Korištenje umjetne inteligencije u alatu Unity

Mravić, Filip

Sveučilište u Rijeci – Odjel za informatiku

Jednopredmetna informatika

Filip Mravić

Primjena AI-a u Unityju

Završni rad

Mentor: doc. dr. sc., Marina Ivašić-Kos

Rijeka, 21.9.2018.

ODJEL ZA INFORMATIKU
SVEUČILIŠTE U RIJECI

Rijeka, 15.2.2018.

Zadatak za završni rad

Pristupnik: **Filip Mravić**

Naziv završnog rada: **Korištenje umjetne inteligencije u alatu Unity**

Naziv završnog rada na eng. jeziku: **Using of AI in Unity engine**

Sadržaj zadatka:

Proučiti odgovarajuće alate za dizajn i razvoj računalnih igara. Proučiti i opisati vrste računalnih igara i elemente igre u kojima se koristi umjetna inteligencija.

Istražiti mogućnosti Unity enginea za implementaciju umjetne inteligencije u igri te razviti elemente igre kojima će se demonstrirati njeno korištenje.

Za svaki od primjera korištenja umjetne inteligencije kao što je kao što je generiranje prepreka, kretanje neprijatelja i slično objasniti odgovarajući algoritam i ključne djelove implementacije za ostvarivanje odgovarajuće funkcionalnost objekata u igri.

Mentor

doc. dr. sc. Marina Ivašić-Kos

Voditelj za završne radove

dr. sc. Miran Pobar

Zadatak preuzet: 15.2.2018.

F. Mravić

(potpis pristupnika)

1 SADRŽAJ

2 ZADATAK ZAVRŠNOG RADA

Zadatak ovog završnog rada je prikazati kako se može primijeniti Umjetna Inteligencija u sklopu alata za izradu igrica Unityja. S obzirom da je pojam jako širok, može se koristiti u raznim područjima kao što su matematika, psihologija, filozofija, robotika, statistika, …, ali mi ćemo biti zainteresirani samo u područje računalne znanosti i to specifično: korisnost u alatu Unity.

3 SAŽETAK I KLJUČNE RIJEČI

U ovom završnom radu istražujemo kako možemo pomoću Unityja kreirati aplikacije(razne simulacije ili igrice) koristeći umjetnu inteligenciju. Kroz primjere ćemo se baviti isključivo sa problemom traženja najkraćeg puta do neke točke u prostoru.

U početku govorimo općenito o traženju puta te koji se algoritam najčešće koristi(opis i implementacija), koristimo u primjerima dvije različite vrste grafova: u obliku matrice te u obliku Navigation Mesha.

Kroz primjere ćemo prikazati najjednostavniju implementaciju Navmesha u Unityju, a u kasnijim primjerima pokazujemo što se događa kada imamo više agenata u simulaciji ili ako želimo implementirati situaciju gdje imamo nekoliko agenata koji obavljaju akciju patrole i potrage.

Ključne riječi: Navmesh, A*, Traženje puta, Agent, Prepreka, Unity

4 UVOD

Umjetna inteligencija (eng. Artificial Intelligence, AI) je veoma važan aspekt u izradi današnjih igrica, koja na neki način daje novi život igrici. Koriste se tzv. NPC-evi [1]za interakciju sa igračem, mogu nam pomoći ili nam mogu biti neprijatelji i pokušavaju se ponašati realistično kako je u stvarnom svijetu.

U svijetu umjetne inteligencije entitet koji oponaša ljudsko biće nazivamo agentom. Agent je programiran od strane programera na način da prima informacije koje su važne za određenu funkciju koju je postavio programer i te informacije dobiva pomoću senzora [1], npr. ako agent obavlja funkciju patrole senzori će biti zrake (u Unityju se koristi raycast) te će agent poduzeti nešto ako zrake dođu u kontakt sa igračem.

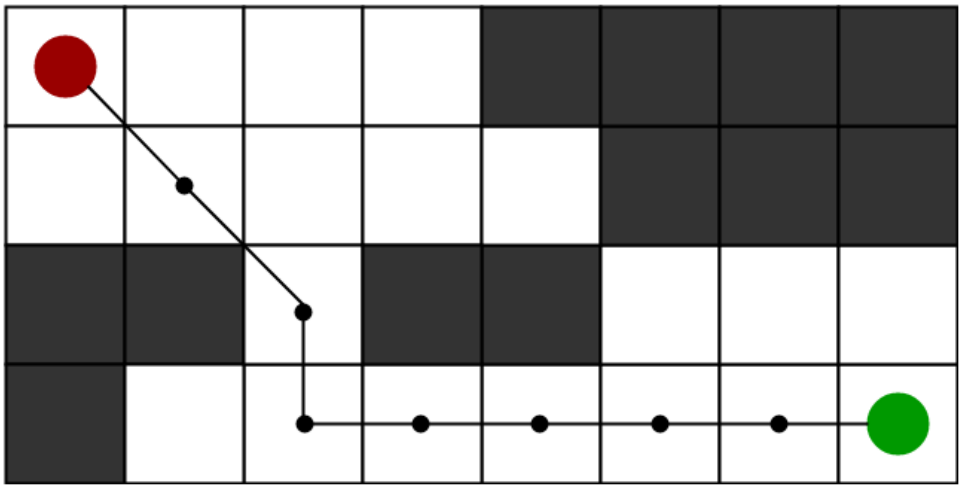
[1] neigriv lik (eng. NPC) je lik u igrici koji nije neprijateljski nastrojen igraču koji igra igricu

5 TRAŽENJE PUTEVA

U ovim projektima ćemo se većinom baviti traženjem puteva od jedne do druge točke. Generalno postoji mnogo algoritama koji se koriste u različitim područjima znanosti no za izradu igrica se najčešće koristi algoritam A* (izgovoreno na eng. kao „A star“) koji je zasnovan na temeljnom algoritmu zvanog kao Dijkstrin algoritam[2].

A* algoritam je nadogradnja na Dijkstrin algoritam, koristi se od 1968.g. kada je grupa od trojice ljudi iz „Stanford Research Institute“ objavila algoritam [2].

Na slici vidimo najkraću putanju od crvene do zelene kružnice.

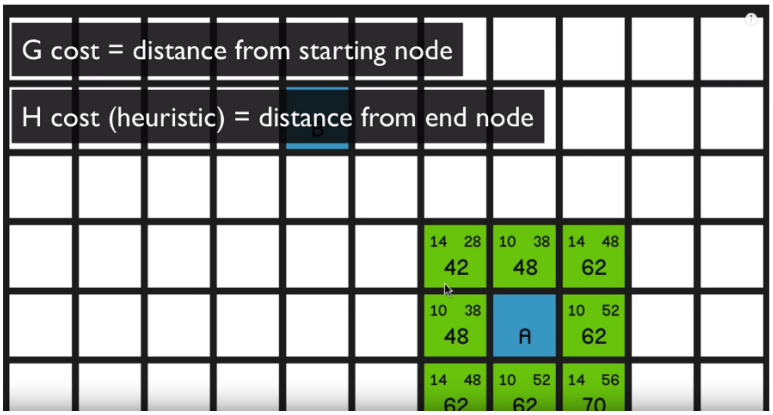


Najkraći put od crvene do zelene kružnice [3]

Algoritam radi na način da računa najmanju vrijednost *f* koja se dobije tako da se zbroje parametri *g* i *h*, gdje g označava udaljenost od početnog čvora do sljedećeg čvora u listi i parametar *h* označava udaljenost od tog sljedećeg čvora do destinacije [4].

Na slici vidimo u gornjem lijevom kutu kvadrata vrijednost parametra *g* i u desnom vrijednost parametra *h*, u sredini imamo ukupnu cijenu odnosno parametar *f*.

[2] Algoritam koji je smislio nizozemski informatičar Edsger W. Dijkstra 1956.g. kao prvi algoritam za traženje najkraćeg puta u grafu(koristio se u početku kako bi se olakšao transport kroz gradove u Nizozemskoj).



Grafički prikaz načina na koji radi A algoritam [5]*

Parametar *g* se može jednostavno dobiti no parametar *h* se može dobiti na 2 načina: možemo ga točno izračunati ili ga možemo aproksimirati.

S obzirom da egzaktno računanje parametra *h* je vremenski zahtjevno (osim ako nema prepreka, tada možemo koristiti euklidsku udaljenost od čvora kojeg gledamo do destinacije) koristi se aproksimacija.

Neki načini koji se koriste za aproksimaciju:

- Manhattanova udaljenost – *h = abs(current_cell.x – goal.x)+abs(current_cell.y - goal.y)* -> koristi se ako se možemo kroz polje kretati u četiri smjera
- Dijagonalna udaljenost – *h = max (abs(current_cell.x – goal.x), abs(current_cell.y - goal.y))* -> koristi se ako se možemo kretati u osam smjerova (slično kao i kralj u šahu)
- Euklidska udaljenost – *h = sqrt((current_cell.x – goal.x) ^2+(current_cell.y - goal.y) ^2)* -> koristi se ako se možemo kretati u bilo kojem smjeru

Kako radi algoritam funkcionira:

1. Inicijaliziramo dvije liste: open (skup čvorova koji nismo procesirali) i closed (skup čvorova koje smo procesirali), ubacujemo početni čvor u open listu te možemo postaviti parametar koji računa trenutnu najmanju udaljenost do sljedećeg čvora: *f*=0
2. U petlji ćemo koristiti varijablu current kao čvor u listi „open“ koji predstavlja najmanji *f*.
3. Izbacujemo čvor current iz liste „open“ te dodajemo u listu „closed“, ako je čvor current jednak destinacijskom čvoru izlazimo iz petlje inače procesiramo svaki susjedni čvor čvoru current i gledamo ako je moguće doći do tog susjeda ili je taj susjed već u listi „closed“, ako je odgovor na to pitanje netočan tada preskačemo tog susjeda i gledamo sljedećeg, inače gledamo ako je udaljenost do tog susjeda manja od onih koje već imamo u „open“ listi ili ako susjed nije u „open“ listi.
4. Postavljamo parametar *f* susjeda trenutnog čvora „current“ te postavljamo koji je roditelj tog susjeda, odnosno od kuda smo došli tog čvora (susjeda).
5. Pitamo se da li je taj susjed u „open“ listi, ako nije ga dodajemo.

Slika prikazuje pseudokod A* algoritma.

```
OPEN //the set of nodes to be evaluated
CLOSED //the set of nodes already evaluated
add the start node to OPEN

loop
 current = node in OPEN with the lowest f_cost
 remove current from OPEN
 add current to CLOSED

 if current is the target node //path has been found
  return

 foreach neighbour of the current node
  if neighbour is not traversable or neighbour is in CLOSED
   skip to the next neighbour

  if new path to neighbour is shorter OR neighbour is not in OPEN
   set f_cost of neighbour
   set parent of neighbour to current
   if neighbour is not in OPEN
    add neighbour to OPEN
```

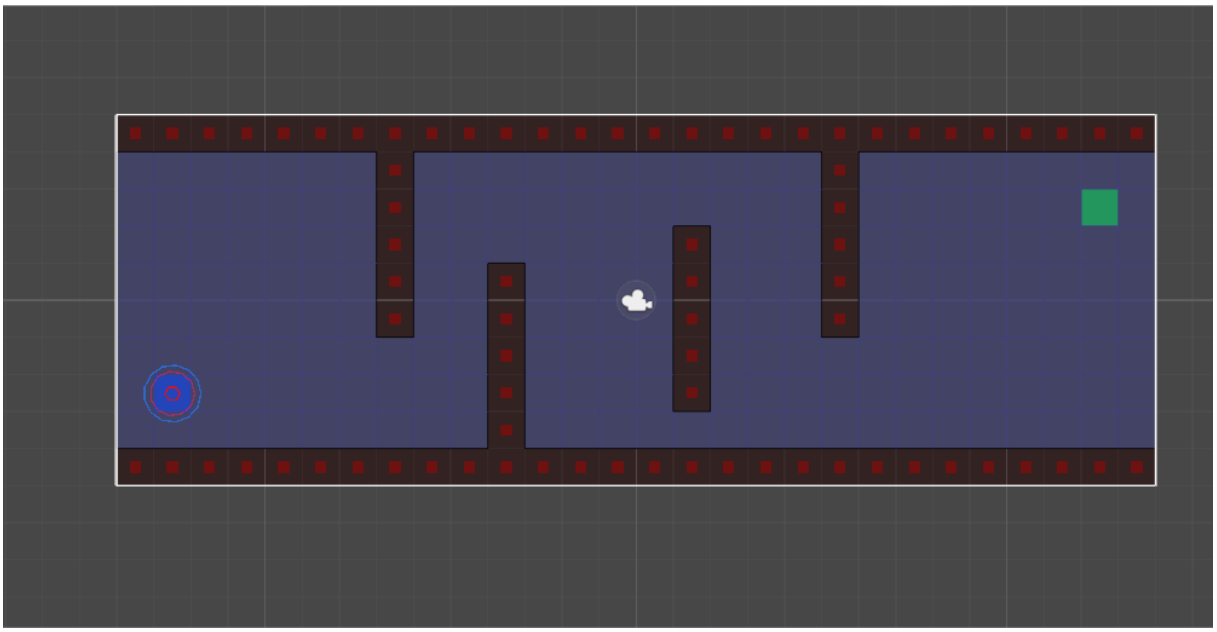
Pseudokod A algoritma [6]*

6 VRSTE GRAFOVA

U našim primjerima ćemo koristiti 2 različite vrste grafova: graf u obliku grida te Navmesh graf.

Kao što ime kaže grid graf ima čvorove u obliku matrice X*Y te se najviše koristi za 2D top-down svijet. Najpoželjniji je ako trebamo ažurirati graf u realnom vremenu, ali gubi performanse ako je matrica prevelika [7].

Na slici možemo vidjeti primjenu grid grafa, crveni kvadratići na preprekama označavaju da nije moguće doći do te pozicije, a plava pozadina označava prostor koji je dostupan prilikom traženja puta (eng. pathfindinga). Izradu te primjenu ovog grafa ćemo kasnije obraditi.



Primjer scene u kojoj se koristi grid graf za traženje najkraćeg puta

Jedan od najkorištenijih vrsta grafova u traženju puta te je ugrađen u Unity za korištenje navigacije: Navmesh graf

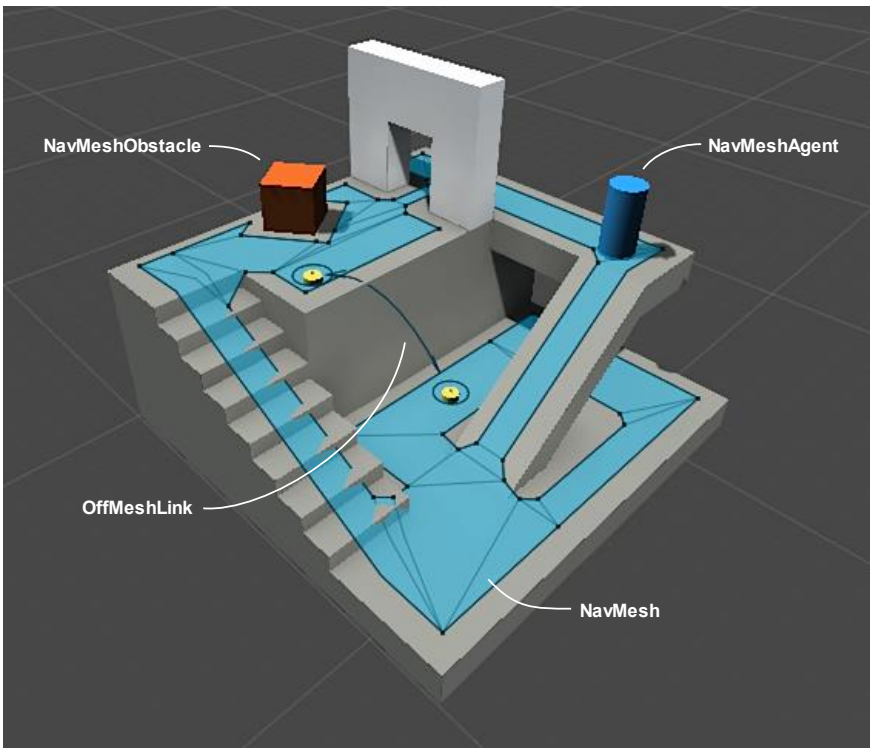
Navmesh (Navigation mesh) je skup 2D poligonskih mreža koje definiraju koja su područja prohodna za agenta. Za razliku od grid grafa ovaj model je poželjan za 3D svijet jer ima puno manje čvorova za procesirati.

7 NAVMESH

Prilikom izrade Navmesha bitno je još znati nešto o sljedećim komponentama [8]:

- Navmesh agent - komponenta Navmesha koja se daje nekom objektu/objektima koja traži najkraći put do cilja izbjegavajući druge agente ako se nalaze na putu(agenti su predstavljeni kao cilindri)
- Navmesh prepreka - komponenta kojom obilježavamo sve prepreke koje agent mora izbjegavati kako bi došao do destinacije na način da Unity "izreže" rupu u Navmeshu
- Off-mesh Link - komponenta sa kojom možemo povezati dvije točke na Navmeshu koje se ponašaju kao kratice jer bi inače put između te dvije točke bio duži, koriste se npr. za skakanje preko jarka ili preko ograde itd.

Slika prikazuje sve prethodno navedene komponente u Unityju.



Scena u kojoj su ugrađene sve najvažnije komponente Unityjeve navigacije [9]

8 IZRADA NAVMESHA

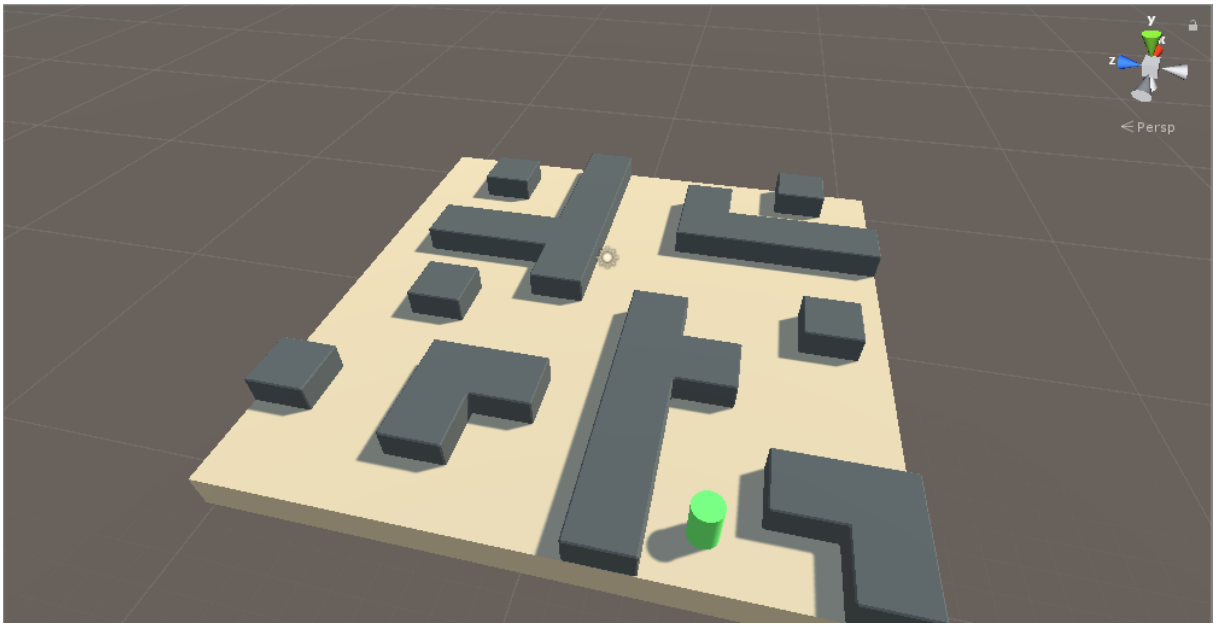
Proces izrade Navmesha u Unityju se zove „Baking“ - Unity prikuplja sve podatke o geometriji objekata koji su na sceni.

U sljedećem primjeru ćemo prikazati kako se na najjednostavniji može izraditi Navmesh te kako traženje puta funkcionira.

8.1 SIMULACIJA 1:

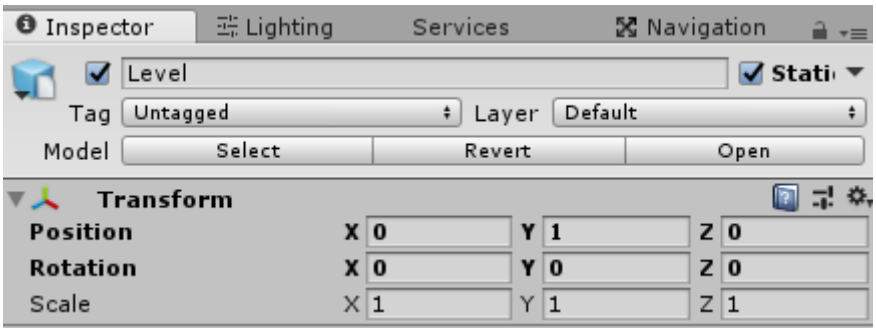
Kratka simulacija će se sastojati od igrača u obliku cilindra koji pomoću lijevog klika bilo gdje na sceni (unutar Navmesh površine) upravlja cilindrom do zadanog cilja.

Prije samog kreiranja Navmesha potrebno je pripremiti sve objekte koji će se koristiti. Scena je kao na slici:



Početna scena sa objektima

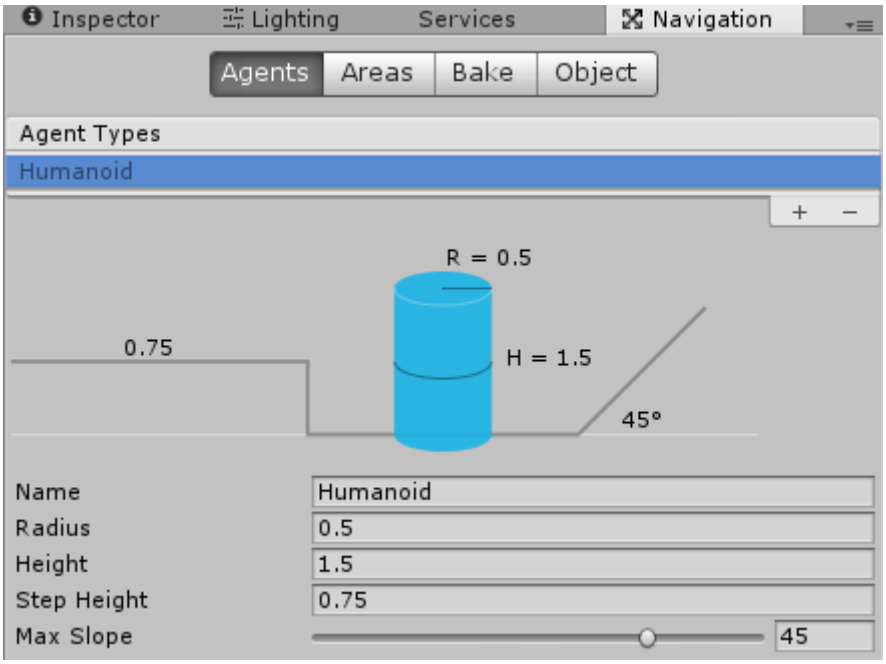
Kada smo pripremili scenu sa objektima koje ćemo koristiti u izradi potrebno je obilježiti koji će objekti biti statični tako da Unity zna koji su objekti prepreke te ih treba izbjegavati. Kliknemo na prazan (eng. empty, u daljnjem tekstu empty) objekt „Level“ koji sadrži tlo i prepreke te ih postavimo na „Navigation Static“ kao na slici.



Postavljamo objekt da bude navigacijski statičan

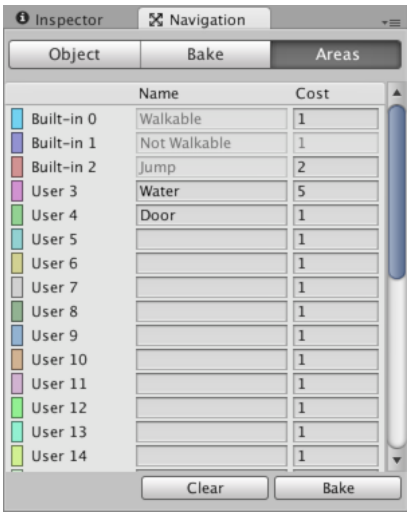
Sad je sve spremno za izradu Navmesha, idemo na Window->AI->Navigation i otvoriti će nam se prozor pored inspektora (eng. Inspector) i osvjetljenja (eng. Lightning). Unity nam nudi nekoliko „podprozora“ unutar Navigation prozora:

Agents: možemo kreirati tip agenta koji će se koristiti za izradu tog navmesha. Nakon što smo odabrali tip agenta možemo namjestiti razne parametre kao što su radijus (koliko blizu može biti od prepreke), visina (kroz koje prostore može proći), maksimalni nagib površine na koju agent može ići i visinu stepenica na koje se može popesti. Agents podprozor je prikazan na slici.



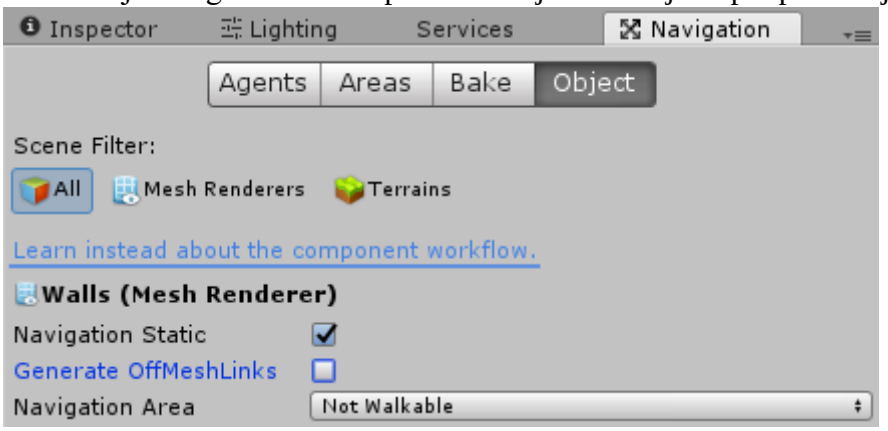
Prikazan je podprozor Agents unutar prozora Navigation

Areas: možemo postaviti prilagođene površine koje koristimo u igrici te promijeniti cijenu površine koju će algoritam(A*) uzeti u obzir. Areas podprozor je prikazan na slici.



Prikazan je podprozor Areas unutar prozora Navigation

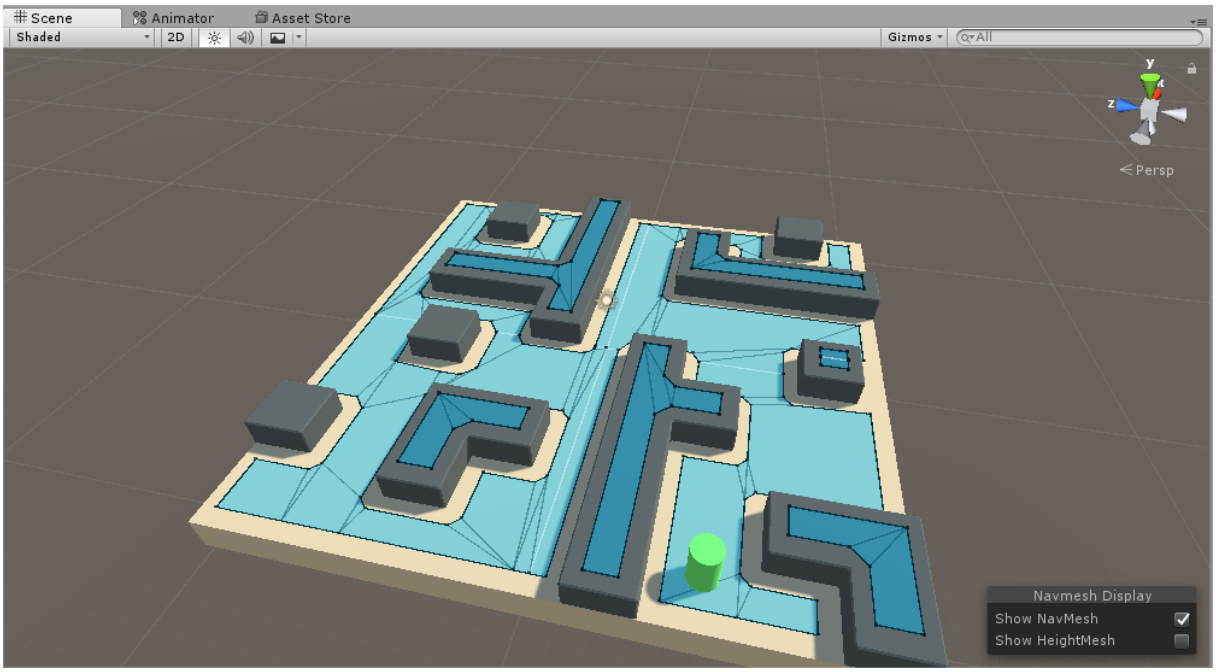
Object: nudi opcije za objekt na sceni kao što je postavljanje objekta da je statičan za navigaciju ili da nije moguće hodati po tom objektu. Object podprozor je prikazan na slici 10.



Prikazan je podprozor Object unutar prozora Navigation

Bake: u ovom prozoru možemo vidjeti detalje vezane za našeg agenta.Klikom na „Bake“ započinjemo proces kreiranja Navmesha.

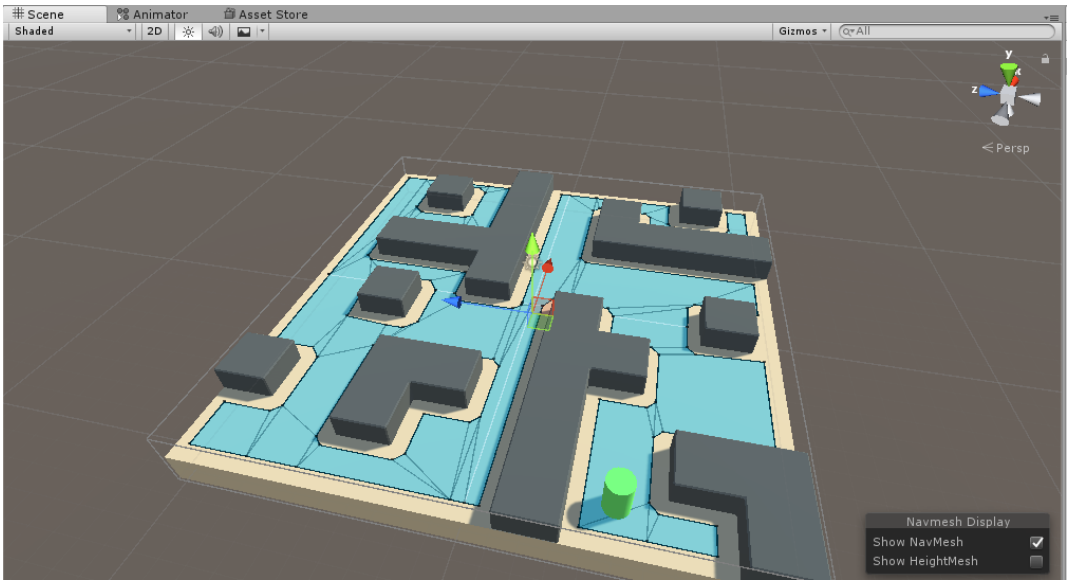
U našem primjeru ako pritisnemo na Bake dobiti ćemo Navmesh kao na slici:



Izgled scene nakon procesa koji se zove „Bake“

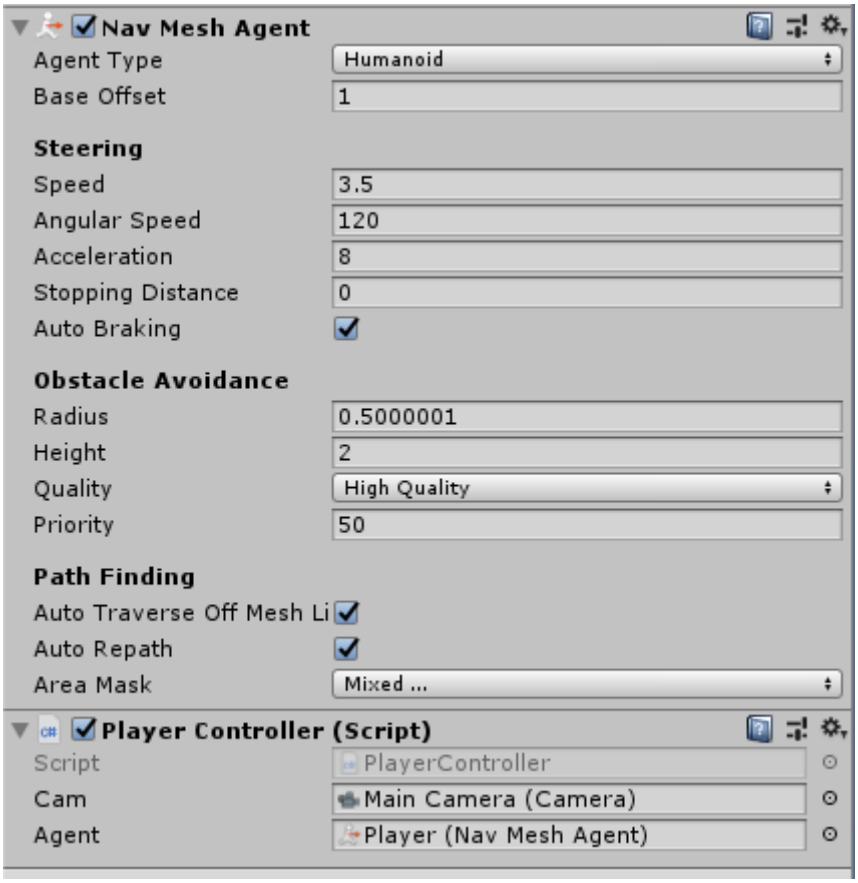
Plave površine označuju prostor po kojima se agent može kretati. Prepreke su „izrezale“ dodatni prostor oko sebe tako da će ih agent pokušati zaobilaziti kada im se približi. Na slici se može zamijetiti da se plava boja nalazi i na zidovima što bi značilo da se agent može kretati i po zidovima. Kako bismo ovu logičku pogrešku otklonili postavljamo sve zidove na „Not Walkable“ u Object podprozoru.

Konačni Navmesh će izgledati kao na slici:



Prikazuje kreiran Navmesh u sceni

Kako bismo manipulirali kretanjem agenta moramo napraviti skriptu koja će omogućiti da se agent kreće do pozicije na tlu na koju je igrač kliknuo. Objektu Player dodajemo 2 komponente: Navmesh Agent te C# skriptu koju možemo nazvati „PlayerController“.Prikazano je na slici.



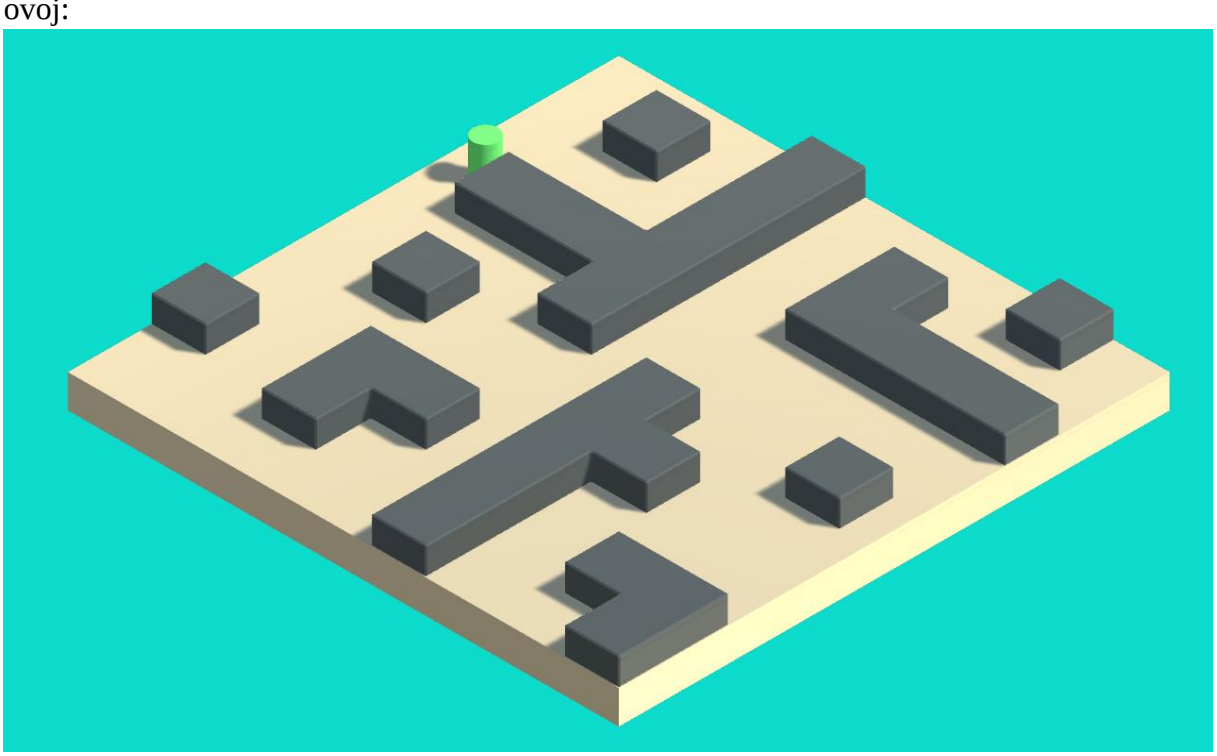
Komponente vezane na objekt Player

Skripta će koristiti biblioteku UnityEngine.AI. Imati ćemo 2 javne(eng. public, nadalje samo public) varijable; referencu na kameru te referencu na agenta. Zatim ćemo izbrisati *Start()* metodu jer nam neće trebati. U *Update()* funkciji provjeravamo je li igrač kliknuo, ako je onda bacamo zraku na poziciju gdje je kliknuo, ako je zraka pogodila tlo odnosno ako je igrač kliknuo negdje na tlo onda će se agent početi micati prema toj poziciji. Skripta je prikazano na slici:

```
using UnityEngine;
using UnityEngine.AI;

public class PlayerController : MonoBehaviour {

    public Camera cam;

    public NavMeshAgent agent;

    // Update is called once per frame
    void Update () {
        // ako se pritisne lijevi klik
        if (Input.GetMouseButtonDown(0))
        {
            Ray ray = cam.ScreenPointToRay(Input.mousePosition); // bacamo zraku od kamere do pozicije miša
            RaycastHit hit; // informacije o tome što je zraka udarila

            if(Physics.Raycast(ray, out hit)) // ako je zraka udarila objekt koji ima "Collider" na sebi (u ovom slučaju će to biti tlo)
            {
                agent.SetDestination(hit.point); // postavi destinaciju agenta na točku gdje je zraka udarila
            }
        }
    }
}
```

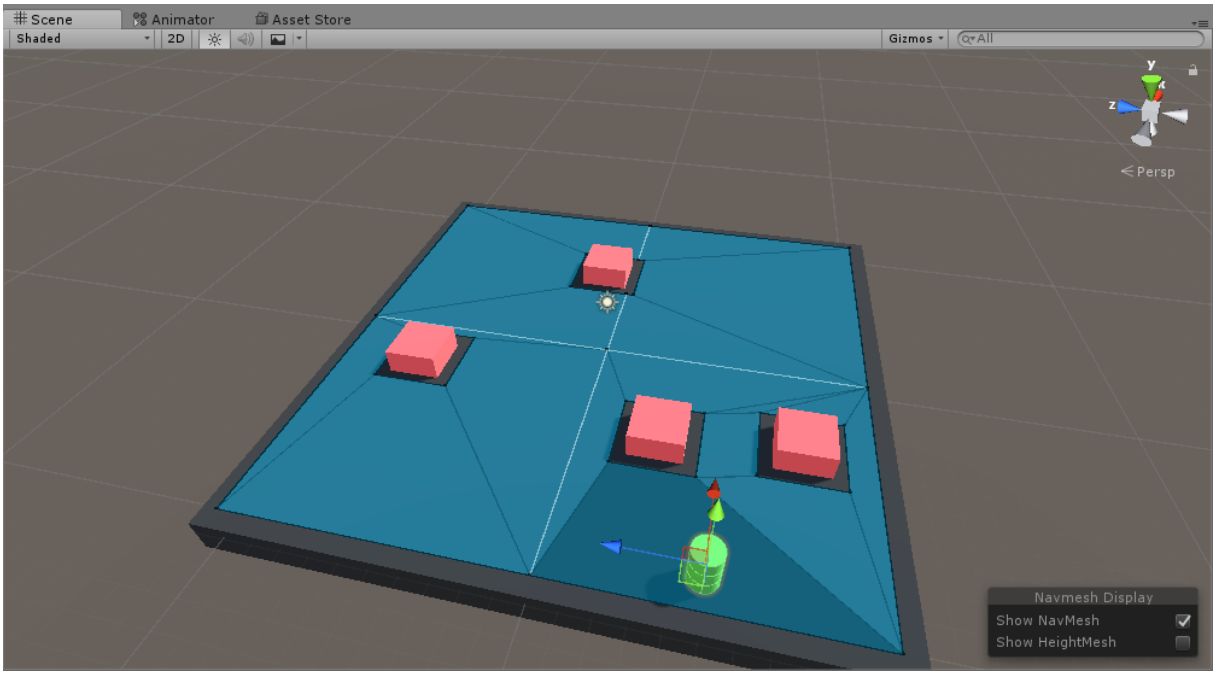
Primjer koda koji se koristi u ovoj simulaciji

Preostao je još jedan korak; s obzirom da funkcija *Physics.Raycast()* vraća informaciju o tome što je zraka pogodila ili zapravo s kojim se tijelom sudarila (eng. collider, nadalje samo collider) moramo kreirati kompontentu „Box Collider“ za naše tlo. Nakon toga će se agent slobodno kretati po površini tla do pozicije na koju smo kliknuli. Simulacija bi trebala izgledati slično ovoj:



Primjer gotove simulacije

Navmesh se može mijenjati u realnom vremenu kao što je vidljivo u sljedećem primjeru. Pomoću skripte koja fluidno miče prepreke u prostoru možemo vidjeti da se cijelo vrijeme se Navmesh prilagođava njihovom položaju, kao što je prikazano na slici:



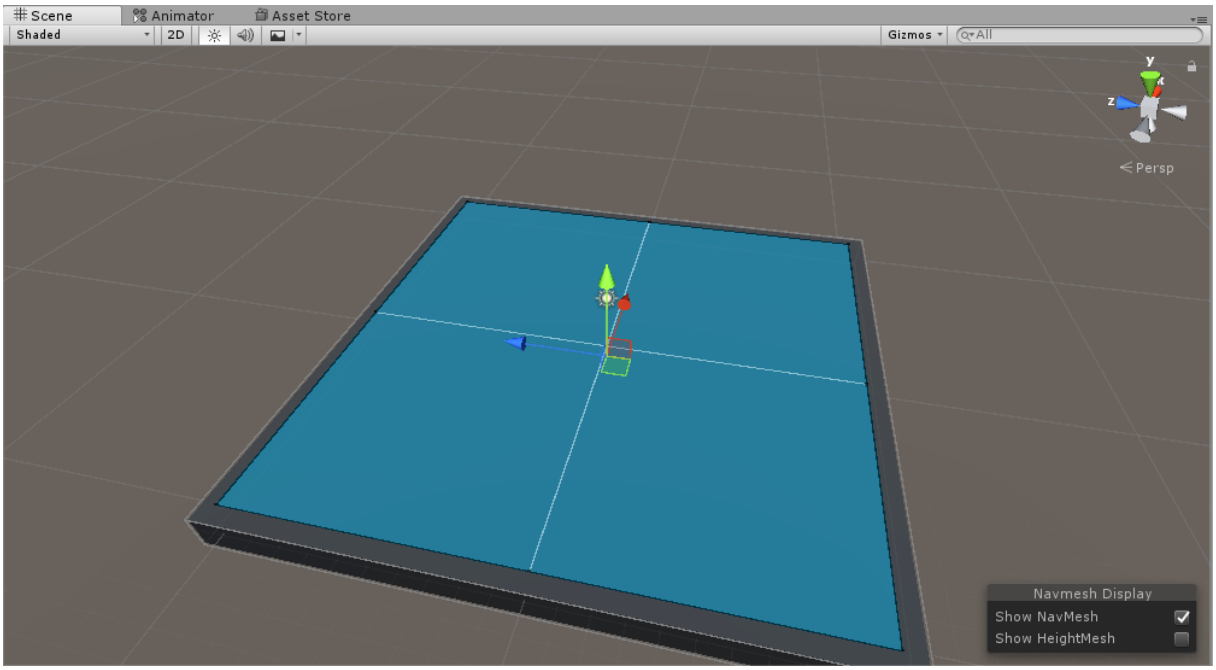
Prikaz scene u kojoj objekti se dinamički miču te time se mijenja Navmesh

U sljedećem primjeru možemo vidjeti da se Navmesh može kreirati u skripti, no trebamo koristiti Unityjev paket koji ne dolazi u standardnom paketu prilikom instalacije. Može se skinuti preko: https://github.com/Unity-Technologies/NavMeshComponents

NavMesh Components paket koristi 4 različite komponente[10]:

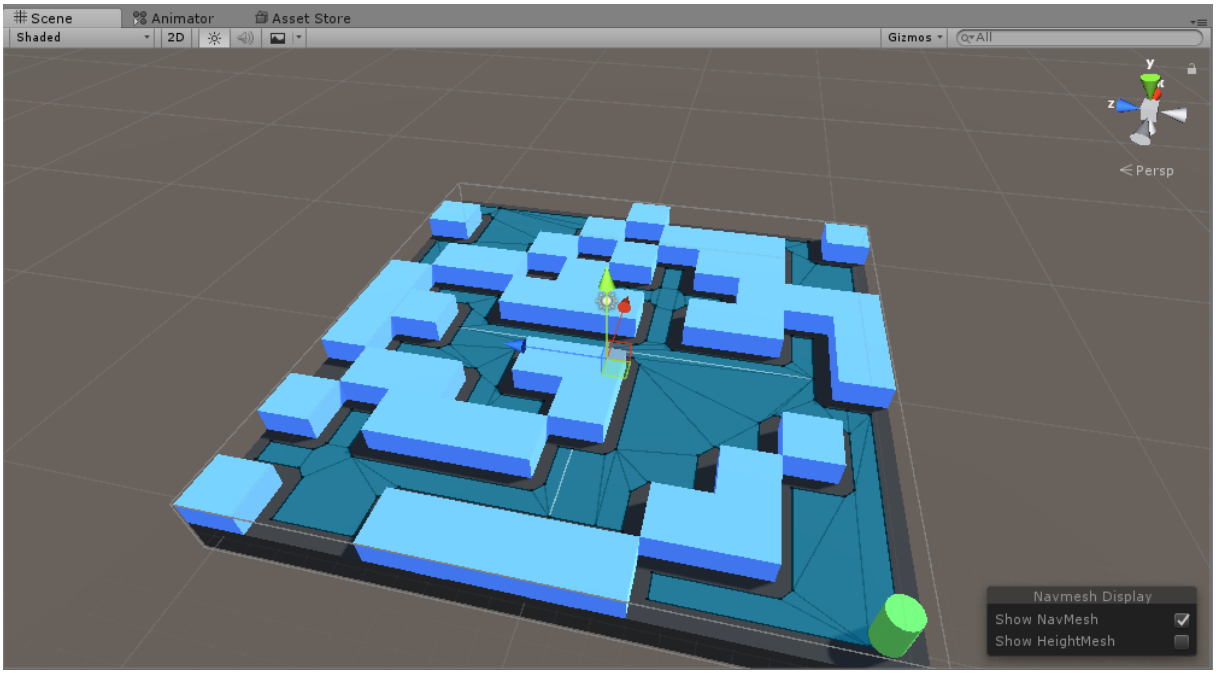
- Nav Mesh Surface - koristi se izgradnju Navmesh površine za jednog agenta
- NavMesh Modifier - koristi se za promjenu ponašanja nekog objekta (npr. možemo postaviti da se do tog objekta može doći samo skokom ili da ne možemo hodati po tom objektu)
- NavMesh Modifier Volume - koristi se za promjenu ponašanja nekog prostora u svijetu (npr. možemo označiti prostor na nekom objektu gdje ne smijemo ići)
- NavMesh Link - Koristi se kao link između dvije lokacije na Navmeshu (npr. ako možemo skočiti na neku platformu)

Mi ćemo koristiti komponentu Nav Mesh Surface za izradu Navmesha, klikom na Navmesh objekt te „Bake“ dobijemo praznu površinu. Iako u skripti kreiramo novi Navmesh moramo prvo kreirati Navmesh na praznoj površini jer Unity neće moći postaviti agenta na Navmesh te se neće moći kretati po Navmeshu. Scena bi trebala biti pripremljena kao na slici.



Izrada Navmesha za teren

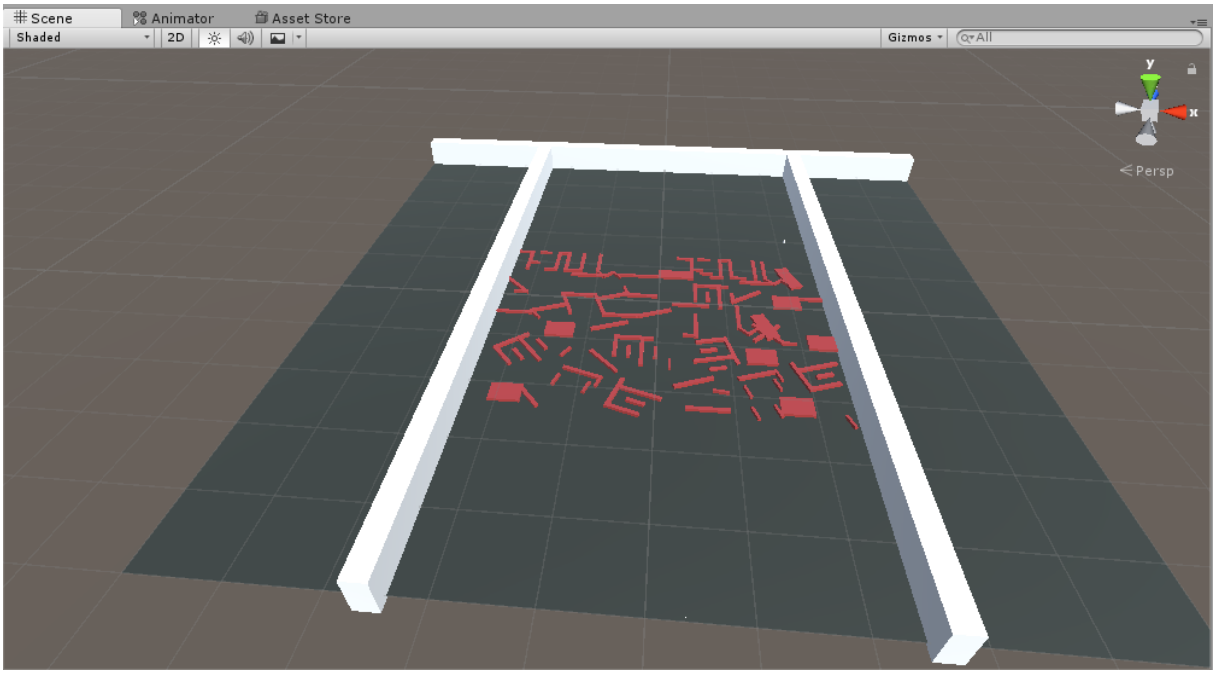
U skripti LevelGenerator imamo već sve spremno za generiranje objekata ali moramo izgraditi novi Navmesh s obzirom da će se promijeniti prilikom poziva funkcije *GenerateLevel()*.To radimo na način da u glavnoj klasi dodamo public varijablu tipa Nav Mesh Surface, možemo je nazvati „surface“. Navmesh gradimo na način da jednostavno nakon funkcije *GenerateLevel()* pozovemo funkciju *surface.BuildNavMesh()*. Nakon toga još moramo povezati referencu Navmesha na skriptu. Pri uspješnom pokretanju programa dobiti ćemo nešto kao na slici:



Prikaz prilagodbe Navmesha kada se izgenerira novi level

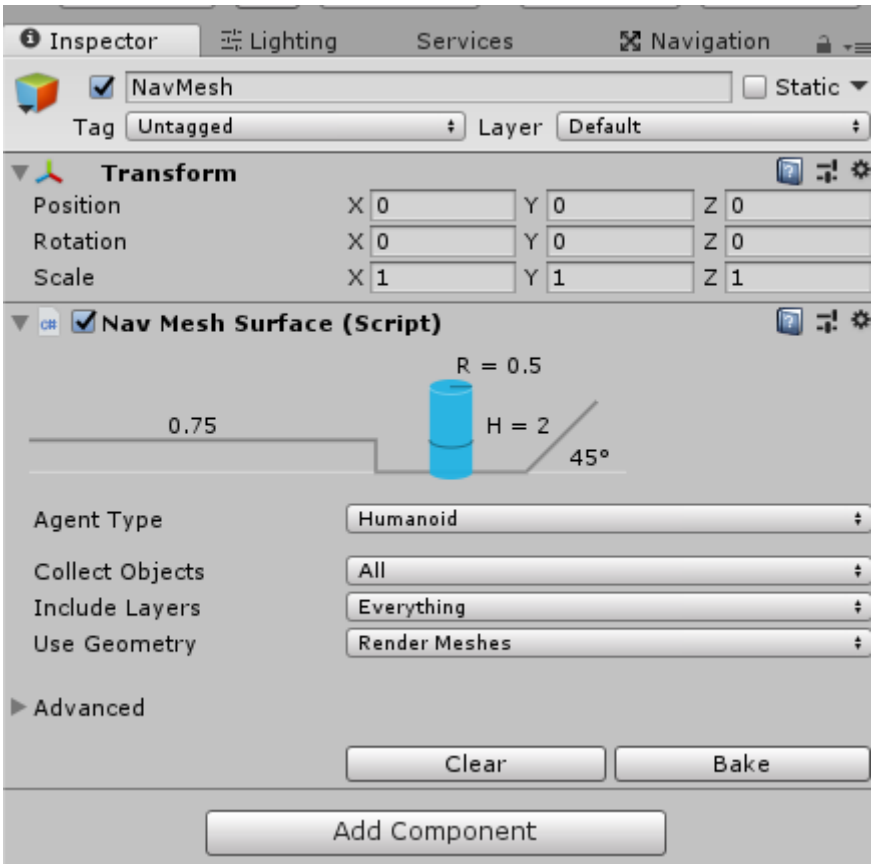
8.2 SIMULACIJA 2:

U sljedećoj simulaciji ćemo prikazati kako možemo generirati na tisuće agenata te ih poslati da prolaze kroz labirint pun prepreka do cilja. Koristiti ćemo NavMesh Components koje se može skinuti na Github linku. U ovom primjeru možemo primijetiti da s obzirom da su objekti preveliki za izgradnju Navmesha po standardnom receptu (označiti koji su objekti statični te onda Navigation->Bake), moramo iskoristiti komponentu Nav Mesh Surface. No prvo izradimo okoliš po kojem će se naši agenti kretati te što će sve izbjegavati, prikazano na slici:



Prikaz svih objekata korištenih u sceni

Nakon toga možemo kreirati našeg agenta po kojem će se ostali agenti generirati (oni će biti kopija njega). U ovom primjeru naš agent ima oblik kapsule (eng. capsule) koji je pozicioniran na gornjoj strani terena. Zatim izrađujemo Navmesh na način da kreiramo Empty objekt te dodamo komponentu Nav Mesh Surface te stisnemo „Bake“, prikazano na slici.



Dodajemo Nav Mesh Surface komponentu empty objektu

Zanimljivo je primijetiti da je duljina trajanja izrade Navmesha ovim procesom trajalo oko 5 sekundi dok za izradu Navmesha bez korištenja NavMesh Components tj. u prozoru Navigation pa klikom na Bake proces traje 1-2 sata kao što vidimo na slici:

Dužina trajanja procesa „Bake“ ako ne koristimo Nav Mesh Components paket

Preostalo je još izraditi skriptu koja će generirati agente te ih poslati kroz labirint do određene destinacije (u ovom slučaju će to biti kocka koja se nalazi na krajnjem donjem rubu površine).

Kreiramo Empty objekt kojeg možemo nazvati „AgentGenerator“. Agente ćemo generirati u obliku matrice što znači da ćemo koristiti dvije for petlje, koristimo dvije public varijable za stupac i redak koje možemo kasnije mijenjati u Inspektoru te još dvije public varijable za razmak između agenata po osi X i Z. Također će nam još trebati 2 reference: objekt po kojem ćemo kopirati(to će biti naša kapsula) i pozicija destinacije kocke.

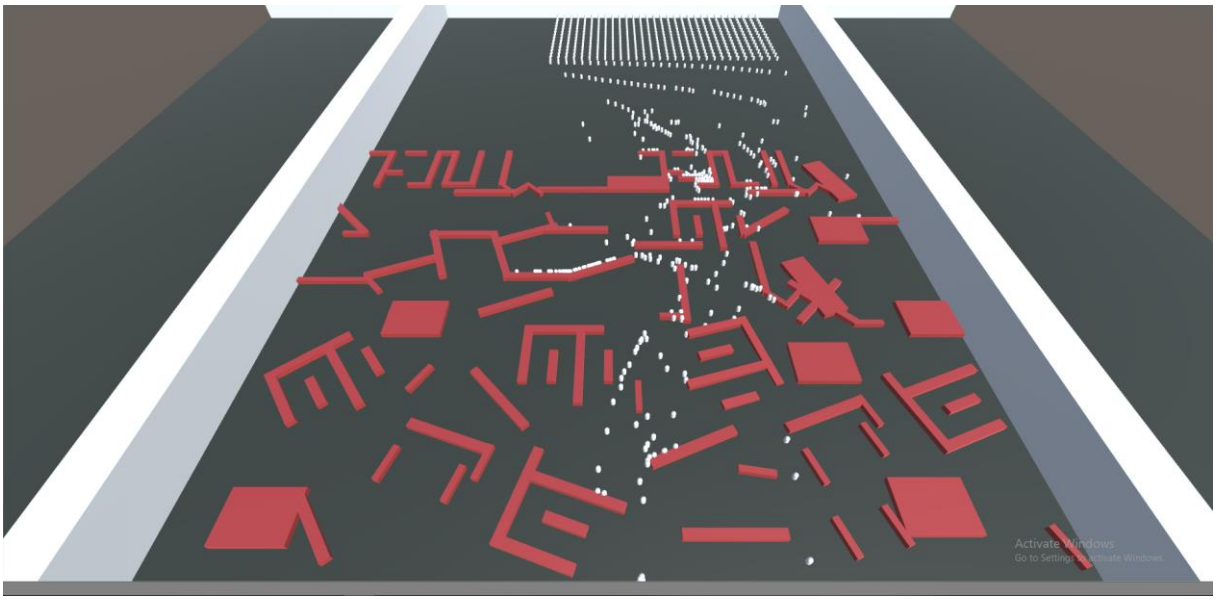
Koristiti ćemo samo funkciju *Start()* jer želimo generirati čim se program pokrene, a ne kasnije.Unutar glavne funkcije imamo referencu na kopirani objekt preko kojeg ćemo pozivati funkcije. U for petlji pozivamo funkciju *Instantiate()* sa parametrima te pridružujemo tu vrijednost varijabli „go“. Nakon toga možemo dodati komponentu NavMesh Agent (ovo smo

inače radili u inspektoru ali to više nije moguće jer agente sad generiramo). Još samo trebamo pozvati funkciju SetDestination() te smo povećamo brzinu agenata na da se kreću brže.

```
public class AgentGenerator : MonoBehaviour {

    public int cols = 30;
    public int rows = 30;
    public float gapX = 10f;
    public float gapZ = 10f;

    public GameObject prefab;
    public Transform target;

    void Start()
    {
        GameObject go;
        for (int i = 0; i < rows; i++)
        {
            for (int j = 0; j < cols; j++)
            {
                go = Instantiate(prefab, new Vector3(213f -j * gapX, 2.3f, (199f + i * gapZ)), Quaternion.identity) as GameObject;
                go.AddComponent<NavMeshAgent>();
                go.GetComponent<NavMeshAgent>().SetDestination(target.position);
                go.GetComponent<NavMeshAgent>().speed = 300;
            }
        }
    }
}
```

Primjer koda koji je korišten u ovom primjeru

Prilikom pokretanja možemo uočiti da se agenti jako teško miču kad su u grupama jer moraju izbjegavati druge agente i ostale prepreke kao što možemo vidjeti na slici.

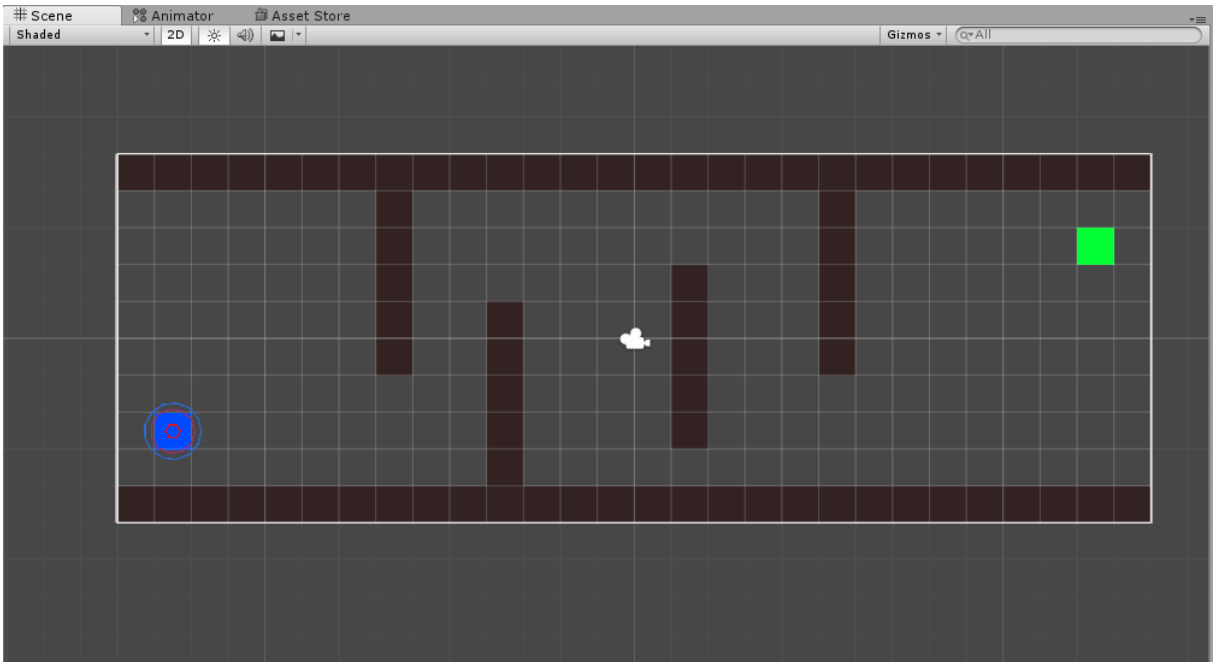


Mnoštvo agenata pokušava doći do cilja kroz labirint

9 2D TRAŽENJE PUTEVA

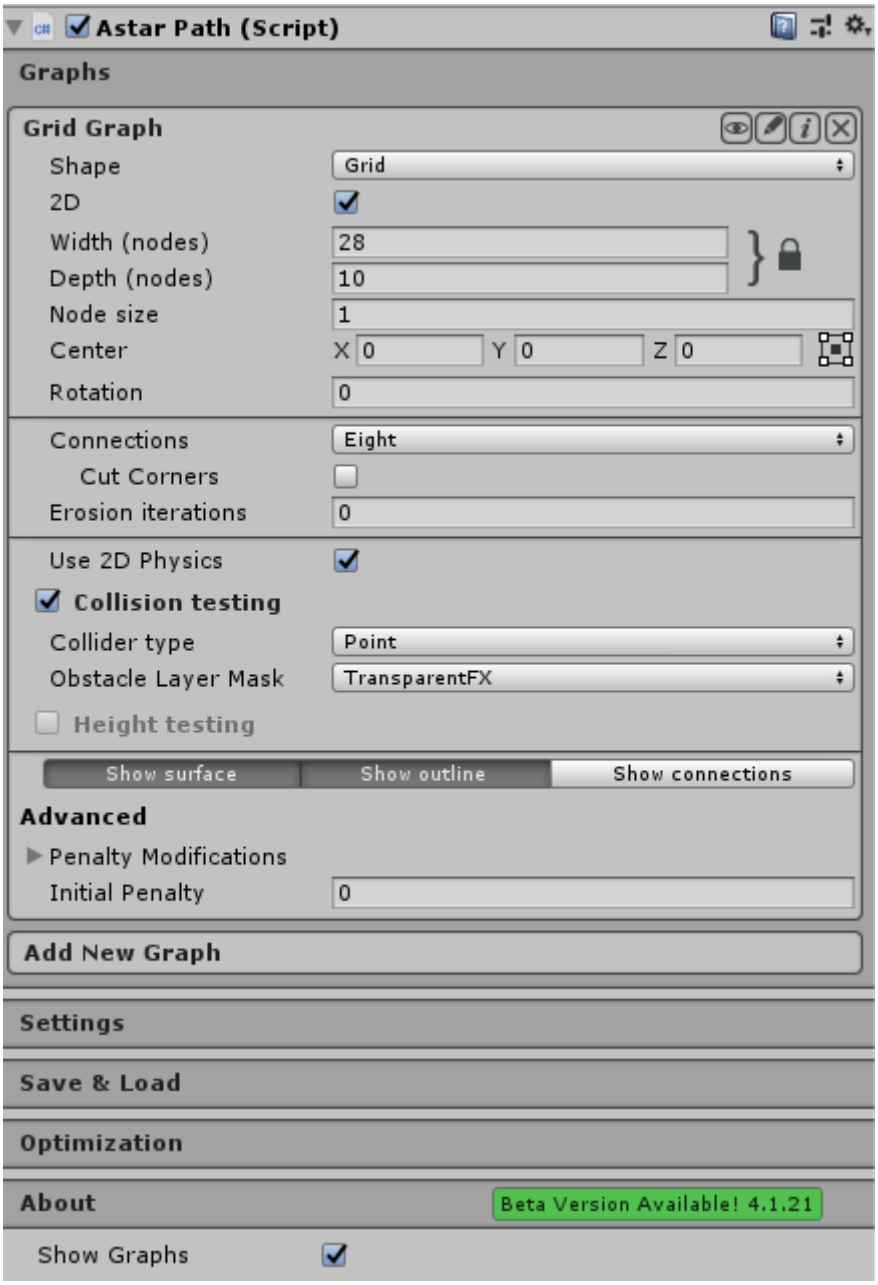
U sljedećem primjeru ćemo prikazati na koji način možemo primijeniti traženje puteva u 2D svijetu. Koristiti ćemo dodatni paket: A* Pathfinding Project koji nudi više opcija nego Unityjev standardni paket. A* Pathfinding Project se može preuzeti sa: https://arongranberg.com/astar/download

Prvo što trebamo kreirati je okolinu koju će A* algoritam procesirati. Kreiramo objekt Grid (2D Object->Grid) koji služi sa lakše manevriranje sa objektima unutar grida kao što je npr. kvadrat koji će se sada micati za točno jedno polje na gridu. Zidove i prepreke ćemo kreirati unutar grida tako da idemo na 2D->Sprite, za svaki kvadrat u Sprite Renderer komponenti označimo za Sprite: „square“ te dodamo komponentu Box Collider kako bi se ponašala kao prepreka. Nakon toga nam treba agent te oznaka cilja(u ovom slučaju je to zeleni kvadrat).Primjer scene nakon što je postavljena okolina dan je na slici:



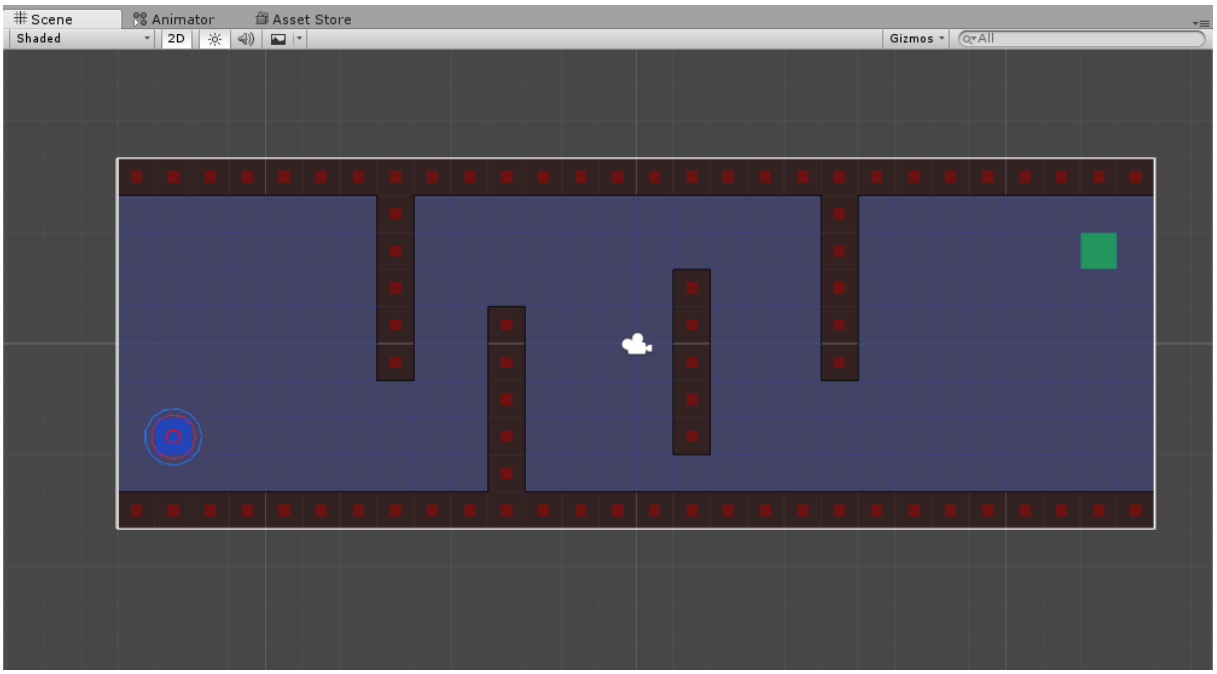
Prikaz scene sa postavljenim objektima u prostoru

Sada kreiramo Grid graf koji će A* algoritam procesirati. To radimo na način da kreiramo Empty objekt i možemo ga nazvati „Astar“ te dodajemo skriptu „Pathfinder“ kao komponentu; pod Graphs dodajemo Grid graph te ćemo promijeniti nekoliko stavki pod opcijama. Postavljamo da je graf 2D te postavljamo koliko će imati čvorova naš graf, u našem slučaju će to biti 28x10 dimenzija. Koristimo 2D fiziku i Collider type Point. Sloj maske na kojem se nalaze prepreke moramo postaviti na nešto drugo tako da ćemo označiti sve prepreke te ih postaviti na neki drugi sloj, npr. „TransparentFX“ te ćemo u izborniku Obstacle Layer Mask označiti taj sloj:



Koristimo Pathfinder komponentu kao skriptu za empty objekt

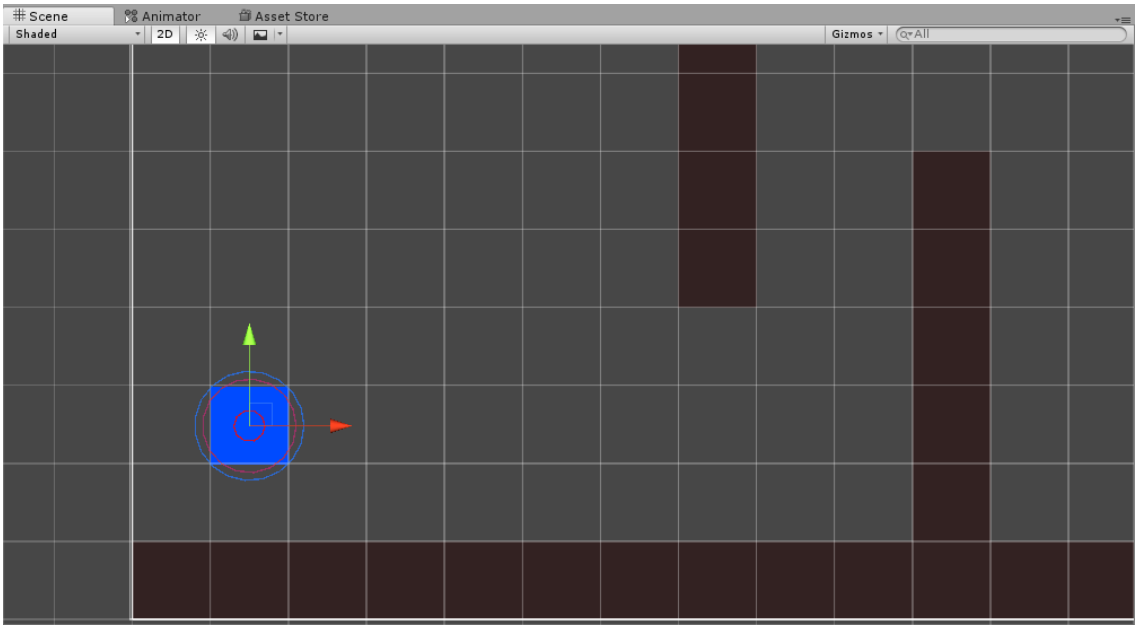
Kada pritisnemo na gumb „Scan“ algoritam procesira sve ulazne informacije te na sceni možemo vidjeti kako će to izgledati. Mali crveni kvadratići označuju prostor gdje se agent ne može kretati odnosno tamo se nalaze prepreke, dok je ljubičasta površina mogući prostor kretanja. Scena će nam izgledati kao što je prikazano na slici.



Scena prilikom uspješnog skeniranja prostora

Zadnje što je preostalo je postaviti plavi kvadrat kao agenta te reći agentu gdje mu je cilj. To radimo na način da klikom na našeg agenta (plavi kvadrat) dodajemo dvije komponente: AI Path koji dodaje dodatnu skriptu „Seeker“. Te dvije skripte omogućuju podešavanje raznih parametara vezane za kretanje agenta kao što je brzina agenta,akceleracija,brzina rotacije,…

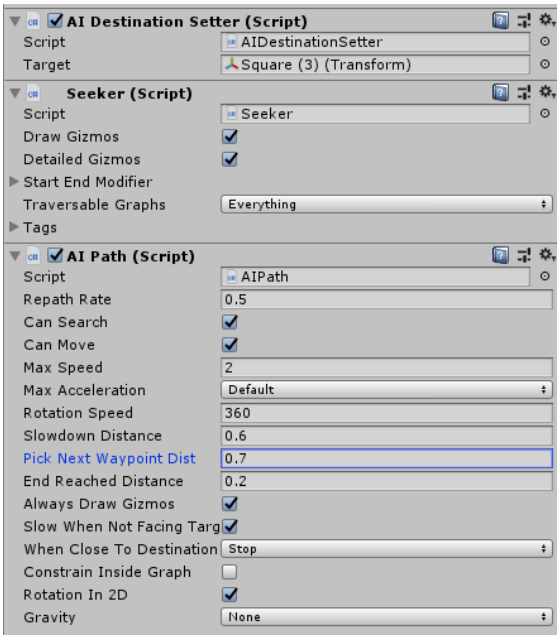
Moramo promijeniti nekoliko opcija u inspektoru na skripti AI Path: Pick Next Waypoint Dist mijenjamo tako da kružnica otprilike opisuje kvadrat. Inače ako je premala vrijednost agent neće točno pratiti putanju do destinacije, a ako je prevelika će pokušati čak i prelazi preko prepreka. Na slici vidimo plavu liniju koja označuje Pick Next Waypoint Dist te crvenu liniju koja je granica zaustavljanja prije dolaska do cilja „End Reached Distance“.



Prikaz plave i crvene kružnice koja upisuje ovaj objekt

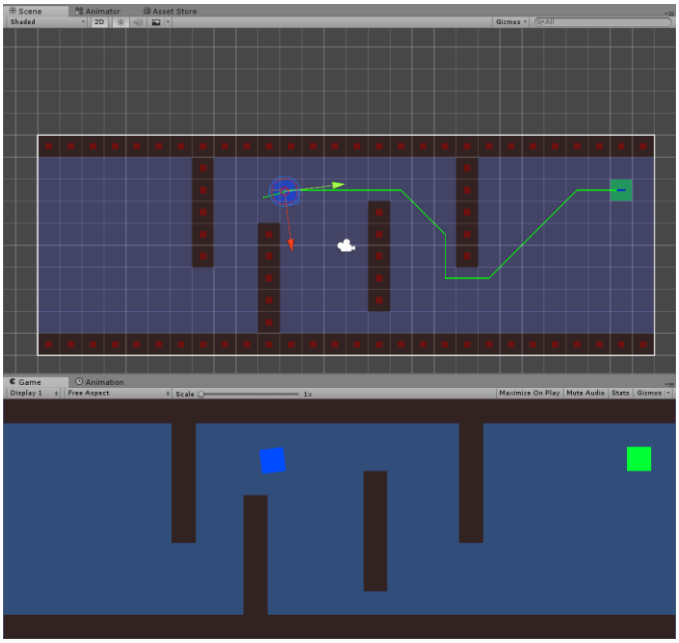
Dodajemo još i opciju 2D rotation te postavljamo da nema gravitacije u ovoj simulaciji.

Sljedeća komponenta koja je važna je AI Destination Setter koja jednostavno određuje koja je destinacija odabranog agenta. Sve bitne komponente su vezane na agenta te su prikazane kao na slici.



Prikaz svih komponenata odnosno skripti u izradi ove simulacije

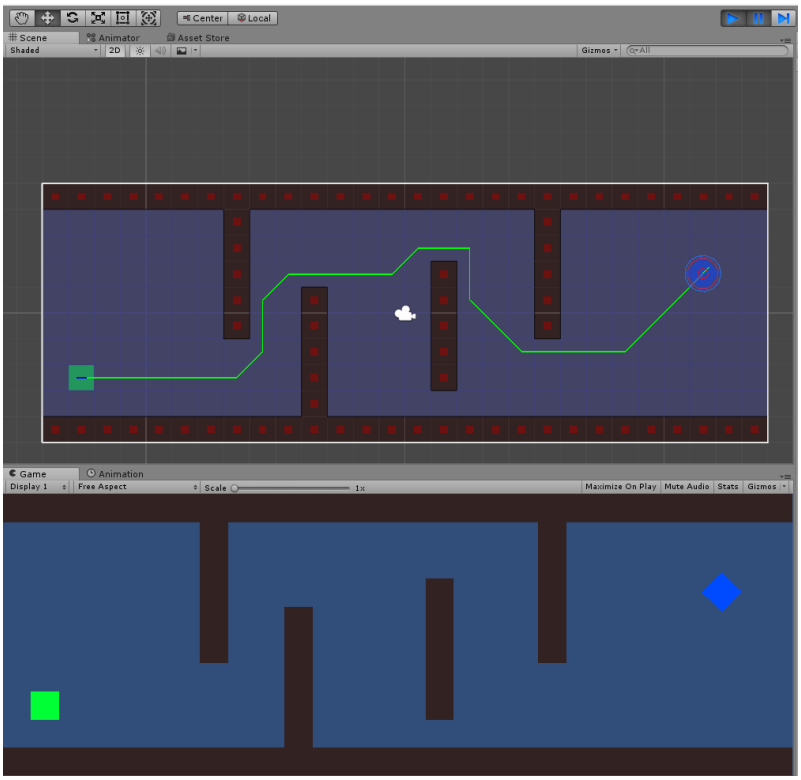
Kada smo gotovi sa mijenjanjem postavki možemo pokrenuti simulaciju. Na slici imamo 2 pogleda na simulaciju, onaj u sceni koji prikazuje putanju do destinacije zelenom linijom i onaj kakvi bi imali u igricama.



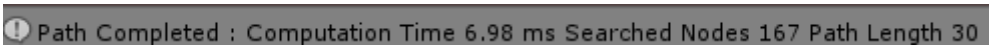
Scenski pogled i pogled u igrici

Također možemo usporediti što će se dogoditi ako zamijenimo početnu i završnu poziciju plavog (agent) i zelenog kvadrata (destinacija). Na sljedećim slikama možemo vidjeti da se put

malo promijenio i to je zato jer će agent pokušati biti što bliže preprekama kada ih zaobilazi. Duljina puta u oba slučaja je jednaka no samo je broj čvorova uračunat u algoritam različit.

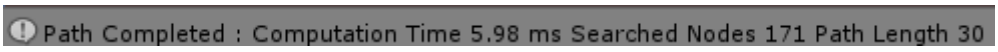


Prikaz najkraćeg puta kada zamijenimo start i cilj



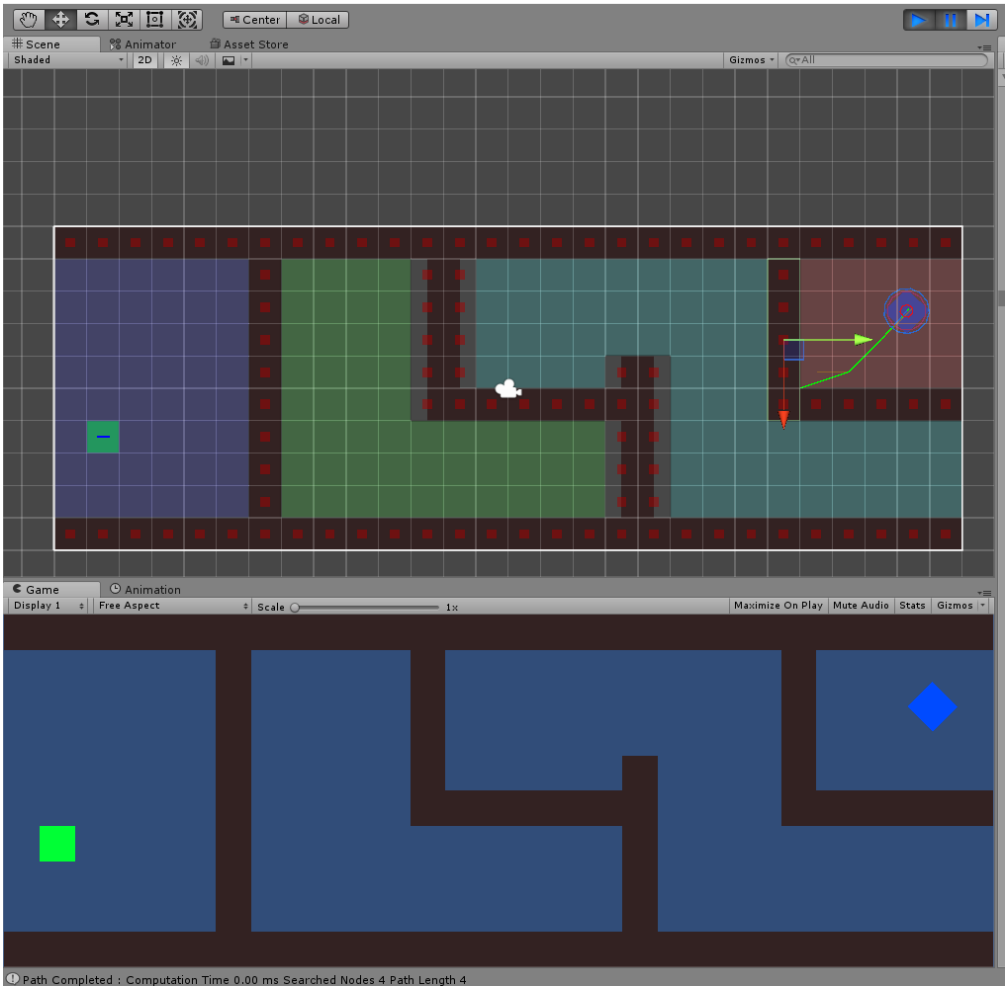
Statistika o putu kada se zamijeni start i cilj

Za usporedbu na sljedećoj slici vidimo statistiku o tome kada agent započinje put u donjem lijevom kutu odnosno kako smo u početku zamislili.



Statistika o putu kada nije zamijenjen start i cilj

Možemo postaviti pitanje što se događa kada nije moguće doći do cilj. U tom slučaju će agent pokušati doći do najbliže moguće točke koja nije prepreka. Ako se dogodi da su neka područja obstruirana odnosno prepriječena možemo vidjeti da nas A* Pathfinding Project paket obavještava o tome tako da oboja određena područja kao što možemo vidjeti na slici:

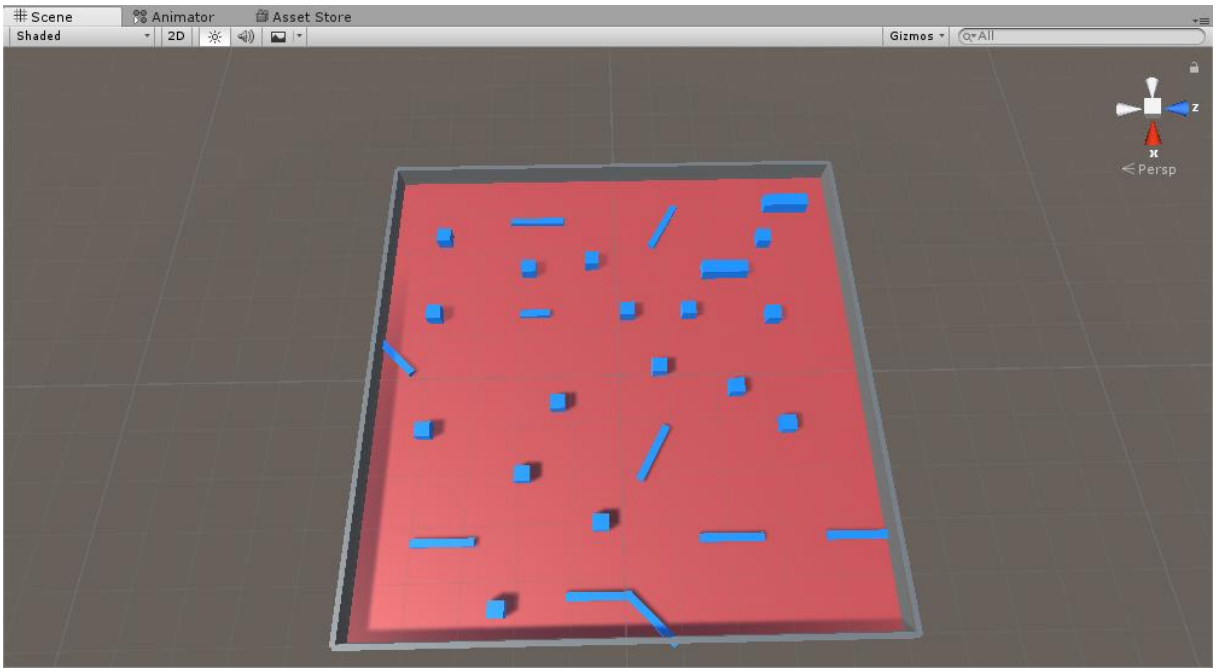


Svako obstruirano područje iz kojeg nije moguće nigdje ići je obojeno različitom bojom, agent pokušava doći do najbliže točke

10 SIMULACIJA: PATROLA I POTJERA

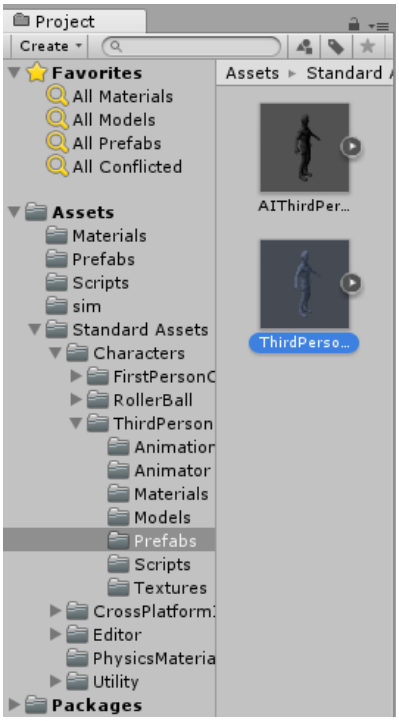
U sljedećoj simulaciji ćemo napraviti simulaciju u kojoj igrač pokušava izbjeći 4 agenata koja patroliraju na određenim putevima. U početku ćemo napraviti da agent zanemari vidno polje odnosno ako je igrač na određenoj udaljenosti od agenta tada će započeti potjera. Kasnije ćemo dodati da potjera započinje samo ako je agent u vidnom polju agenta.

Kao okolinu koristimo teren (3D Object->Plane) te 4 zida koja okružuju teren sa svake strane (3D object->Cube). Koristiti ćemo prepreke (3D object->Cube) različitih veličina. To će sve izgledati kao na slici:



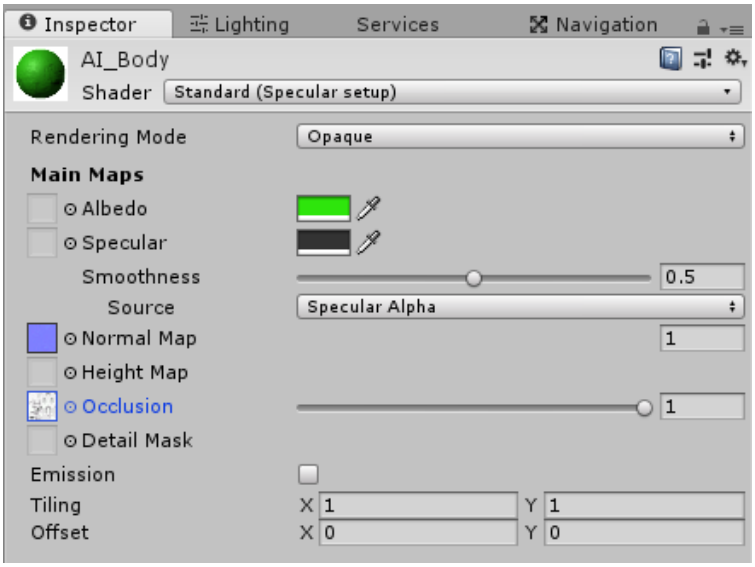
Priprema objekata u prostoru

U ovoj simulaciji ćemo koristiti modele igrača i agenata koji izgledaju malo realističnije, stoga ćemo uvesti Unityjev standardni asset: Assets->Import Package->Characters. Koristiti ćemo Third person likove za igrača i agente tako da u našem projektu idemo na Standard Assets->Characters->Third Person->Prefabs te povučemo ThirdPersonController prefab u scenu kako bismo dobili našeg igrača. Nakon toga možemo povući AIThirdPersonController prefab (kasnije možemo duplicirati agente koliko god želimo sad možemo raditi na jednom agentu). Na slici vidimo prefab objekt koji ćemo koristiti kao našeg igrača.



Koristimo ThirdPersonController prefab u ovoj simulaciji

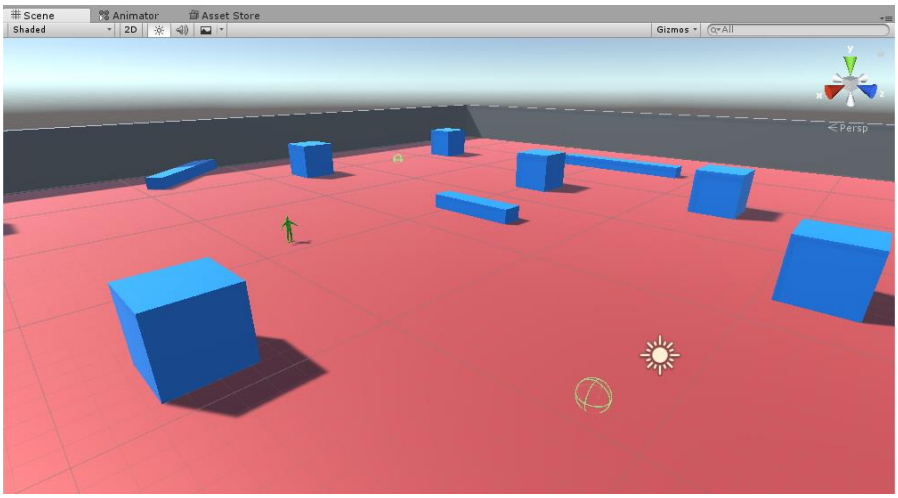
Igrač i agent sada imaju svoje animacije za trčanje i različite materijale za npr. naočale i tijelo. Želimo razlikovati agenta od igrača tako da ćemo promijeniti materijal naočala (Ethan Glasses komponenta) i tijelo (Ethan Body komponenta). Kreiramo novi materijal u našem projektu: Create->Material, postavljamo shader na „Standard (specular setup)“. Promijeniti ćemo boju (Albedo) na npr. zelenu, Normal map na „EthanNormals“ i Occlusion na „EthanOcclusion“. Slika prikazuje postavke novokreiranog materijala:



Postavke kreiranog materijala za tijelo agenta

Također možemo promijeniti materijal i za naočale u neku drugu boju koristeći Standard shader te Normal Map i Occlusion isto kao na prethodnoj slici.

Prije izrade koda nam još samo trebaju različite točke na terenu (eng. Waypoint, u nastavku waypoint) koji predstavljaju pozicije do kojih agenti patroliraju.U ovom primjeru je korišteno 6 različitih točaka predstavljenih kao sfere na terenu. Na slici možemo vidjeti izraz različitih sfera.



Prikaz scene gdje možemo vidjeti zeleni izraz sfera postavljenih u prostoru(waypointi)

Mi ne želimo da su sfere prikazane jer igrač onda zna po kojim putanjama agent prolazi tako da ih možemo isključiti ako kliknemo na sferu i u inspektoru isključimo Mesh Renderer. Također moramo svakoj sferi dodati tag „Waypoint“ kako bi kasnije koristili u skripti.

Navmesh nećemo pretjerano koristiti u ovoj simulaciji tako da možemo jednostavno podesiti okolinu da je Navigation Static te u prozoru Navigation->Bake možemo izraditi Navmesh za našu okolinu.

Kako u početku nećemo koristiti vidno polje agenta nego ćemo samo ispitati ako je došao blizu igrača, moramo dodati komponentu: Sphere Collider te postaviti radijus koji će odrediti koliko blizu igrač može biti do agenta prije nego se sudari sa colliderom.Također za našeg agenta možemo dodati komponentu NavMesh Agent.

Zadnje što je preostalo je kreirati skriptu koja će kontrolirati sve procese agenata. Možemo ju nazvati „Ai“.

Funkcija *Update()* nam neće trebati te ju možemo izbrisati. Koristimo sljedeće biblioteke kao na slici:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;
using UnityStandardAssets.Characters.ThirdPerson;
```

Biblioteke korištene u skripti

Varijable koje ćemo koristiti su: referenca na agenta i igrača, varijabla State tipa enumerate(poprima vrijednosti Chase i Patrol), varijabla tipa State koja prati u kojem je stanju agent i boolean varijabla „alive“. Također koristimo varijable specifične samo za patrolu ili za potjeru. Koristimo polje GameObject-a u kojoj spremamo naše točke (sfere) koje smo postavili. Varijabla waypointIndex nam govori o tome koja je točka izabrana sljedeća kada agent dođe do prethodne točke (nasumično će odabrati destinaciju). Koristimo [patrol/chase]Speed koja označava kojom brzinom se agent kreće u određenom stanju te target varijabla govori o tome gdje se agent kreće kada naiđe na igrača.

```
public NavMeshAgent agent;
public ThirdPersonCharacter character;

// automat
public enum State
{
    PATROL, CHASE
}

public State state;
private bool alive;

// varijable za patrolu
public GameObject[] waypoints;
private int waypointIndex;
public float patrolSpeed = 0.5f;

// varijable za potragu
public float chaseSpeed = 1f;
public GameObject target;
```

Priprema i inicijalizacija varijabli

U *Start()* funkciji, odnosno prilikom pokretanja programa inicijaliziramo agenta i igrača. Isključujemo ažuriranje rotacije prilikom traženja puta zato jer će nam skripta Third Person Character to raditi za nas automatski. Pomoću funkcije *FindGameObjectsWithTag()* popunjujemo polje sa GameObject-ima koje smo postavili (sfere). Za određivanje sljedećeg waypointa agenta koristimo *Random()* funkciju koja će izabrati nasumični indeks od polja „waypoints“. Inicijalizirati ćemo početno stanje na „patrol“ te kažemo da je agent „živ“. Nakon toga pozivamo korutinu „FSM“ koju naknadno implentiramo. Slika prikazuje dio koda:

```
// Use this for initialization
void Start()
{
    agent = GetComponent<NavMeshAgent>();
    character = GetComponent<ThirdPersonCharacter>();

    agent.updatePosition = true;
    // rotacijama se bavimo preko animacija
    agent.updateRotation = false;

    waypoints = GameObject.FindGameObjectsWithTag("Waypoint");
    waypointIndex = Random.Range(0, waypoints.Length - 1);

    state = ai.State.PATROL;
    alive = true;

    //FSM --> Finite State Machine

    StartCoroutine("FSM");

}
```

Start() funkcija – što se događa prilikom pokretanja programa

Automat implementiramo tako da provjeravamo je li agent živ pa onda u kojem je trenutno stanju i ovisno o stanju pozivamo funkciju koja odgovara tom stanju. Slika prikazuje dio koda:

```
IEnumerator FSM()
{
    while (alive)
    {
        switch (state)
        {
            case State.PATROL:
                Patrol();
                break;
            case State.CHASE:
                Chase();
                break;
        }
        yield return null;
    }
}
```

Implementacija automata u obliku koda

U funkciji *Patrol()* postavljamo brzinu agenta na postavljenu brzinu prilikom inicijalizacije. Tada se pitamo koliko je agent udaljen od trenutnog waypointa, ako je udaljen >=2 (u ovom

slučaju se ta brojka čini optimalna) tada će se agent kretati prema tom waypointu, inače ako je udaljen za manje od 2 onda će odabrati sljedeći waypoint kao destinaciju. Slika prikazuje dio koda:

```
void Patrol()
{
    agent.speed = patrolSpeed;
    // razlika izmedju lokacije AI-a i waypointa
    if(Vector3.Distance(this.transform.position, waypoints[waypointIndex].transform.position) >= 2)
    {
        agent.SetDestination(waypoints[waypointIndex].transform.position);
        character.Move(agent.desiredVelocity, false, false);
    }
    else if(Vector3.Distance(this.transform.position, waypoints[waypointIndex].transform.position) < 2)
    {
        waypointIndex = Random.Range(0, waypoints.Length - 1);
    }
    else
    {
        character.Move(Vector3.zero, false, false);
    }
}
```

Implementacija patroliranja agenata

Metodu *Chase()* ćemo implementirati na jednostavan način: postavljamo brzinu agenta u stanju potjere na brzinu potjere prilikom inicijalizacije. Tada pomoću funkcija *SetDestination()* te *Move()* kažemo agentu da se kreće prema igraču. Slika prikazuje dio koda potreban:

```
void Chase()
{
    agent.speed = chaseSpeed;
    agent.SetDestination(target.transform.position);
    character.Move(agent.desiredVelocity, false, false);
}
```

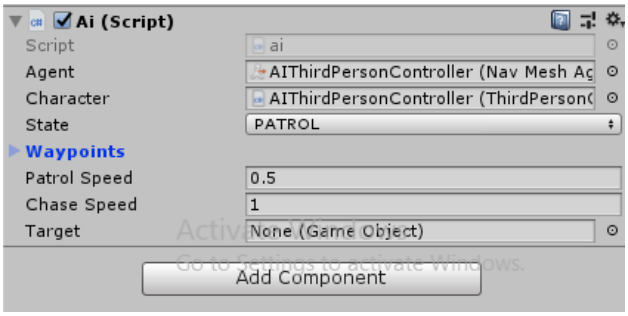
Implementacija potjere

Zadnje što je preostalo implementirati su funkcije koja se brine o sudaru sa objektima odnosno *OnTriggerEnter()*. Objekt sa kojim se sudario agent mora imati tag „Player“, tada postavljamo stanje agenta na „chase“, a cilj („target“) će biti igrač. Slika prikazuje dio koda potreban:

```
private void OnTriggerEnter(Collider other)
{
    if(other.tag == "Player")
    {
        state = ai.State.CHASE;
        target = other.gameObject;
    }
}
```

Funkcija OnTriggerEnter() koja je korisna kada imamo sudar objekata

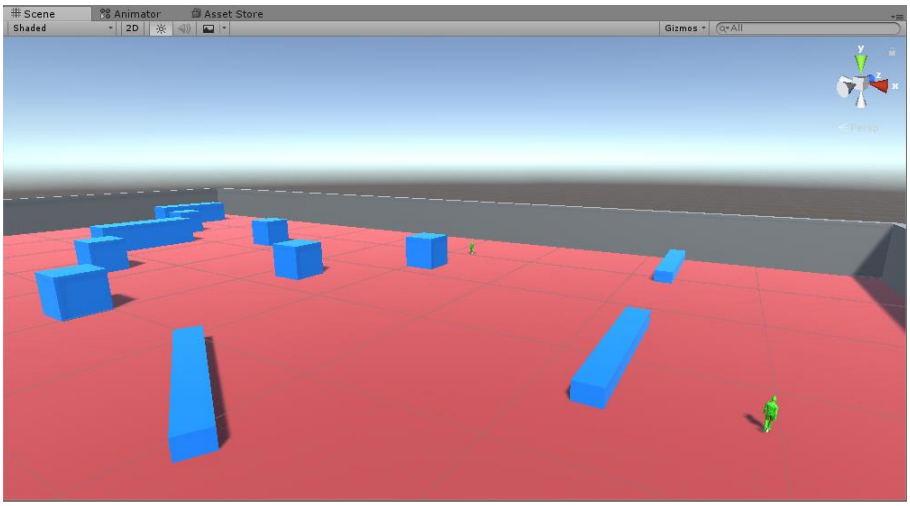
Sada kada smo gotovi sa kodiranjem moramo referencirati igrača i agenta u skripti, to činimo kao na slici:



Refenciramo Navmesh Agent i Character komponentu

Sljedeće što možemo napraviti je dodati još nekoliko agenata da bude zanimljivije te postaviti kameru da prati igrača (objekt „Main Camera“ postavimo da bude dijete odnosno „child“ objekta „ThirdPersonController“). Ako smo sve dobro riješili simulacija će izgledati kao na slikama:

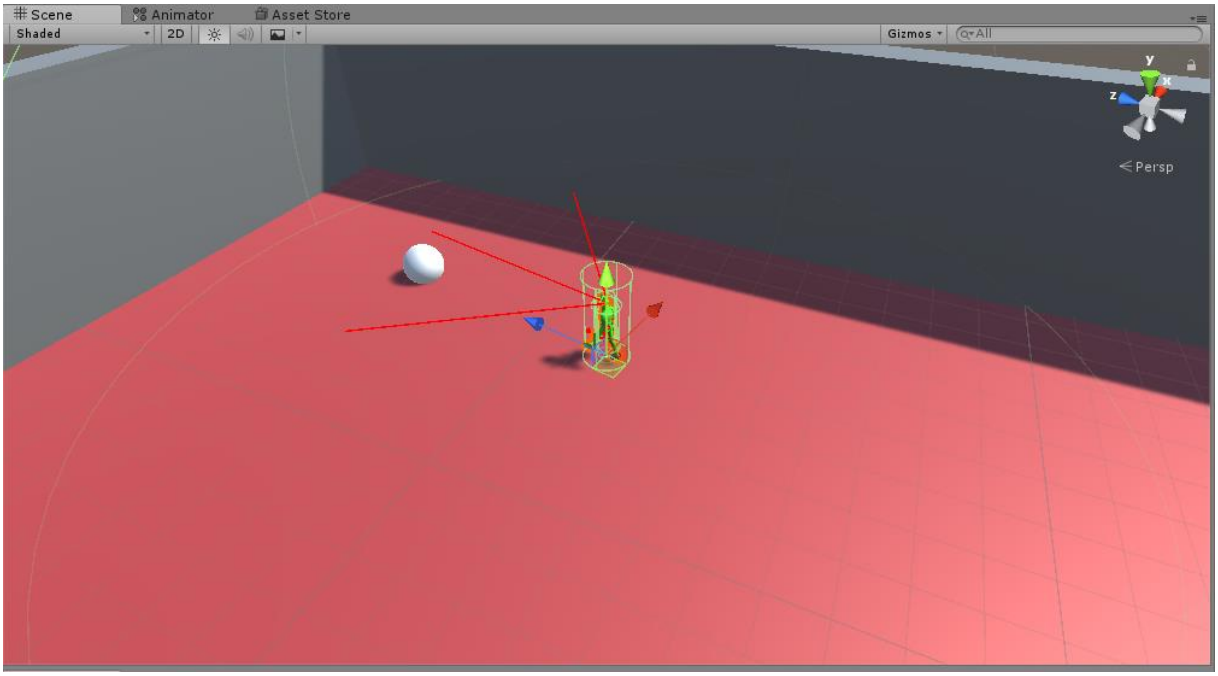
Čim se simulacija pokrene agent se nalazi u stanju patrole i to se cijelo vrijeme događa jer se stalno poziva funkcija *Patrol()* u automatu dok nešto ne izazove promjenu stanja agenta.



Agenti obavljaju funkciju patroliranja prostora

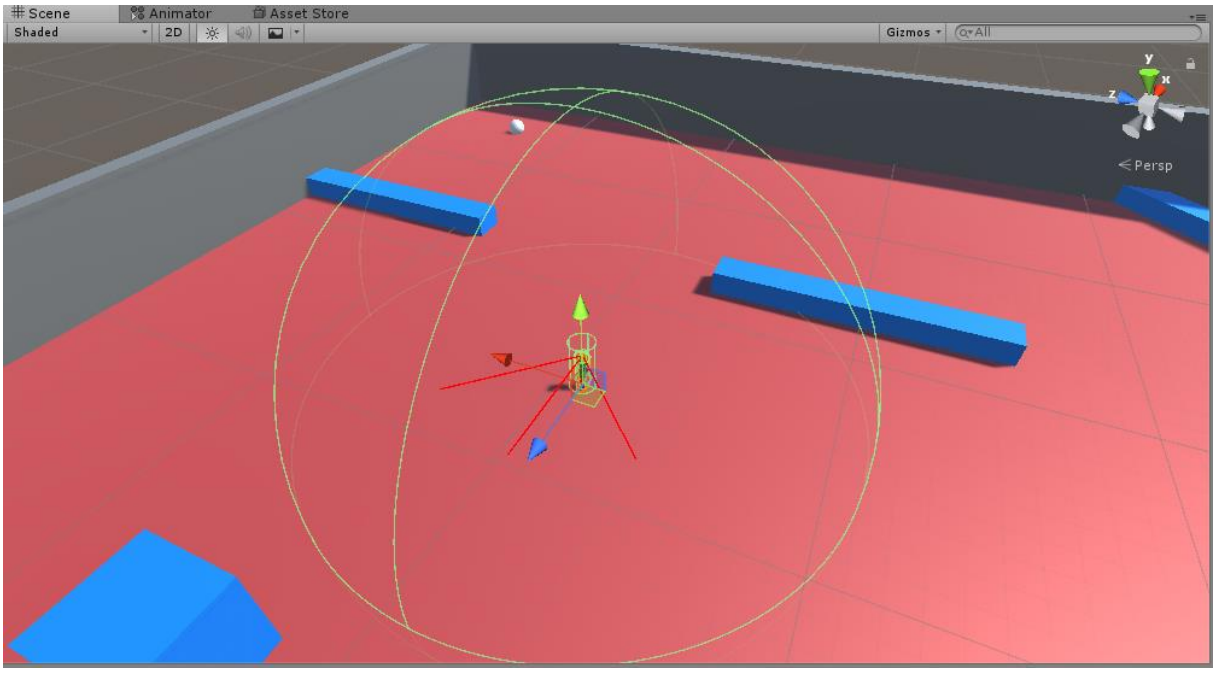
Cilj agenta je nasumično odabrani waypoint koji se prvo inicijalizira pri pokretanju programa te se onda nakon što se nalazi blizu tog waypointa bira novi sljedeći i tako redom. U programu

se postavili da ne prikazuje model waypointa u prostoru jer nije realistično no ako označimo da pokazuje možemo vidjeti kako agent se kreće prema toj sferi.

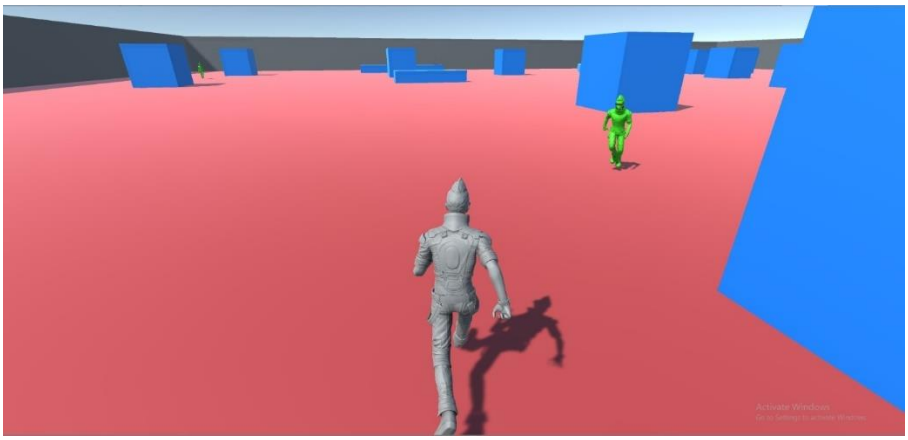


Agent se kreće prema waypointu koji smo označili da se prikazuje njegov model

Funkcija potjere se inicira ako je igraš u radijusu sfere koja opisuje agenta koja se ponaša kao trigger collider. Možemo ju vidjeti na sljedećoj slici:



Sfera koja opisuje agenta koja se ponaša kao trigger collider. Kada igrač uđe u sferu započinje funkcija Chase()



Agent obavlja funkciju potjere nad igračem

Sljedeći zadatak je malo poboljšati način na koji agent može opaziti igrača. Uvesti ćemo funkciju istrage odnosno *Investigate()*. Plan je da bacamo 3 zrake iz područja očiju agenta: jedna lijevo pod 45°, jedna u sredini te jedna desno pod 45°. Kada se igrač između tih zraka započinje Chase() funkciju koju smo već implementirali, no sada ako se igrač nalazi u trigger collideru neće više započeti *Chase()* funkciju nego *Investigate()* funkcija gdje će agent stati neko vrijeme i promatrati igrača i samo ako se igrač dovoljno približi te na kraju upadne između zraka agenata tada započinje *Chase()* funkcija.

Ponovno otvaramo skriptu „ai.cs“ te dodajemo novo stanje Investigate u automatu. Nakon toga kreiramo nove varijable za istragu i za vid agenta. Kao varijable za istragu ćemo imati: private Vector3 investigateSpot koje će biti mjesto na kojem se nalazi igrač, private float timer koji odbrojava vrijeme istrage i public float investigateWait koji sadrži koliko je maksimalno vrijeme istrage. Varijable za vid agenta će nam biti public float heightMultiplier tako da se prilikom izračuna pozicija pozicioniramo na oči agenta i public float sightDist je udaljenost zraka iz očiju agenata.

```
// automat
public enum State
{
    PATROL,
    CHASE,
    INVESTIGATE
}

public State state;
private bool alive;

// varijable za patrolu
public GameObject[] waypoints;
private int waypointIndex;
public float patrolSpeed = 0.5f;

// varijable za potragu
public float chaseSpeed = 1f;
public GameObject target;

// varijable za istragu
private Vector3 investigateSpot;
private float timer = 0;
public float investigateWait = 10;

// varijable za vid AI-a
public float heightMultiplier;
public float sightDist = 5;
```

Inicijalizacija varijabli

Prije nego što započnemo implementaciju *Investigate()* funkcije moramo inicijalizirati public float heightMultiplier=1.36f u *Start()* funkciji.

Sljedeće što moramo je implementirati *FixedUpdate()* funkciju koja se pokreće u svakoj sličici(eng. frame) te ćemo pomoću *Debug.DrawRay()* funkcija nacrtati zrake za lakše debugiranje u pogledu scene. Nakon toga se pitamo ako je jedna od zraka pogodila igrača i ako je promijeni stanje u Chase.

```
private void FixedUpdate()
{
    RaycastHit hit;
    Debug.DrawRay(transform.position + Vector3.up * heightMultiplier, transform.forward * sightDist, Color.red);
    Debug.DrawRay(transform.position + Vector3.up * heightMultiplier, (transform.forward + transform.right).normalized * sightDist, Color.red);
    Debug.DrawRay(transform.position + Vector3.up * heightMultiplier, (transform.forward - transform.right).normalized * sightDist, Color.red);
    if (Physics.Raycast(transform.position + Vector3.up * heightMultiplier, transform.forward, out hit, sightDist))
    {
        if (hit.collider.gameObject.tag == "Player")
        {
            state = ai.State.CHASE;
            Debug.Log("TARGET HIT");
            target = hit.collider.gameObject;
        }
    }
    if (Physics.Raycast(transform.position + Vector3.up * heightMultiplier, (transform.forward + transform.right).normalized, out hit, sightDist))
    {
        if (hit.collider.gameObject.tag == "Player")
        {
            state = ai.State.CHASE;
            Debug.Log("TARGET HIT");
            target = hit.collider.gameObject;
        }
    }
    if (Physics.Raycast(transform.position + Vector3.up * heightMultiplier, (transform.forward - transform.right).normalized, out hit, sightDist))
    {
        if (hit.collider.gameObject.tag == "Player")
        {
            state = ai.State.CHASE;
            Debug.Log("TARGET HIT");
            target = hit.collider.gameObject;
        }
    }
}
```

Implementacija funkcije FixedUpdate() te bacanje zraka iz očiju agenata provjeravajući koji je objekt udaren

Trebamo modificirati funkciju *OnTriggerEnter()* tako da se više ne poziva funkcija *Chase()* kao što smo prije tako postavili nego će se pozivati funkcija *Investigate().*

```
private void OnTriggerEnter(Collider other)
{
    if(other.tag == "Player")
    {
        state = ai.State.INVESTIGATE;
        Debug.Log("INVESTIGATE");
        investigateSpot = other.gameObject.transform.position;
    }
}
```

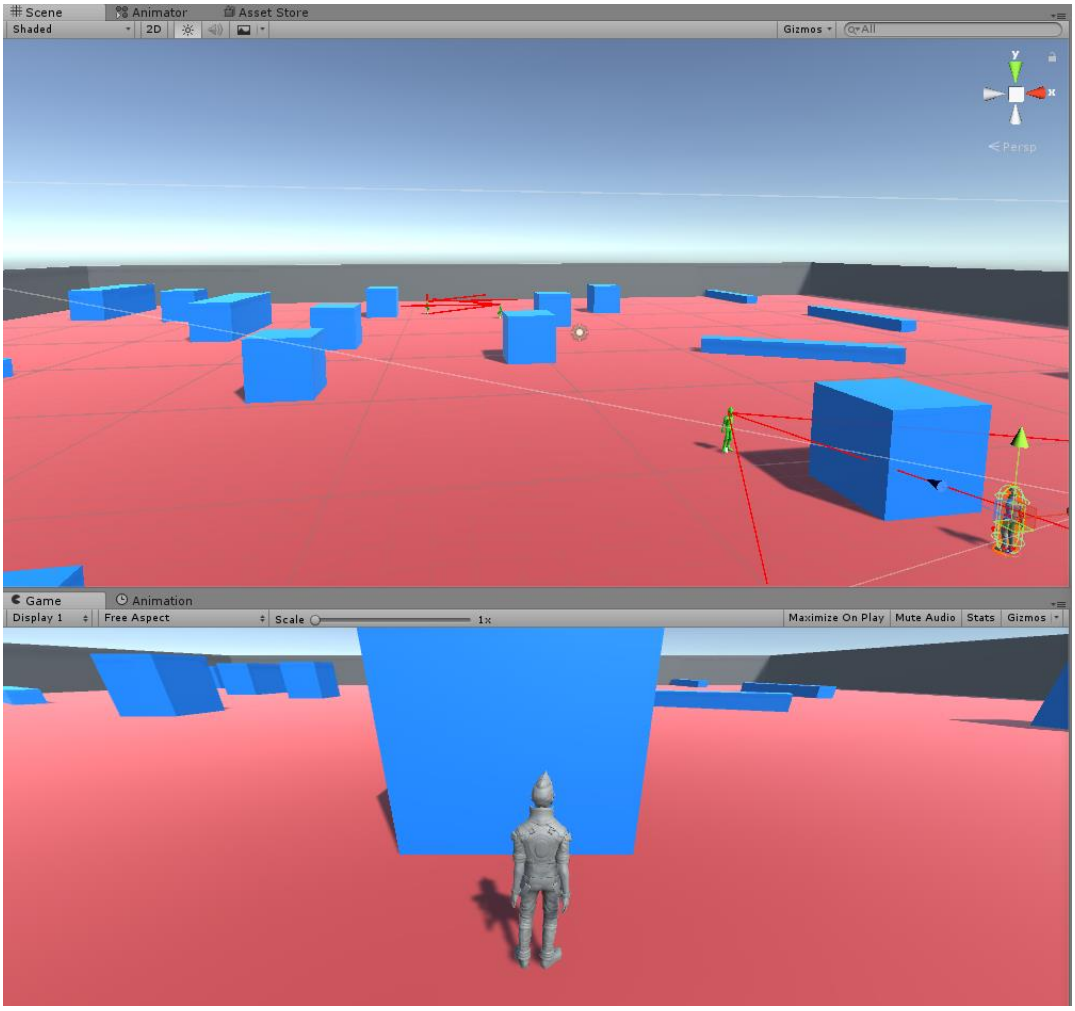
Modifikacija funkcije OnTriggerEnter() gdje se pitamo ako je udaren objekt igrač tada promijeni stanje u stanje istrage

Zadnje je preostalo implementirati funkciju *Investigate()*. Započnemo timer koji će se povećavati, agenta zaustavimo na način da postavimo njegovu destinaciju gdje se trenutno nalazi te njegovu brzinu na 0. Nakon toga pomoću funkcije *LookAt()* agent se rotira i gleda u igrača, ako unutar vremena na koji je postavljen investigateWait() igrač se previše približi agentu započinje *Chase()* funkcija jer se tada nalazi između zraka no ako se igrač ne pomakne ili ako se udalji od agenta tada će to agent zanemariti i nastaviti obavljati svoju funkciju patroliranja.

```
void Investigate()
{
    Debug.Log("Investigate");
    timer += Time.deltaTime;
    agent.SetDestination(this.transform.position);
    character.Move(Vector3.zero, false, false);
    transform.LookAt(investigateSpot);
    if (timer >= investigateWait)
    {
        Debug.Log(investigateWait);
        state = ai.State.PATROL;
        timer = 0;
    }
}
```

Implementacija funkcije Investigate()

Na sljedećoj slici možemo vidjeti što se događa kada agent opazi igrača odnosno kada igrač uđe u trigger collider no nalazi se iza drugog objekta te ga zbog toga agent ne može vidjeti. Agent će pričekati sve dok timer nije veći od zadanog vremena čekanja investigateWait te će tek tada nastaviti patrolirati i zanemariti igrača.



Igrač se nalazi iza kocke te ga zbog toga agent ne može vidjeti

11 ZAKLJUČAK

Kroz nekoliko primjera smo pokazali kako se može primijeniti umjetna inteligencija u Unityju koristeći podatkovnu strukturu: Navmesh.

U početku su objašnjeni najosnovniji pojmovi vezani za temu te na koji način algoritam radi. Nakon toga u prvom primjeru je prikazano kako možemo na najjednostavniji način implementirati traženje puta putem Unityjevog Navmesha te pomoću jednostavne skripte napraviti da kada igrač klikne na neku poziciju da se agent pomakne tamo. U sljedećim primjerima smo vidjeli kako možemo generirati Navmesh tijekom pokretanja programa te kako traženje puta funkcionira kada imamo na tisuće agenata koji pokušavaju doći do destinacije.

Nadalje je objašnjeno kako se može primijeniti i u 2D svijetu, npr. bilo kakav neprijatelj u raznim platformer igricama te u konačnici smo sa malo većim projektom gdje smo testirali funckije patrole, potjere i istrage.

12 POPIS LITERATURE

1. Wikipedia (posljednja izmjena: 11.8.2018.), dio teksta preuzet sa: https://en.wikipedia.org/wiki/Intelligent_agent, pristupano 4.9.2018.
2. Wikipedia(posljedna izmjena: 10.7.2018.), dio teksta preuzet sa: https://en.wikipedia.org/wiki/A*_search_algorithm, pristupano 17.8.2018.
3. GeeksforGeeks, slika preuzeta sa: https://www.geeksforgeeks.org/a-search-algorithm/, pristupano 2.9.2018.
4. GeeksforGeeks(objavljeno 2016.g.), dio teksta preuzet sa: https://www.geeksforgeeks.org/a-search-algorithm/, pristupano 17.8.2018.
5. Sebastian Lague, slika preuzeta sa isječka Youtube videa(2:00 min): https://www.youtube.com/watch?v=-L-WgKMFuhE, pristupano 2.9.2018.
6. Sebastian Lague, slika preuzeta sa isječka Youtube videa(8:30 min): https://www.youtube.com/watch?v=-L-WgKMFuhE, pristuapno 2.9.2018.
7. Aron Granberg, dio teksta preuzet sa: https://arongranberg.com/astar/docs/graphtypes.html, pristupano 19.8.2018.
8. Unity Technologies(objavljeno 2017.g.), preuzeto sa: https://docs.unity3d.com/Manual/nav-NavigationSystem.html, pristupano 19.8.2018.
9. Unity Technologies(objavljeno 2017.g.), slika preuzeta sa: https://docs.unity3d.com/Manual/nav-NavigationSystem.html, pristupano 19.8.2018.