti podatke o pojedinim ćelijama tamnice, dok lista cellQueue privremeno pamti red pozivanja za proširivanje tamnice novim ćelijama.

Prije glavne petlje na 46. liniji, liste cells i cellQueue su prazne, te ih je prije ulaza u glavnu petlju potrebno popuniti početnim vrijednostima. Metoda initCellLists() upravo to omogućava.

```
private void initCellLists(List<Cell> cells, List<Cell> queue) {
        // Set the first cell

        /*
        About firstNeighborsCode:
        It can generate these possible values: 3, 5, 7, 9, 11, 13, 15
        This output is generated so that the first cell can have neighbors on
        any of following sides: (front, right, left).
        Exception is that the first cell ALWAYS has neighbor from the back,
        since that is Start Cell, which is predefined in editor.
        */
        byte firstNeighborsCode = (byte)(Random.Range(1, 8) * 2 + 1);

        // Position begins at origin, so positionX and positionY should be 0.
        cells.Add(new Cell(firstNeighborsCode, 0, 0));

        // Store initial cell neighbors for expansion (except for the one from
           the back, the Start Cell already exists as a neighbor, so we don't
           need to expand the dungeon on the back).
        for(byte i = 1; i < 4; i++) {
                if((firstNeighborsCode >> i & 1) == 1) {
                        queue.Add(new Cell(i, 0, 0));
                }
        }
}
```

Isječak koda 3.5. Metoda initCellLists().

3.2.3. Metoda initCellLists()

Ova metoda postavlja poziciju početne ćelije i u queue nasumično učitava pozive za buduće susjedne ćelije, kojih može biti više. Inicijalna ćelija se stvori u odnosu na Start ćeliju. Start ćelija je unaprijed definirana na poziciji (0, -1) s otvorom na strani N. Drugim riječima očekuje susjednu ćeliju na poziciji (0, 0). Upravo susjedna ćelija Start ćelije jest inicijalna ćelija koju stvori metoda initCellLists.

Nadalje inicijalna ćelija treba imati susjednu Start ćeliju na svojoj strani S. Također inicijalna ćelija treba proširiti buduće susjedne ćelije na najmanje jednu stranu (ali može više). Koje će joj nasumično određene susjedne ćelije biti (uz Start ćeliju) na kompaktan način opisuje 187. linija.

```
byte firstNeighborsCode = (byte)(Random.Range(1, 8) * 2 + 1);
```

Isječak koda 3.6. Nasumično određivanje budućih susjednih ćelija početnoj ćeliji.

Kako je spomenuto kod liste cells, pozicije bita stranama susjednih ćelija su određene zbog praktičnih razloga. Taj razlog je upravo ova formula. Pomoću te formule jednom linijom se može definirati bilo koja kombinacija susjednih ćelija, uz unaprijed definiranu Start ćeliju na strani S inicijalne ćelije.

Nakon što se odredila inicijalna ćelija, samo ju je potrebno pohraniti u listu cells.

```
cells.Add(new Cell(firstNeighborsCode, 0, 0));
```

Isječak koda 3.7. Dodavanje početne ćelije u listu cells.

Ipak, tu nije kraj, inicijalna ćelija ima susjedne ćelije, koje još trebaju biti stvorene. Te susjedne ćelije je potrebno, jednu po jednu, pohraniti u queue. To se postiže jednostavnom for petljom.

```
for(byte i = 1; i < 4; i++) {
   if((firstNeighborsCode >> i & 1) == 1) {
      queue.Add(new Cell(i, 0, 0));
   }
}
```

Isječak koda 3.8. Dodavanje budućih susjednih ćelija u listu queue (koja je zapravo referenca na listu cellQueue unutar metode generateDungeon()).

Nakon što su lista cells i red cellQueue postavljeni na početne vrijednosti algoritam može preći na svoj glavni dio, koji se odvija u while petlji na 46. liniji.

3.2.4. Glavna while petlja

Unutar while petlje se događa slično što se događalo unutar metode initCellLists(). Ćelije koje još nisu stvorene, skidaju se sa reda cellQueue, te se postavljaju na svoju pripadajuću poziciju, nakon toga nasumično odabiru gdje će imati nove susjedne ćelije, koje se zatim postavljaju u cellQueue. Taj proces se ponavlja sve dok se u potpunosti ne isprazni red cellQueue. Kako se cellQueue brže puni, nego što se prazni (jer novonastala ćelija pozivač može proširiti buduće susjedne ćelije u više smjerova odjednom, što znači više od jednog zauzetog mjesta u cellQueue po iteraciji, dok u jednoj iteraciji sa reda se skida samo jedna ćelija), te tamnica ima tendenciju proširivati se u beskonačnost. Da bi se to spriječilo, postavila se konstanta za maksimalan moguć broj ćelija u tamnici.

```
public const ushort MAXCELLSNUM = 100;
```

Isječak koda 3.9. Konstanta MAXCELLSNUM. Nalazi se na početku klase DungeonGenerator. Trenutno joj je vrijednost 100, za srednje velike tamnice.

Kada ukupan broj ćelija koje su već stvorene (u listi cells) i koje se trebaju stvoriti (u redu cellQueue) dosegne MAXCELLSNUM. Algoritam više neće stvarati ćelije koje se mogu dalje proširiti. Umjesto toga, sve ćelije koje su još u redu će biti stvorene kao slijepi hodnici *(engl. Dead End)*. Slijepi hodnici imaju samo jedan otvor prema onoj ćeliji pozivatelju sa koje su se proširili, te ne mogu dalje proširivati tamnicu. To će rezultirati pražnjenjem reda cellQueue dok se u potpunosti ne isprazni i time algoritam završava.

Uvjet koji regulira stvaranje ćelija. Ukoliko uvjet nije zadovoljen, algoritam će preskočiti mogućnost proširivanja ćelije, te je ostaviti samo kao slijepi hodnik:

```
// Switch newCell.neighborsData from cellQueue to it's actual neighbors data.
newCell.neighborsCode = neighborCodes[(newCell.neighborsCode + 2) & 3];

// Check if total number of cells has reached MAXCELLSNUM.
if(cells.Count + cellQueue.Count <= MAXCELLSNUM) {
        /* (...) */
}
```

Isječak koda 3.10. Uvijet koji regulira stvaranje ćelija.

U suprotnom, ukoliko je uvjet zadovoljen, ćelija će se proširiti prirodnim putem, slično kao u metodi initCellLists().

Uz sve dosada navedene metode, uvjete i strukture podataka algoritam u općem smislu pravilno radi, tj. proširuje tamnicu sve dok ona ne dosegne maksimalni broj ćelija.

Međutim, postoji problem, tzv. problem kolizije ćelija. Naime, riječ je o tome da se može dogoditi slučaj da ćelija koja se treba stvoriti sa reda cellQueue, stvori se na mjesto već postojeće ćelije koja se stvorila u nekoj od prethodnih iteracija. Nova ćelija bi se tada stvorila na isto mjesto kao prethodno stvorena ćelija, što bi uzrokovalo da jedna ćelija prolazi kroz drugu *(engl. clipping)*, što nema smisla.

Zato treba postaviti uvjet koji će ispitivati je li došlo do kolizije i otkloniti problem.

```
private ushort checkCellCollisions(Cell newCell, List<Cell> cells) {
        ushort numOfCells = (ushort)cells.Count;

        ushort i = 0;
        while(i < numOfCells) {
                if(cells[i].positionX == newCell.positionX
                && cells[i].positionY == newCell.positionY) {
                        break;
                }
                i += 1;
        }

        return i;

}
```

Isječak koda 3.11. Metoda koja ispituje je li došlo do kolizije ćelija. Uvjet se nalazi unutar while petlje.

Rješenje problema je da se jednostavno ćelija koja treba biti stvorena ukloni, a postojeća ćelija da igra ulogu novonastale ćelije. Drugim riječima, postojeća ćelija kao dodatnu susjednu ćeliju ima ćeliju pozivatelja uklonjene ćelije. Zato, umjesto da se stvori nova ćelija koja će bi se dalje trebala proširivati, postojeća ćelija jednostavno ažurira svoje susjedne ćelije.

```
// Index of the element from cells list.
// Shows if newCell is trying to be placed at the position of already existing
cell.
ushort collision = checkCellCollisions(newCell, cells);

// Check for possible collisions for current cell.
if(collision < cells.Count) {
        // New cell is trying to be placed where the other cell is already
           placed.
        // Solution: don't create current cell, and for already placed cell set
           previous cell as neighbor.
        if(((cells[collision].neighborsCode >> ((newCell.neighborsCode + 2) &
           3)) & 1) == 0) {
                cells[collision].neighborsCode += (byte)(1 <<
                ((newCell.neighborsCode + 2) & 3));
        }
}
```

Isječak koda 3.12. Dio algoritma koji rješava problem kolizije.

Uz uvjet kolizije algoritam je upotpunjen i može izraditi tamnicu u numeričkom obliku. Kako vraća listu ćelija (cells), ovaj generator može koristiti drugim klasama u Unityu, koje će koristiti listu kao nacrt za postavljanje 3D modela ćelija na pravo mjesto. Više o tome u slijedećem poglavlju.

4. IMPLEMENTACIJA GENERATORA TAMNICA U UNITY ENGINEU

4.1. Uvod u Unity

Unity je jedan od najpopularnijih i najmoćnijih enginea za izradu video igra. Unity ima vrlo veliki broj značajki, kao što su AI, grafičko sučelje, simulacija fizike, podrška za online multiplayer način igranja, i mnoge druge. Zbog toga je najomiljeniji alat među game devleoperima za izradu igra, kako individualnim developerima ili malim timovima „indie" igara (Cup Head, Super Hot, Slender: The Eight Pages, HuniePop), tako i velikim firmama AAA igara (Angry Birds 2, Pokemon Go, Deus Ex: The Fall, The Elder Scrolls: Legends). Unity u sebe ima ugrađen već spomenuti C#, te JavaScript programski jezik, koji se mogu kombinirano koristiti. Igre izrađene u Unity engineu mogu se čak kompajlirati za Windows, Linux i macOS mašine, te za Android i iOS mobilne uređaje i tablete.

4.2. Sučelje u Unity engineu

Sučelje u Unity engineu je uglavnom grafičko, te obiluje velikim brojem opcija, gumbova i datoteka, što može biti zbunjujuće onima koji se prvi put susreću sa Unity engineom. Radi jednostavnost, Unity sučelje se može podijeliti na 5 djelova:

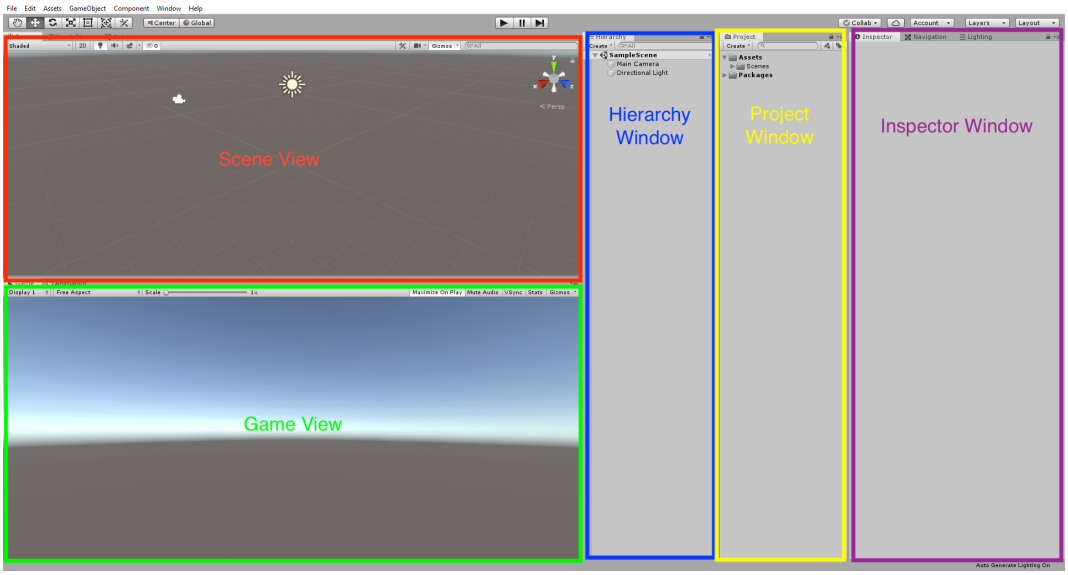
Scene View – Prikaz izgleda scene u igri prije pokretanja.

Game View – Interaktivni vizualni prikaz igre, dok je igra u runtimeu.

Hierarchy Window – Sadrži popis svih objekata, koji se trenutno koriste u igri.

Project Window – Popis svih datoteka u projektu, u njega se mogu učitati C# skripte, 3D modeli, teksture, zvukovi i još mnogo toga.

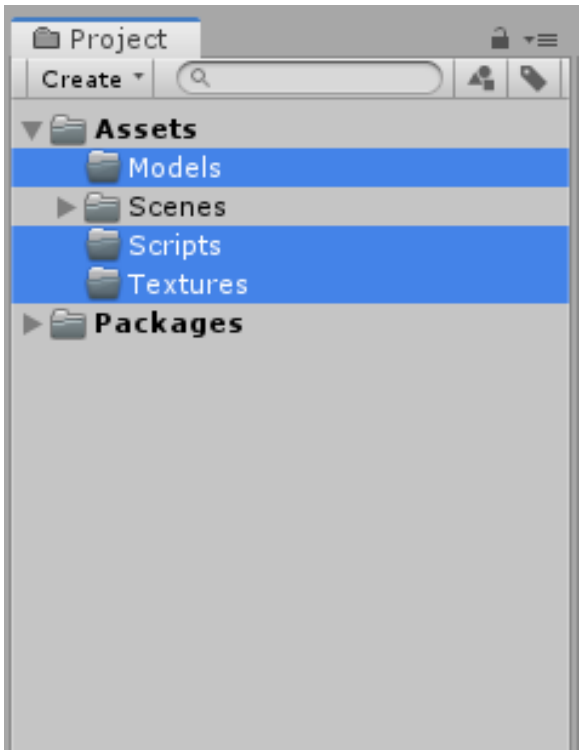
Inspector Window – Prikaz svojstva i detalja pojedinog objekta u igri.



Sl. 4.1. Karakteristično sučelje u Unity engineu.

4.3. Implementacija generatora tamnica

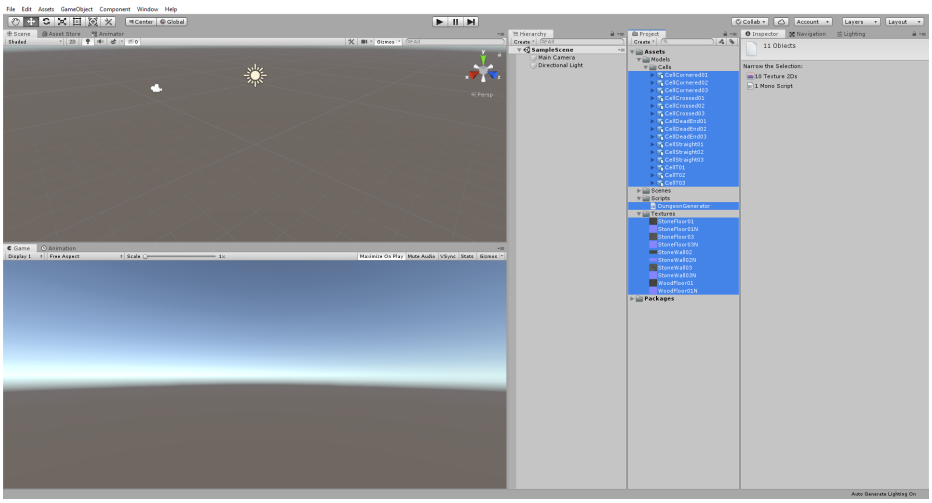
Kako bi Unity ispravno stvarao tamnice, potrebno je u njega uvesti skriptu DungeonGenerator.cs, 3D modele ćelije i njihove teksture. Najprije u Project Window desnim klikom stvorimo odgovarajuće mape: „Models”, „Textures” i „Scripts”.



Sl. 4.2. Nove stvorene mape u Project Windowu.

Sada je jednostavno moguće povući datoteke sa 3D modelima ćelija u Project Window u Models mapu. Svi 3D modeli ćelija su izvezeni iz Blendera u .fbx datoteku. Iako Unity može čitati .blend datoteke, zbog jednostavnosti koristi se .fbx format za 3D modele.

Isto to vrijedi za pripadajuće teksture 3D modela, samo što njih treba povući unutar Textures mape. DungeonGenerator.cs se također može povući unutar Scripts mape.



Sl. 4.3. Objekti (modeli, skripte, teksture) dodani u Unity.

Međutim, tu nije kraj. Treba nekako povezati konkretne 3D modele sa pripadajućim numeričkim vrijednostima ćelija, koje izradi DungeonGenerator. Kako je spomenuto u prethodnom poglavlju DungeonGenerator klasa može se sa lakoćom uključiti u druge klase.

Zato, treba napraviti novu C# skriptu koja će istovremeno u sebi držati 3D modele ćelija, te DungeonGenerator. Tada će od numeričkih vrijednosti iz klase DungeonGenerator i 3D modele konstruirati objekte unutar igre.

Novu C# skriptu moguće je napraviti označavanjem mape Scripts popratno sa desnim klikom → „Create” → „C# Script”. Novostvorenoj C# skripti moguće je dodijeliti bilo koje ime. U ovom projektu dodijeljeno je ime „Main” (datoteka Main.cs, prilog P. 4.1.), jer ona sadrži sve glavne elemente koji su potrebni za izradu tamnice, te sadrži glavnu metodu Start(), koja se pokreće kada god pokrenemo igru (slično kao funkcija main() u jezicima C i C++). Prilikom otvaranja Main.cs datoteke, datoteka sadrži klasu imenom „Main” te dvije prazne metode Start() i Update().

Metoda Update() se koristi u Unity engineu za ažuriranje igre po svakoj sličici. Međutim, nema potrebe ništa ažurirati, te se metoda Update() može ukloniti. Izvorni kod Main.cs bi trebao izgledati:

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

public class Main : MonoBehaviour {
        // Start is called before the first frame update.
        void Start() {

        }
}
```

Isječak koda 4.1. Prazna klasa Main.

Zasada je ova klasa prazna. Treba ju najprije popuniti konstantama i atributima.

Odmah na početku klase Main, prije metode Start() treba dodati slijedeće konstante:

```
// Constants
private const byte CELLSNAPUNIT = 16;
private const byte NUMCELLVARIANTS = 3;
private const byte NUMCELLTYPES = 5;
```

Isječak koda 4.2. Konstante koje su nadodane na početak klase Main.

Konstanta CELLSNAPUNIT sadrži stvarnu fizičku veličinu 3D modela ćelija (tj. njezine ograničavajuće kutije). Kako su numeričke vrijednosti pozicije ćelija u skripti DungeonGenerator izražene preko cjelobrojnih k jedinica pomaka, $k \in \mathbb{Z}$, stvarna pozicija ćelija se dobiva množenjem CELLSNAPUNIT-a sa k, tj. $16k$.

Konstanta NUMCELLVARIANTS sadrži broj mogućih varijanti za svaki tip ćelije, odnosno broj drugačijih oblika za isti tip ćelije. Za ovaj projekt izradilo se za svaki tip ćelije 3 različita 3D modela, koji će nasumično biti odabrani za svaku ćeliju. Tamnica tada dobiva na raznolikosti i manje je monotona.

Konstanta NUMCELLTYPES govori koliko ima tipova ćelija. Naravno, kako je objašnjeno u 2. poglavlju, postoje 5 tipa ćelija (Ravne, L-oblik, T-oblik, križne, slijepi hodnici).

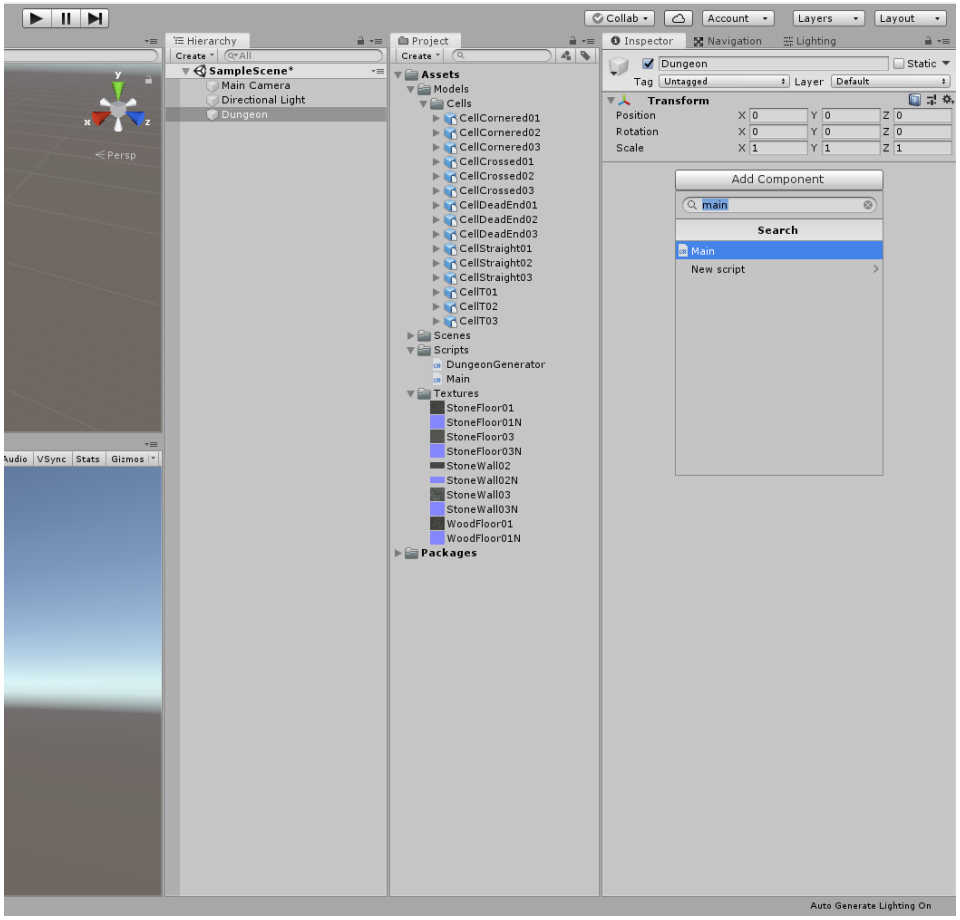
Nakon konstanti, treba definirati atribute, tako da 3D modeli u Unity engineu budu vidljivi klasi Main.

```
// Attributes
public Transform[] cellModels;
private List<Transform> cells;
```

Isječak koda 4.3. Atributi kojeisu nadodani na početak klase Main.

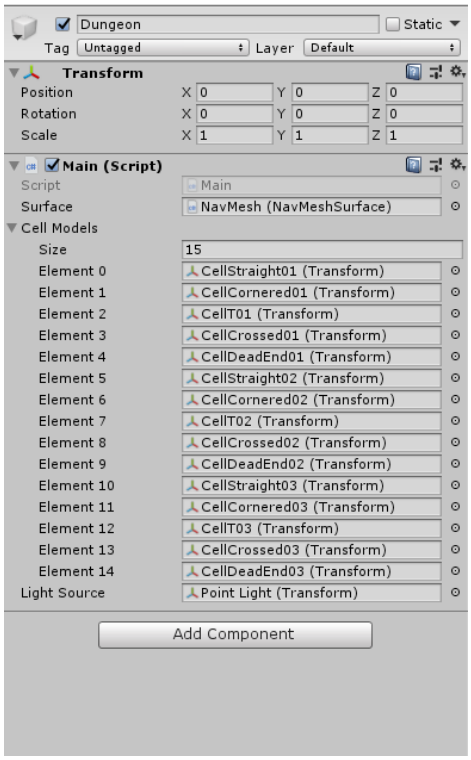
Atribut cellModels sadrži definicije 3D modele svih tipova ćelija i svih njihovih varijanti, dok atribut cells sadrži instance tih 3D modela kao objekti u igri, tj. oni čine konkretnu tamnicu.

Nakon definiranja atributa cellModels, treba sa tim atributom povezati 3D modele ćelija u Unityu. Da bi se to odradilo, potrebno je stvoriti prazni objekt u Hierarchy Window djelu Unity sučelja, sa prikvačenom Main.cs skriptom na njega. Jednostavno to se može napraviti desnim klikom u prazni prostor Hierarchy Windowa, te iz izbornika odabrati „Create Empty". To će stvoriti prazan unaprijed definirani objekt unutar igre, možemo ga nazvati Dungeon, jer će taj na taj objekt biti prikvačena skripta Main.cs, te će taj objekt biti kontejner za sve ćelije koje klasa Main instancira. Nakon što se Dungeon objekt uspješno stvori, sa lijevim klikom potrebno ga je označiti i na Inspector Windowu pritisnuti gumb „Add Component", te odabrati Main.



Sl. 4.4. Dodavanje skripte Main na objekt Dungeon.

Nakon što je Main odabran, skripta je uspješno prikvačena na Dungeon objekt. Sada je potrebno atribute popuniti 3D modelima. Najprije je potrebno odrediti veličinu polja Cell Models *(engl. Array)* za modele ćelija. Kako svaki tip ćelije ima 3 varijante, a ima 5 tipova ćelija, to je ukupno 15 modela, zato na „Size" polje treba unijeti broj 15. Nakon unosa veličine polja Cell Models, u Inspector Windowu nastat će još dodatnih 15 polja koje treba popuniti, kao pojedini elementi polja Cell Models. Polja se mogu popuniti povlačenjem 3D modela ćelija sa Project Windowa prema praznim poljima unutar Inspector Windowa. Elementi polja Cell Models se popunjuju tako da se najprije dovuku svi tipovi ćelija jedne varijante, pa druge varijante i na kraju treće varijante. Od tipova ćelija prvo se postavljaju ravne ćelije, onda ćelije oblika L, nakon toga ćelije oblika T, pa križaste ćelije i na kraju slijepi hodnici.



Sl. 4.5. Izgled atributa cellModels nakon što se Cell Models polje popuni.

Sada kada su svi atributi postavljeni, može se nastaviti raditi na Main klasi.

U metodi Start() potrebno je postaviti DungeonGenerator klasu koja će stvoriti podatke po kojim će se sagraditi tamnica, te inicijalizirati listu cells, koja će imati objekte ćelija unutar igre.

```
void Start() {
        this.cells = new List<Transform>();
        List<Cell> dungeon = generator.generateDungeon();
        this.deployDungeon(dungeon)
}
```

Isječak koda 4.4. Metoda Start().

Treba još samo implementirati metodu deployDungeon(), koja će stvoriti tamnicu koristeći podatke dobivene iz generatora tamnica.

```
private void deployDungeon(List<Cell> dungeon) {
        /* (...) */
}
```

Isječak koda 4.5. Definicija metode deployDungeon().

Metoda funkcionira tako da prođe kroz cijelu listu dungeon u kojoj se nalaze numerički podaci ćelija.

Unutar petlje, koja se nalazi na početku metode deployDungeon(), najprije se odrede koliko trenutna ćelija ima svojih svojih susjeda i sa kojih strana.

```
ushort dungeonSize = (ushort)dungeon.Count;

for(ushort i = 0; i < dungeonSize; i++) {
        byte neighborBack = (byte)(dungeon[i].neighborsCode & 1);
        byte neighborRight = (byte)(dungeon[i].neighborsCode >> 1 & 1);
        byte neighborFront = (byte)(dungeon[i].neighborsCode >> 2 & 1);
        byte neighborLeft = (byte)(dungeon[i].neighborsCode >> 3 & 1);

        byte neighborsCount = (byte)(neighborBack + neighborRight +
                              neighborFront + neighborLeft);
        /* (...) */
}
```

Isječak koda 4.6. Dio algoritma koji određuje koliko trenutna ćelija ima susjednih, i na kojim stranama.

Na osnovu broja susjednih ćelija moguće je odrediti tip ćelije, a pomoću razmještaja susjednih ćelija i rotaciju trenutne ćelije.

```
byte cellType = 0;
ushort cellRotation = 0;

if(neighborsCount == 1) {
        // Cell must be dead end.
        cellType = 5;

        if(neighborBack == 1) {
                cellRotation = 0;
        }
        else if(neighborRight == 1) {
                cellRotation = 270;
        }
        else if(neighborFront == 1) {
                cellRotation = 180;
        }
        else {
                cellRotation = 90;
        }
}
else if(neighborsCount == 2) {
        // Cell is either straight or cornered.

        // Lets assume it is straight.
        cellType = 1;
        if(neighborBack == 1 && neighborFront == 1) {
                // Cell rotation can be either 0 or 180 degs, it doesn't matter
                   because it's straight.
                cellRotation = 0;
        }
        else if(neighborRight == 1 && neighborLeft == 1) {
                // Cell rotation can be either 90 or 270 degs, it doesn't
                   matter because it's straight.
                cellRotation = 90;
        }
        else {
                // Cell is not straight, so it must be cornered.
                cellType = 2;
                if(neighborBack == 1 && neighborRight == 1) {
                        cellRotation = 0;
                }
                else if(neighborRight == 1 && neighborFront == 1) {
                        cellRotation = 270;
                }
                else if(neighborFront == 1 && neighborLeft == 1) {
                        cellRotation = 180;
                }
                else {
                        cellRotation = 90;
                }
```

```
        }
}
else if(neighborsCount == 3) {
        // Cell must be T-shaped.
        cellType = 3;
        if(neighborBack == 0) {
                cellRotation = 180;
        }
        else if(neighborRight == 0) {
                cellRotation = 90;
        }
        else if(neighborFront == 0) {
                cellRotation = 0;
        }
        else {
                cellRotation = 270;
        }
}
else {
        // Cell must be cross-shaped.
        cellType = 4;

        // Cell rotation can be ether 0, 90, 180 or 270, it doesnt matter
           because it has all 4 sides open.
        cellRotation = 0;
}
```

Isječak koda 4.7. Dio algoritma koji određuje tip ćelije i njezinu rotaciju.

Nakon što je ustanovljen tip i rotacija ćelije može se instancirati kao objekt u igri.

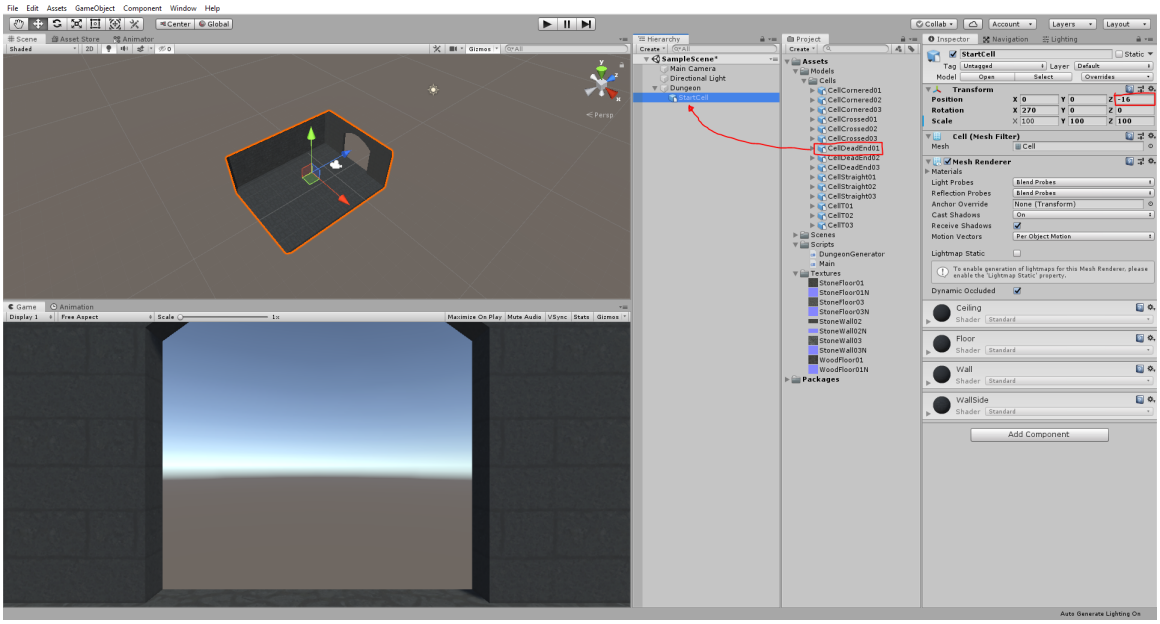
```
// Deploy the cell model into scene.
Transform newCellModel;
newCellModel = cellModels[Random.Range(0, NUMCELLVARIANTS) * NUMCELLTYPES +
cellType - 1];

int cellX = dungeon[i].positionX * CELLSNAPUNIT;
int cellY = dungeon[i].positionY * CELLSNAPUNIT;

cells.Add(Instantiate(newCellModel, new Vector3Int(cellX, 0, cellY),
          Quaternion.Euler(270, (cellRotation + 180) % 360, 0),
          this.transform) as Transform);
```

Isječak koda 4.8. Postavljanje objekta ćelije u igru.

Prije nego, što se pokrene igra, treba u scenu ručno dodati Start ćeliju, kako bi tamnica bila u potpunosti zatvorena. Također, kako je objašnjeno u poglavlju 2, Start ćelija je referentna ćelija i označava početak tamnice. Tu ćeliju treba ručno postaviti na poziciju 0, 0, -16. Jednostavno iz Project Windowa povučemo bilo koju varijantu modela slijepog hodnika u Hierarchy View, te pomoću Inspektor Viewa ručno promijenimo poziciju.



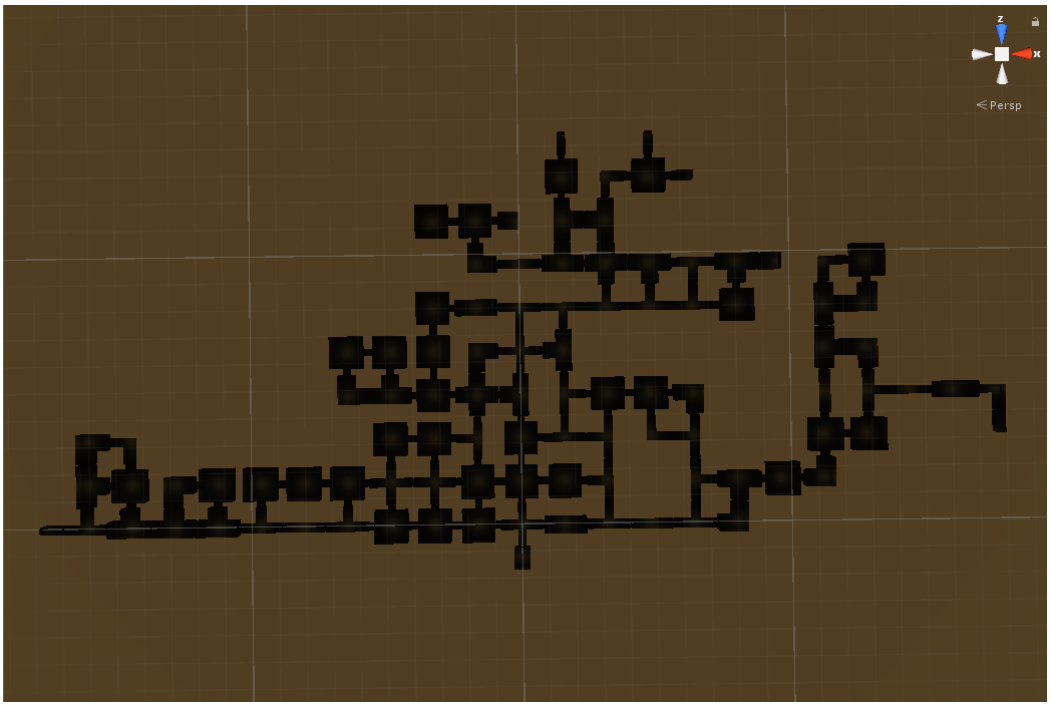
Sl. 4.6. Postavljanje Start ćelije u igru.

Sada možemo pokrenuti igru.

Igra u Unity engineu se pokreće odabirom gumba na vrhu sučelja:

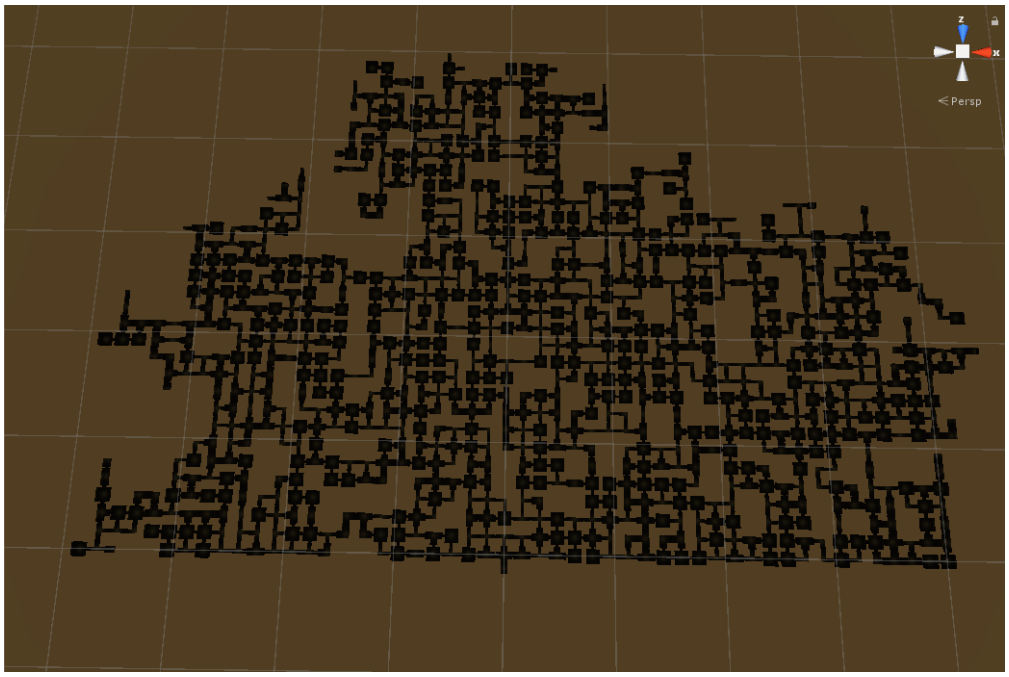
Sl. 4.7. Gumb za pokretanje igre.

5. REZULTATI

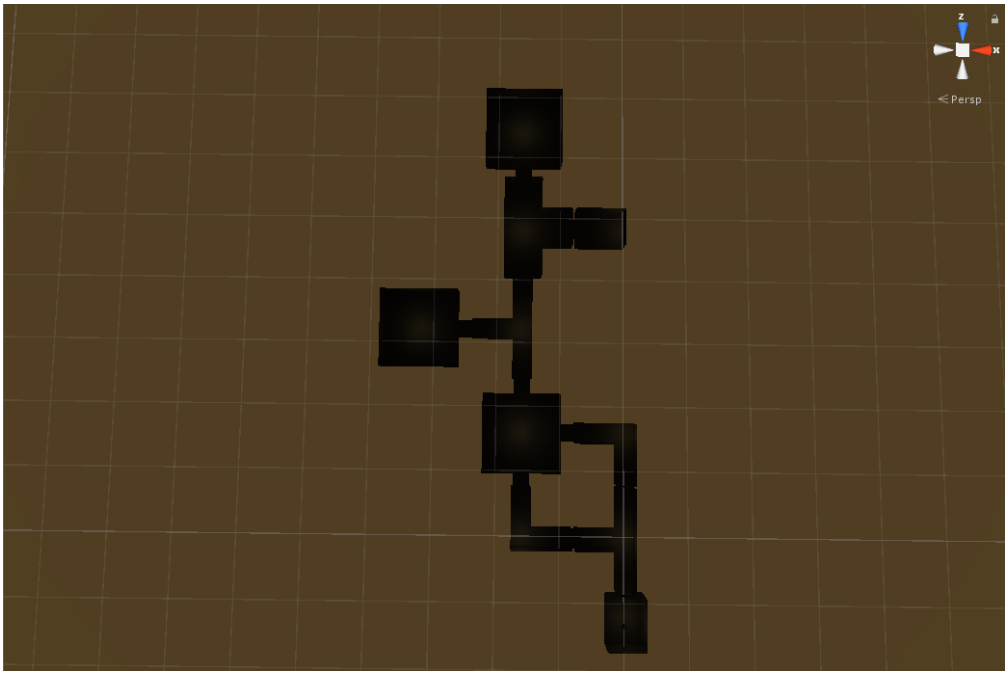


Sl. 5.1. Tamnica sa 100 ćelija.

Ukoliko tamnica ne zadovoljava željenu veličinu, može se lako proimjeniti. Jednostavno treba izmijeniti vrijednost konstante MAXCELLSNUM u DungeonGenerator klasi u željenu vrijednost.



Sl. 5.2. Ogromna tamnica sa 1000 ćelija.



Sl. 5.3. Mini tamnica sa 10 ćelija.

6. ZAKLJUČAK

Generator tamnica vrlo je kompaktan i prilagodljiv algoritam za brzu generaciju osnovnih razina tamnice. Ipak, algoritam nema naprednijih značajki kao što su širenje tamnice i po visini. Naime, algoritam može širiti tamnicu samo u dužinu i širinu, a ne i u visinu. Također, algoritam ne može nasumično stvarati NPC likove, i tamnica je uvijek prazna. Međutim zbog svoje jednostavnosti moguće ga je ugraditi u druge klase koje bi bile dodatak i nadogradnja za stvaranje NPC likova i raznih detalja. Zato je ovaj generator tamnica preporučljiv developerima početnicima za izradu jednostavnijih igara.

LITERATURA

[1] https://en.wikipedia.org/wiki/List_of_Unity_games [Pristupljeno: lipanj 2019.]

[2] https://learn.unity.com/ [Pristupljeno: prosinac 2018.]

[3] https://donjon.bin.sh/d20/dungeon/ [Pristupljeno: srpanj 2015.]

[4] https://www.blender.org/support/tutorials/ [Pristupljeno: prosinac 2018.]

[5] https://en.wikibooks.org/wiki/Blender_3D%3A_Noob_to_Pro [Pristupljeno: siječanj 2019.]

[6] https://cgcookie.com/flow/introduction-to-blender/ [Pristupljeno: prosinac .2018.]

[7] https://docs.microsoft.com/en-us/dotnet/csharp/ [Pristupljeno: rujan .2018.]

[8] https://csharp.net-tutorials.com/hr/104/getting-started/introduction/ [Pristupljeno: ožujak .2019.]

[9] http://gamedev.machina.hr/unity-3d-game-engine-tvornica-igara-indie-studija/ [Pristupljeno: lipanj 2019.]

SAŽETAK

Problem dugih i mukotrpnih sati izrade razina u video igrama moguće je riješiti programatski pomoću algoritma za nasumično stvaranje tamnice i izradom 3D modela gradivnih blokova pomoću kojih se izrađuje cjelokupna tamnica, tzv. čelija, u Blenderu. Algoritam generatora tamnica napisan je u programskom jeziku C# za laku primjenu unutar Unity enginea. Uz pomoć Unity enginea, moguće je povezati numeričke podatke rezultata koje stvori generator tamnica sa pripadajućim modelima ćelija, te na kraju testirati rezultate unutar igre.

Ključne riječi: Ćelija, generator tamnica, Unity Engine, Blender, C#, video igre

ABSTRACT

Problem of long and painstalking hours of level design in video games is possible to solve programmaticaly using algorithm for random dungeon generation and by creating 3D models of building blocks, which are used to create the dungeon as whole, so called cells, in Blender. With help of Unity engine, it is possible to logically connect numerical data which are generated by Dungeon Generator with respective 3D models of cells, and in the end testing those results within the game.

Keywords: Cell, Dungeon Generator, Unity Engine, Blender, C#, video games

ŽIVOTOPIS

Filip Garmaz rođen je u Osijeku 26. lipnja 1996. godine. Godine 2011. završava Osnovnu školu Bratoljuba Klaića u Bizovcu. Godine 2015. završava prirodoslovno-matematičku gimnaziju u Osijeku, te iste godine upisuje Elektrotehnički fakultet u Osijeku, danas Fakultet elektrotehnike, računarstva i informacijskih tehnologija, gdje trenutno pohađa treću godinu Preddiplomskog sveučilišnog studija računarstva.

PRILOZI

P 3.1. Generator tamnica, datoteka izvornog koda DungeonGenerator.cs za nasumično stvaranje tamnica

P 4.1. Datoteka glavnog izvornog koda za Unity, Main.cs

Prilozi se nalaze na CD-u.