

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 774

**PRIMJENA ALGORITAMA
TRAŽENJA PUTA U PONAŠANJU
RAČUNALNO-GENERIRANIH
LIKOVА U VIDEOIGRI**

Dominik Arih

Zagreb, lipanj 2022.

SADRŽAJ

1. Uvod	1
2. Navigacijski sustav radnog okruženja Unity	2
2.1. Komponente navigacijskog sustava	3
2.2. Unutarnji mehanizmi navigacijskog sustava	4
2.2.1. Prohodna područja i pronalaženje puta	4
2.2.2. Praćenje puta i izbjegavanje prepreka	6
2.2.3. Globalna i lokalna navigacija	8
3. Algoritmi traženja puteva	9
3.1. Uvod u algoritme	9
3.2. Slijepo pretraživanje	11
3.2.1. Pretraživanje u širinu (BFS)	11
3.2.2. Pretraživanje s jednolikom cijenom (UCS)	12
3.2.3. Pretraživanje u dubinu (DFS)	13
3.2.4. Iterativno pretraživanje u dubinu (IDFS)	14
3.3. Usmjereno pretraživanje	15
3.3.1. Heuristika	15
3.3.2. Algoritam A*	15
3.3.3. Modifikacija algoritma A*	17
3.3.4. Svojstva heuristike	18
4. Opis pokazne videoigre	20
4.1. Dizajn okoline	20
4.1.1. Labirint	20
4.1.2. Ambijent	21
4.2. Mehanika igre	22
4.2.1. FPS Mehanika	22

4.2.2. Interaktivni elementi	23
4.3. Izazovi	23
4.3.1. Cilj igre	24
4.3.2. Vrste neprijatelja	24
4.4. Tijek igre	25
5. Razvoj pokazne videoigre	26
5.1. Razvojna okolina Unity	26
5.2. Postavljanje scene	26
5.2.1. Izrada terena	27
5.2.2. Putne točke	30
5.3. Izrada navigacijske mreže	31
5.4. Igrač	33
5.4.1. Mehanika kretanja	33
5.4.2. Mehanika pucanja	34
5.4.3. Detekcija kolizije	40
5.5. Oblikovanje neprijatelja	43
5.6. Elementi za sakupljanje	51
5.7. Heads-up zaslon	53
6. Rezultati	55
7. Zaključak	58
Literatura	59

1. Uvod

Pronalaženje puta jedan je od osnovnih problema umjetne inteligencije, a samim time i neizostavni predmet proučavanja industrije videoigara. U videoograma, ono što najčešće želimo ostvariti je naizgled inteligentno ponašanje jednog ili više agenata, koji na temelju implementiranih algoritama pokušavaju pronaći najkraći put između dvije točke terena. Sustav odabire početnu i odredišnu točku te sastavlja putanju od niza susjednih točaka. Ono što predstavlja velik problem prilikom korištenja takve strategije jest iscrpno trošenje resursa računalnog procesora koje raste proporcionalno s vremenom potrebnim za pronalaženje puta [1].

S obzirom na spomenuti problem, želimo da takav algoritam bude optimalan. Kako bi se ostvarila optimalnost i potpunost, s godinama se razvijaju novi, ali i varijacije starijih algoritama slijepog i usmjerenog pretraživanja. Algoritam A*, pokazao je velik potencijal u industriji te postavio temelje za primjenu i daljnje proučavanje u kontekstu razvoja digitalnih igara [2].

U ovom će radu biti opisan jedan od najčešće korištenih načina uporabe algoritma za traženje puta u videoograma, koristeći razvojno okruženje Unity. U tu svrhu, razvijena je pokazna igra, koja osim spomenutih algoritamskih rješenja, sadržava i elemente ambijentalnog FPS-a (engl. *First-person shooter*) kako bi se dobio cjeloviti i zaokruženi proizvod.

2. Navigacijski sustav radnog okruženja Unity

Navigacijski sustav unutar radnog okruženja Unity, implementiran sa svim svojim komponentama, omogućava inteligentno kretanje likova na zadatom terenu ka zadanim ciljevima, izbjegavajući postavljene prepreke. Vizualizacija takvog sustava, unutar sučelja radnog okruženja, postignuta je trodimenzionalno s jasno označenim komponentama, kako bi se za svaku komponentu mogle odrediti zasebne postavke.

Lik upravljan algoritmom umjetne inteligencije naziva se **inteligentni agent**. Ono što je osnova intelligentnog agenta u svim domenama, pa tako i u području digitalnih igara, jest: smještenost unutar nekog okruženja uz obavljanje neke autonomne akcije kako bi postigao prepostavljene ciljeve. Naglasak se stavlja na **autonomnost**, što znači da su takvi agenti potpuno neovisni i da donose vlastite odluke [3].

Kako bismo ostvarili funkcionalnost intelligentnog agenta u Unityju, potrebno je riješiti dva problema: **statički/globalni** i **dinamički/lokralni**. Statički problem uzima u obzir čitavu scenu i vezan je isključivo uz pronalaženje najisplativije putanje. Dinamički problem odnosi se na kretanje definiranom putanjom te izbjegavanje bilo kakvih prepreka ili drugih intelligentnih agenata na tom putu [4].

2.1. Komponente navigacijskog sustava

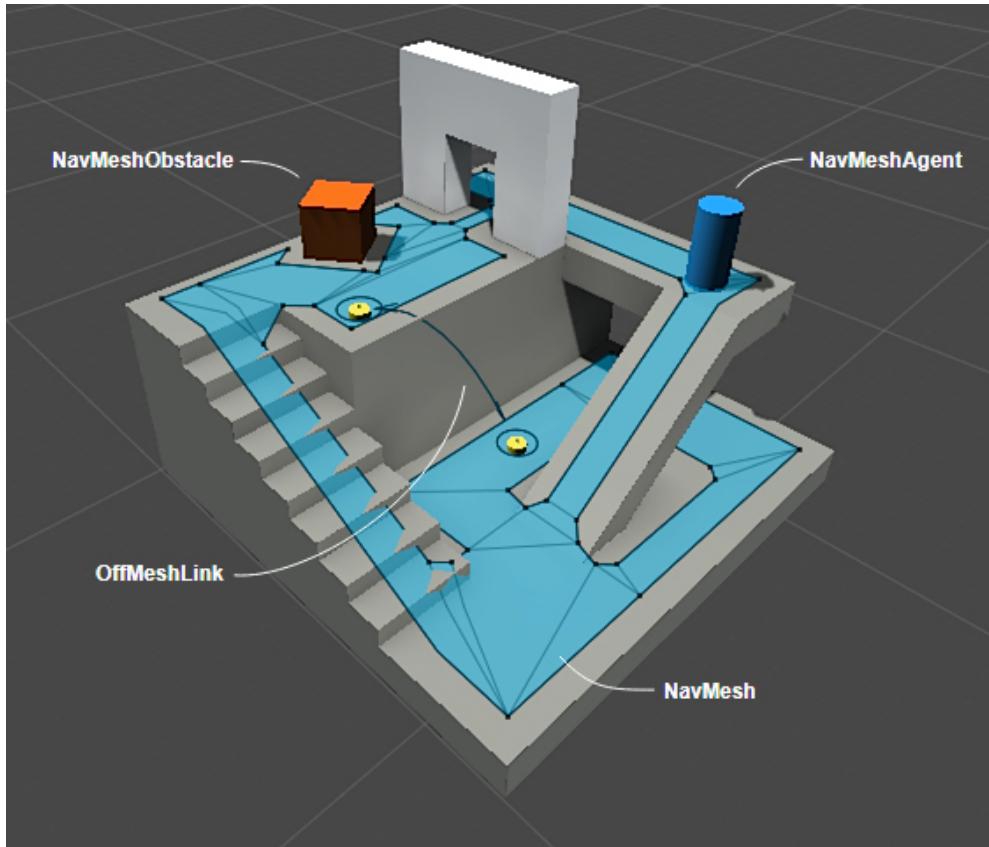
Navigacijski sustav (Slika 2.1) sastoji se od 4 komponente: navigacijska mreža, agent, izvan-mrežna poveznica i prepreka.

Navigacijska mreža (engl. *NavMesh*) je strukturu podataka koja predstavlja dijelove terena po kojima je moguće hodati te se kao takvi uzimaju u obzir prilikom kalkulacije putanje do željenog cilja (Slika 2.1). Postoji automatizirani sustav izgradnje takve mreže prema karakteristikama terena.

Agent (engl. *NavMesh Agent*) koristi navigacijsku mrežu i njene prohodne površine kako bi stigao do cilja što povoljnijom putanjom. Njegov je zadatak izbjegavati sve vrste prepreka kao i druge agente. Na slici 2.1 je agent prikazan plavim valjkom.

Izvan-mrežne poveznice (engl. *Off-Mesh Link*) agentu nude dodatne informacije o prohodnosti terena. S obzirom na to da su navigacijskom mrežom definirane samo neprekinute prohodne površine, ovakvim poveznicama moguće je definirati i zaobilazne puteve koji bi mogli agentu omogućiti izgradnju efikasnije putanje do cilja. Definiranjem poveznica izvan navigacijske mreže moguće je označiti ograde i ponore koje je moguće preskočiti ili vrata koja je potrebno prvo otvoriti kako bi se kroz njih prošlo [4]. Na slici 2.1 je izvan-mrežna poveznica prikazana zakriviljenom crtom između dva žuta oblika pri vrhu i dnu platforme.

Prepreke (engl. *NavMesh Obstacle*) obilježavaju dijelove scene koje je potrebno zaobilaziti i izbjegavati. Takve prepreke mogu biti u statičkom ili dinamičkom obliku, ali će neovisno o tome utjecati na odluke agenta pri rješavanju statičkog problema izgradnje svoje putanje. Na slici 2.1 je prepreka prikazana narančastom kockom na vrhu platforme.



Slika 2.1: Navigacijska mreža sa svojim komponentama. Slika preuzeta iz [4]

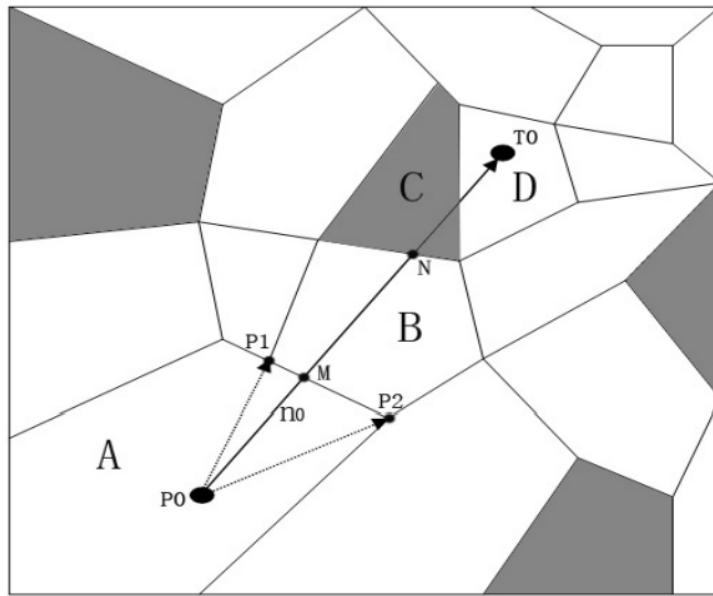
2.2. Unutarnji mehanizmi navigacijskog sustava

Već je ranije spomenuto kako agent unutar navigacijskog sustava rješava i statički i dinamički problem prilikom traženja puteva. Sada kada znamo koje komponente sačinjavaju takav sustav, oba problema bismo mogli razložiti još i na manje pod-probleme koji zajedno čine unutarnje mehanizme navigacijskog sustava. Potrebno je sagledati dvije etape kroz koje inteligentni agent prolazi u svojem razmišljanju kako bi došao da optimalnog i potpunog rješenja.

2.2.1. Prohodna područja i pronalaženje puta

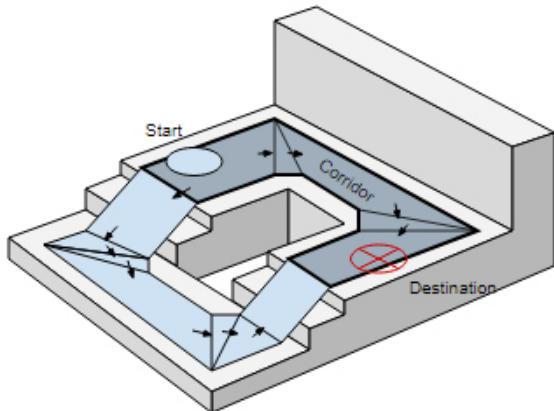
Prohodna područja koja nastaju izgradnjom navigacijske mreže u osnovi su predstavljena mrežom geometrijskih oblika. Sustav na inteligentnog agenta gleda kao na valjak, pa se prohodna područja geometrijski računaju kao područja na kojima valjak može stajati. Kada su sva takva područja zabilježena, bivaju međusobno povezana u mrežu konveksnih poligona. Na slici 2.2 su prikazana tri prohodna područja (A, B, D) i jedno neprohodno područje (C), a zadatku algoritma je pronaći odgovarajući put

od točke P_0 do točke T_0 . Točke P_1, P_2, M i N povezane sa početnom točkom predstavljaju niz mogućih puteva do odredišne točke. [5].



Slika 2.2: Prikaz terena kao mreže poligona. Slika preuzeta iz [5]

Pronalaženje puta započinje zadavanjem početne i ciljne točke. Na slici 2.3, početna točka označena je kao "Start", ciljna kao "Destination", a "Corridor" označava područje između tih točaka. S obzirom na to da je izgradnjom navigacijske mreže teren pretvoren u mrežu konveksnih poligona, početni i ciljni čvor se dodjeljuju poligonu kojem su najbliži. Pronalaženje puta je u osnovi prolazak po čvorovima i vezama usmjerenog grafa pa se tako u ovom sustavu pretražuju poligoni i njihove međusobne veze. Proširivanjem pojedinih poligona pa njihove djece, gradi se stablo pretraživanja koje će na kraju rezultirati najpovoljnijom mogućom kombinacijom poligona koje je potrebno posjetiti. Ugrađeni algoritam koji prolazi takvim stablom je **Algoritam A***. Algoritam A* će efikasno i garantirano pronaći putanju između početnog i ciljnog čvora kada god ona postoji [6].



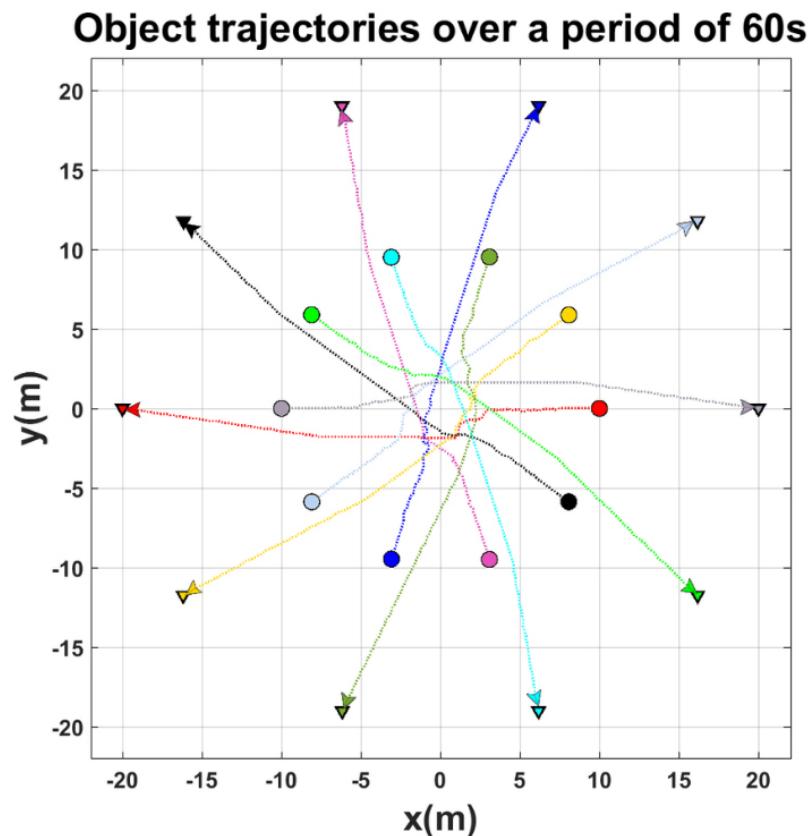
Slika 2.3: Prikaz pronalaženja puta unutar navigacijske mreže. Slika preuzeta iz [4]

2.2.2. Praćenje puta i izbjegavanje prepreka

Praćenje puta pripada dinamičkom dijelu problema koji inteligentni agent rješava. Kako bismo jasnije definirali novonastalu putanju koju agent treba pratiti, definiramo pojam **koridora**. Koridor predstavlja sekvencu poligona od početnog do ciljnog, za koju je agent ustanovio da je najisplativiji put [4]. Način na koji se agent kreće koridorom je pomicanje prema idućem vidljivom kutu koridora. S obzirom na to da je takav oblik navigacije potrebno ostvariti i za više od jednog agenta, ako uzmemu u obzir njihovo međusobno izbjegavanje, prilikom kretanja pojavljuju se sitnija odstupanja u odnosu na originalnu putanju no međusobna povezanost konveksnih poligona omogućava nam brz popravak koridora [4].

Izbjegavanje prepreka jedna je od sastavnica praćenja puta. Agent se prema željenom dijelu koridora pomiče brzinom za koju sam smatra da je idealna čak i ako se na putu nalaze statičke prepreke. Nešto kompleksnija situacija nastaje ako se na putu prema destinaciji pojavi još jedan inteligentni agent kojeg je potrebno izbjegći (dinamička prepreka). Korištena metoda za izbjegavanje dinamičkih prepreka naziva se **Metoda recipročnih brzina**, poznatija kao **RVO** (engl. *Reciprocal Velocity Obstacles*). Svaki agent posjeduje informacije o svim ostalim agentima poput: brzine, trenutne lokacije i oblika. Umjesto da je svakom agentu dodijeljena potpuno nova brzina, oba agenta će moći proizvoljno odabrati svoju novu brzinu (najbližu prethodnoj) te će tako biti jednako zaslužni za izbjegavanje sudara [7].

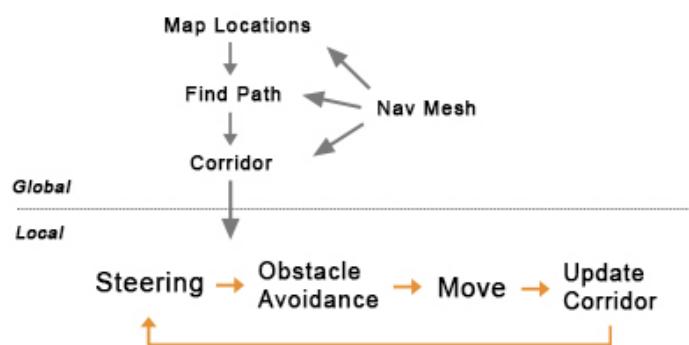
Slika (2.4) prikazuje eksperiment s 10 intelektualnih agenata te promjenu njihovih trajektorija kretanja korištenjem metode RVO za izbjegavanje sudara [7]. Slika 2.4 prikazuje način na koji su trajektorije objekata (prikazane različitim bojama) u središtu grafa zakrivljene uslijed izbjegavanja objekata iz drugih smjerova. Kada ne bi postojala nikakva metoda izbjegavanja, trajektorije bi bile potpuno ravne.



Slika 2.4: Utjecaj metode RVO na trajektorije agenata. Slika preuzeta iz [8]

2.2.3. Globalna i lokalna navigacija

Važno je razlikovati lokalnu od globalne navigacije. Globalna navigacija odnosi se na općeniti problem detekcije prohodnog koridora na temelju opservacije cijelog terena. Nastajanjem takvog koridora, poligoni se spremaju u fleksibilnu strukturu koja se može lokalno mijenjati tijekom kretanja agenta [4]. Zadaća lokalne navigacije svodi se na efikasno kretanje koridorom uz izbjegavanje postojećih prepreka. Globalna navigacija istovjetna je statičkom, a lokalna navigacija dinamičkom problemu koji rješava inteligentni agent. Na slici (2.5) jasno su vidljive odvojene zadaće lokalne i globalne navigacije.



Slika 2.5: Prikaz odnosa komponenata globalne i lokalne navigacije

3. Algoritmi traženja puteva

Pretraživanje prostora stanja može riješiti mnoge analitičke probleme. Ono što je zajedničko svakom takvom problemu su **skup stanja** i **slijedovi akcija**. Točnije, u skupu svih stanja nalazi se početno stanje te jedno ili više ciljnih stanja međusobno povezanih prijelazima. Cjelokupni problem može se shvatiti kao pretraživanje usmjerenog grafa čija struktura zadovoljava prethodno opisane sastavnice odnosno sadrži **čvorove** i **grane**. Pretraživanje mora biti sustavno zato što je u pitanju najčešće velik broj stanja i mogućih izbora.

S obzirom na karakteristike usmjerjenog grafa i zahtjeve problema, za izvođenje pretraživanja moguće je koristiti različite algoritme. Efikasnost, u smislu vremena izvođenja, memorijskog prostora i cijene potrebne za dostizanje ciljnog čvora, predstavlja središnje razlikovno pitanje takvih algoritama [9].

3.1. Uvod u algoritme

Stablo pretraživanja jedan je od središnjih pojmoveva čije je razumijevanje potrebno za ispravno baratanje algoritmima. Ono nastaje postepenim pretraživanjem usmjerjenog grafa tako da se, pomoću funkcije sljedbenika, generiraju (proširuju) svi sljedbenici pojedinog čvora. Prošireni čvorovi nazivaju se još i **zatvoreni**, a ne-prošireni **fronta**. Redoslijed kojim se čvorovi proširuju određuje **strategiju pretraživanja**.

Važno je istaknuti razliku između pojmoveva **čvor** koji se koristi u kontekstu stabala i **stanje** vezanog uz usmjerene grafove. Čvor je u osnovi samo podatkovna struktura u koju se pohranjuju pojedine informacije, a među ostalim i stanje. Postoje i ostale karakteristike koje bismo željeli pohranjivati poput pokazivača na roditeljski čvor ili dubine na kojoj se čvor u stablu nazali no kompleksnost strukture čvora ovisi o odabiru algoritma.

Na primjeru pseudokoda općenitog algoritma za pretraživanje (3.1), pokazane su dosad teorijski objašnjene sastavnice na implementacijskoj razini.

```

1  function search(s0,succ,goal)
2      openList = [initial(s0)]
3      while openList != [ ] do
4          n = removeHead(openList)
5          if goal(state(n)) then
6              return n
7          for m in expand(n,succ) do
8              insert(m,openList)
9      return fail
10

```

Programski isječak 3.1: Pseudokod općenitog algoritma za pretraživanje

Funkcija kao argumente prima početno stanje $s0$, funkciju sljedbenika $succ$, te listu ciljnih stanja $goal$. Na početku, lista otvorenih čvorova inicijalizira se početnim stanjem. Tada započinje petlja koja, dok god lista otvorenih čvorova nije prazna, uzima jedan čvor iz liste i provjerava je li riječ o ciljnem stanju. Ako trenutni čvor nije ciljno stanje, taj čvor se proširuje, a njegovi se sljedbenici stavljuju u listu otvorenih čvorova funkcijom $insert$. Ako provjeravani čvor jest ciljni, funkcija taj čvor vraća.

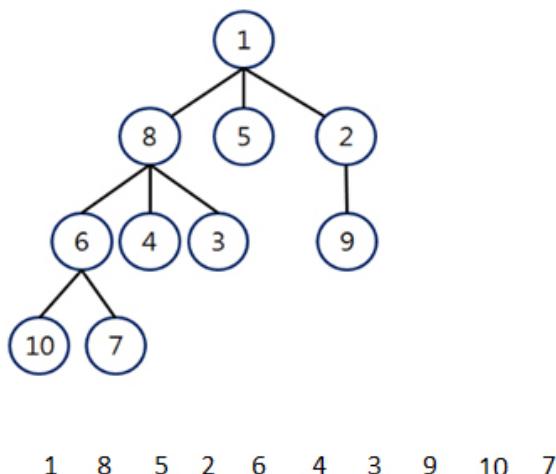
Prije nego što detaljnije razmotrimo implementacije pojedinih algoritama, potrebno je spomenuti važne karakteristike prema kojima ćemo te algoritme razlikovati. Svaki problem pretraživanje prostora sadrži sljedeće karakteristike: ukupan broj stanja, faktor grananja stabla pretraživanja, dubinu optimalnog rješenja te maksimalnu dubinu stabla pretraživanja. Navedene karakteristike koristimo kako bi algoritme mogli razlikovati po potpunosti (algoritam pronalazi rješenje uvijek kada ono postoji), optimalnosti (algoritam pronalazi rješenje s najmanjom cijenom), vremenskoj složenosti (broj generiranih čvorova) te prostornoj složenosti (broj pohranjenih čvorova).

3.2. Slijepo pretraživanje

Ranije smo spomenuli kako redoslijed proširivanja čvorova određuje strategiju pretraživanja. Postoje dvije osnovne takve strategije: slijepo pretraživanje i usmjereno pretraživanje. Ovaj odjeljak daje pregled slijepog pretraživanja koje se ponekad naziva i **uniformno**. Jedino što slijepo pretraživanje može jest razlikovati ne-ciljna stanja od ciljnih te ne postoje nikakve dodatne vanjske pretpostavke i znanja na temelju koji se odabire idući čvor za proširenje [10].

3.2.1. Pretraživanje u širinu (BFS)

Pretraživanje u širinu (engl. *Breadth-first search*) koristi strategiju proširivanja čvorova takvu da se nakon proširenja korijenskog čvora, proširuju njegova djeca, pa djeca djece i tako sve dok nije dosegnut ciljni čvor. Čvorovi na dubini d se proširuju nakon što su se već proširili svi čvorovi na razini $d-1$. Kada dosegnemo zadnju razinu, generiraju se djeca svih čvorova osim ciljnog. Slika (3.2) prikazuje redoslijed kojim se proširuju čvorovi stabla pretraživanja.



Slika 3.2: Prikaz redoslijeda proširivanja čvorova algoritmom BFS. Slika preuzeta iz [11]

Implementacijska izvedba ovog algoritma podrazumijeva način dodavanja generirane djece **na kraj** liste otvorenih čvorova pa takva lista sada funkcioniра као **red čekanja** (engl. *queue*).

Klasifikacija algoritma pretraživanja u širinu, prema svojstvima, jamči da je ovakvo pretraživanje potpuno i optimalno zato što se u svakom koraku proširuje najplići čvor stabla. Negativna strana BFS-a je eksponencijalna vremenska i prostorna složenost, a to je upravo ono što kod ovakvih algoritama želimo izbjegći i postići optimizaciju prema linearnoj ili logaritamskoj složenosti.

3.2.2. Pretraživanje s jednolikom cijenom (UCS)

Pretraživanje s jednolikom cijenom (engl. *Uniform-cost search*) razlikuje se od pretraživanja u širinu po strukturi čvora i po načinu dodavanja čvorova u listu. Osnovna pretpostavka je da sada pretražujemo usmjereni **težinski** graf te da su prijelazima između čvorova dodijeljene numeričke vrijednosti (cijene prijelaza). U strukturu čvora više ne bilježimo dubinu čvora na kojoj se on nalazi unutar stabla već ukupnu cijenu puta do tog čvora koju će u ovom slučaju ažurirati funkcija prijelaza. U listu otvorenih čvorova, elementi se dodaju sortirano uzlazno prema vrijednosti funkcije za izračun cijene.

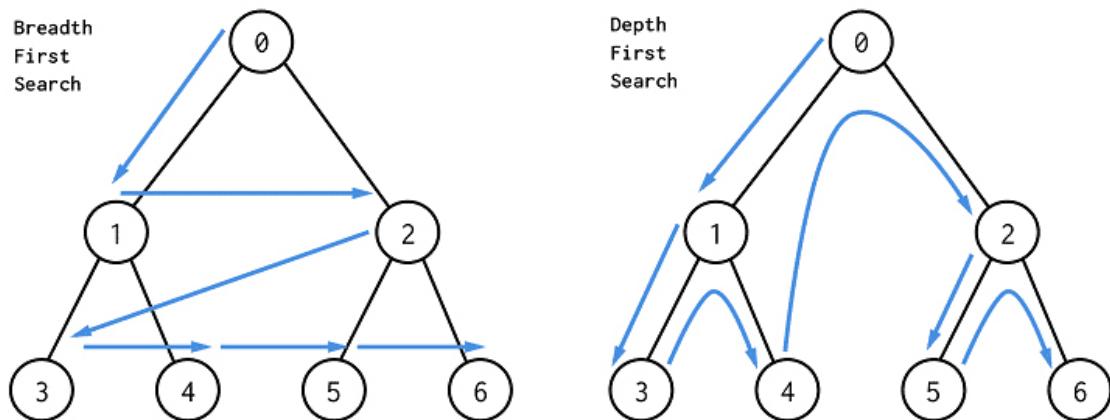
Isto kao i BFS i ovaj algoritam je potpun i optimalan. Vremenska i prostorna složenost su i dalje eksponencijalne, ali sada su za njihovo izračunavanje potrebni još i optimalna cijena do cilja te minimalna cijena prijelaza u stablu.

```
1  function uniformCostSearch(s0, succ, goal)
2      openList = [initial(s0)]
3      while openList != [] do
4          n = removeHead(openList)
5          if goal(state(n)) then
6              return n
7          for m in expand(n, succ) do
8              insertSortedBy(price(m), m, openList)
9      return fail
10 
```

Programski isječak 3.3: Pseudokod algoritma UCS uz naglašene razlike u odnosu na opći algoritam

3.2.3. Pretraživanje u dubinu (DFS)

Pretraživanje u dubinu (engl. *Depth-first search*) poboljšava prostornu složenost algoritma pretraživanja u širinu tako da uvijek generira prvo dijete prvog (najdubljeg) neproširenog čvora [9]. Implementacijsku razliku čini to da se sada generirani čvorovi dodaju u na početak, a ne na kraj list otvorenih. Možemo reći da takva lista otvorenih čvorova sada funkcioniра kao **stog**. Slika (3.4) prikazuje razlike obilaska stabla pretraživanja između pretraživanja u širinu i dubinu.



Slika 3.4: Razlike obilaska čvorova stabla kod algoritama BFS i DFS. Slika preuzeta iz [12]

Osnovna varijanta ovog algoritma nije niti potpuna niti optimalna zato što postoji mogućnost zaglavljivanja u beskonačnoj petlji i ne pretražuje se razina po razini. Iz istih razloga, DFS je preporučeno izbjegavati ako je u pitanju velika (ili beskonačna) maksimalna dubina stabla.

3.2.4. Iterativno pretraživanje u dubinu (IDFS)

Iterativno pretraživanje u dubinu (engl. *Iterative depth-first search*) važna je inačica osnovnog algoritma DFS zato što kombinira prednosti pretraživanja u širinu i dubinu te tako postiže optimalnost, potpunost te pogodnu vremensku i prostornu složenost. Za izvođenje algoritma zadužene su dvije metode. Vanjska metoda povećava brojač od 0 do *beskonačno* te trenutnu vrijednost brojača predaje unutarnjoj metodi. Unutarnja metoda takvu vrijednost koristi kao ograničenje dubine do koje se stablo pretražuje algoritmom DFS. Pseudokod (3.5) prikazuje način na koji su metode ugniježđene.

```
1  function iterativeDepthSearch(s0, succ, goal)
2      for k = 0 to infinity do
3          result = depthLimitedSearch(s0, succ, goal, k)
4          if result != fail then
5              return result
6
```

Programski isječak 3.5: Pseudokod ugniježđenih metoda iterativnog pretraživanja u dubinu

Postiže se pretraživanje razine po razine, ali i linearna prostorna složenost. Vremenska složenost ostaje eksponencijalna, ali je ovakva inačica algoritma pogodna za probleme s velikim prostorom stanja i nepoznatom dubinom rješenja.

3.3. Usmjereno pretraživanje

Usmjereno pretraživanje naziva se još i **heurističkim**. Posebnost ove strategije pretraživanja je korištenje informacija o prirodi problema koje nisu egzaktne kao kod slijepog pretraživanja. Korištenje takvih informacija moglo bi se interpretirati tako da unaprijed znamo u kojem smjeru se rješenje problema nalazi pa se tako ubrzava i sama pretraga. Važno je spomenuti kako korištenje heuristike definira pravila po kojima se problem može riješiti, ali ne garantira i samo rješavanje problema [10].

3.3.1. Heuristika

Heuristika je naziv za skup iskustvenih pravila o prirodi problema. Algoritam A* je primjer algoritma koji koristi takva pravila kako bi na što brži i što jeftiniji način bio dosegnut cilj. Implementacijski, heuristika se iskazuje heurističkom funkcijom koja svakom čvoru dodjeljuje njegovu heurističku vrijednost.

Jednostavan primjer koji može predložiti pogodnosti heuristike je korištenje zračnih udaljenosti kako bi se aproksimiralo trajanje i cijena putovanja između određenih točaka. Ako potrebno stići od točke A do točke B mrežom, koja se sastoji od još nekoliko međusobno povezanih točaka, ne znamo koliko će nam vremena biti potrebno za postizanje cilja niti koji put će nam omogućiti da naše putovanje bude što jeftinije. Činjenica da prije početka našeg putovanja posjedujemo informacije o zračnim udaljenostima između određenih točaka, uvelike će nam olakšati odabir puteva kako bi naše putovanje završilo na optimalan način.

3.3.2. Algoritam A*

Primjeri algoritama koji koriste heuristiku su **Pretraživanje "najbolji prvi"** (engl. *Greedy best-first search*), **Pretraživanje usponom na vrh** (engl. *hill-climbing search*) te najpoznatiji i najviše korišteni algoritam A* odnosno (engl. *A-Star search*). A* možemo shvatiti kao križanac algoritma pretraživanja s jednolikom cijenom i pohlepnog algoritma "najbolji prvi". Već se kod pohlepnih algoritama koristi heuristika kako bi se na temelju tih informacija pokušala optimizirati putanja do cilja no ne uzima se u obzir cijena puta i zato takvi algoritmi nisu optimalni. Ovaj će algoritam ostvariti svoju potpunost i optimalnost upravo tako da se čvorovi uspoređuju po funkciji **ukupne cijene** koja je zapravo zbroj heurističke vrijednosti čvora i cijene puta do tog čvora. Međutim, korištenje heuristike nam ne može samo po sebi jamčiti optimalnost. Kako bi algoritam A* bio optimalan, potrebno je da heuristika bude **optimistična**, ali

takva svojstva bit će objašnjena u idućem odjeljku.

Jedna od pogodnosti ovog algoritma također je i pogodnija vremenska i prostorna složenost koja je samo u najgorem slučaju eksponencijalna. U najboljem slučaju računa se kao umnožak **faktora grananja** i **veličine prostora stanja**. Ono što pridonosi takvom poboljšanju složenosti je korištenje liste zatvorenih čvorova koja se koristi za dodatnu provjeru prilikom umetanja čvorova u listu otvorenih. Pseudokod (3.6) služi za pojašnjenje algoritma A* na implementacijskoj razini.

```
1  function aStarSearch(s0, succ, goal, h)
2      openList = [initial(s0)]
3      closedList = []
4      while open != [] do
5          n = removeHead(openList)
6          if goal(state(n)) then
7              return n
8          closedList.add(n)
9          for m in expand(n, succ) do
10             if closedList.contains(m') or openList.contains(m') such that state(m') =
11                 state(m) then
12                 if price(m') < price(m) then
13                     continue
14                 else
15                     remove(m0, closedList and/or openList)
16                     insertSortedBy(totalCost, m, openList)
17             return fail
18 where totalCost(n) = price(n) + heuristic(state(n))
```

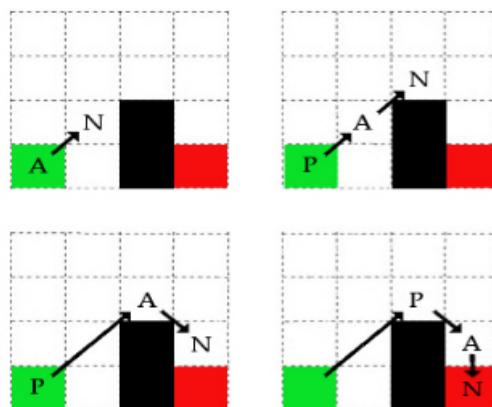
Programski isječak 3.6: Pseudokod algoritma A*

Pretraživanje kao argumente prima početno stanje $s0$, funkciju sljedbenika $succ$, listu ciljnih stanja $goal$, te heuristiku h . Prije svega inicijaliziraju se lista otvorenih čvorova s početnim stanjem te zatvoreni čvorovi kao prazna lista. Sve dok ima čvorova u listi otvorenih, provjerava se odgovara li trenutni čvor ciljnom stanju. Također, čvor se dodaje u listu zatvorenih i generiraju se njegovi sljedbenici funkcijom $expand$. Za svaki generirani sljedbenik slijedi provjera postoji li u listi otvorenih i zatvorenih čvorova neki čvor koji sadrži isto takvo stanje, a da smo do njega došli jeftinijim putem. Ako taj uvjet vrijedi, trenutni provjeravani čvor se preskače i nastavlja se na idući sljedbenik. U protivnom se, iz liste otvorenih i/ili zatvorenih, uklanja skup lji čvor, a jeftiniji se dodaje u listu otvorenih, sortirano prema ukupnoj cijeni (zbroj heurističke vrijednosti i cijene puta do tog čvora).

3.3.3. Modifikacija algoritma A*

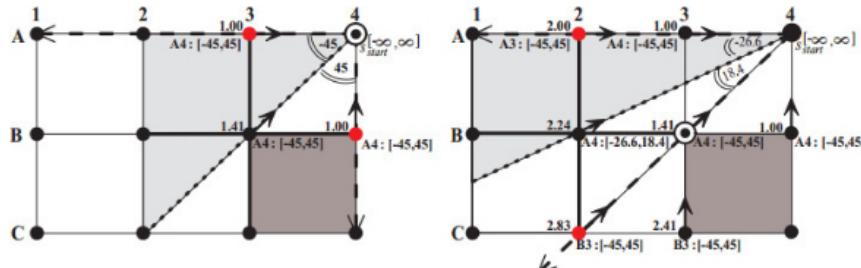
Postoje razne modifikacije algoritma A* koje omogućavaju njegovo kvalitetnije korištenje u raznim područjima pa tako i u digitalnim igrama. U ranijim poglavlјima je objašnjeno kako razvojno okruženje Unity teren pretvara u mrežu konveksnih poligona te da se na takvoj mreži izvodi traženje puteva između početne i ciljne točke. Osnovni se algoritam izvodi pretraživanje tako da se mrežom prolazi do susjednog poligona pod kutovima od isključivo 45 i 90 stupnjeva (engl. *zigzag style*). Postoji mogućnost da između početnog i ciljnog poligona postoji nešto slobodnog prostora među poligonima koji nisu povezani na takav način pa se zbog toga uvodi pretraživanje po svim kutovima [13]. Algoritmi koji koriste takav način pretraživanja su **Theta*** i **Phi***.

Slika (3.7) prikazuje način na koji algoritam Theta* dostiže ciljni poligon (označeno crvenom bojom) iz početnog (označeno zelenom bojom), a prepreke su predstavljene crnim poljima. Ključan trenutak postupka je drugi korak vizualizacije (gornja desna slika) u kojem je se zanemaruje točka A zato što između točke P i N postoji izravna vidljivost (jedna drugu "vide") [13].



Slika 3.7: Vizualizacija osnovnog algoritma Theta*. Slika preuzeta iz [13]

Phi* je samo nadogradnja algoritma Theta*. Dovoljno je samo spomenuti da se osnovna funkcionalnost ostvaruje tako da svaka ćelija posjeduje spektar kutova unutar kojih sljedbenik ćelije može biti pronađen (3.8).



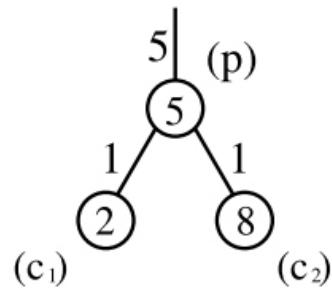
Slika 3.8: Vizualizacija spektra kutova za svaku ćeliju algoritma Phi*. Slika preuzeta iz [13]

3.3.4. Svojstva heuristike

Jedno od osnovnih svojstava heuristike, koje je ujedno i uvjet optimalnosti algoritma A*, je optimističnosti. Heuristika je **optimistična** ili **dopustiva** (engl. *optimistic, admissible*), ako za svaki čvor unutar stabla pretraživanja vrijedi da njegova heuristička vrijednost nikad nije veća od prave cijene do cilja. Možemo reći da se konačna udaljenost nikad **ne precjenjuje**. Kada heuristika ne bi bila optimistična, lako bi se moglo desiti da pretraga zaobilazi najoptimalniji put zato što se on čini skupljim nego što jest. S druge strane, ako se koristi heuristika koja nije optimistična (nego je **pesimistična**), smanjuje se broj generiranih čvorova (što bi potencijalno moglo pogodovati vremenskoj složenosti), ali se smanjuje kvaliteta rješenja. Postoje tri načina na koje se optimistična heuristika može oblikovati:

1. **Relaksacijom problema** prepostavlja se da je problem rješiv na neki nerealno jednostavan način, te se za vrijednosti heuristike uzimaju informacije o problemu dobivene takvim načinom rješavanja. Osigurava se optimističnost i konzistentnost.
2. **Kombiniranjem optimističnih heuristika** dobiva se jedna dominantna, koja će predstavljati rješenje s najvišim mogućim vrijednostima koje i dalje ne precjenjuju stvarne vrijednosti.
3. **Učenje heuristike** moguće je izvesti metodama strojnog učenja pri čemu se određuju i testiraju koeficijenti za različite značajke stanja.

Konzistentnost heuristike označava činjenicu da ukupna funkcija puta (zbroj vrijednosti heuristike i cijene puta do čvora) za svaki čvor u stablu monotono raste u odnosu na prethodni. Takva heuristika omogućit će nam da kod ponovljenih stanja ne trebamo provjeravati cijenu već zatvorenih čvorova zato što će ona uvijek i sigurno biti manja ili jednaka. Drugim riječima, svaki čvor koji prvi ispitamo i zatvorimo, bit će čvor s najmanjom cijenom za stanje koje taj čvor sadrži. Slika (3.9) prikazuje primjer **nekonzistentne** heuristike što je vidljivo na prijelazu iz čvora p do čvora $c1$: heuristička vrijednost čvora p (5) manja je od zbroja cijene prijelaza i heurističke vrijednosti čvora $c1$ (3).



Slika 3.9: Primjer nekonzistentne heuristike. Slika preuzeta iz [14]

4. Opis pokazne videoigre

Ovo poglavlje opisuje pokaznu igru koja je razvijena u svrhu demonstracije primjenjivosti algoritama za traženje puteva na računalno-generirane likove. Jedan od ciljeva implementacije je ostvariti atmosferičan dizajn okoline u kombinaciji s mehanikom u prvog lica. Igra je razvijena koristeći razvojno okruženje Unity.

4.1. Dizajn okoline

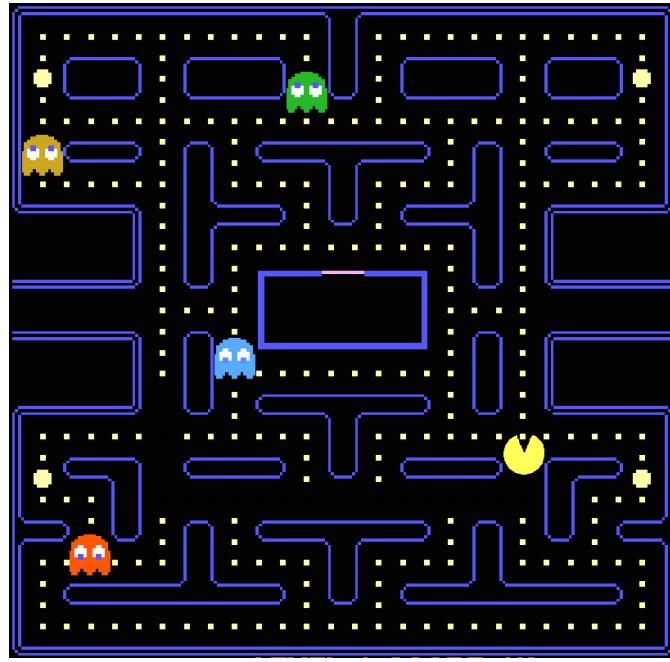
Pokazna igra zamišljena je kao akcijska s elementima strave. Kako bi elementi strave došli do izražaja, potrebno je okolinu dizajnirati na takav način da se, već samom arhitekturom terena, osvjetljenjem i zvukom, ostavlja dojam na igrača pri prvom ulasku u igru. U naredna dva potpoglavlja opisana je ideja dizajna okoline sa svim svojim elementima.

4.1.1. Labirint

Teren je sačinjen od labirinta koji svojim izgledom podsjeća na tamnicu. Kretanje igrača se ograničava na prostor veličine labirinta, a dodatno i preprekama od kojih se labirint sastoji. Tako se postiže dojam skučenosti prostora koji bi kod igrača za vrijeme slobodnog kretanja labirintom, a posebno prilikom interakcije s neprijateljima, trebao izazvati dodatnu nelagodu i povećati pritisak. Takav psihološki podražaj odrazit će se na donošenje odluka igrača, točnije, igrač će biti primoran pomnije planirati svoju taktiku kretanja. Pomoću kompleksnosti arhitekture labirinta bit će postignut prvi utjecaj zastrašujućeg i nelagodnog ambijenta na igrača.

Kao referencu, za ostvarenje strukture labirinta, uzimamo vrlo poznati labirint iz igre *Pac-Man* (4.1). Cilj je kreirati nekolicinu elemenata koji će predstavljati vrste prepreka te ih posložiti unutar zidova tako da je dobivena kompozicija sastavljena isključivo od prohodnih puteva za sve vrste likova. Za razliku od Pac-Mana, u ovoj igri, struktura labirinta se ne mijenja sa svakom prijeđenom razinom zato što je igra sastav-

ljena samo od jedne razine. Slika (4.1) prikazuje primjer karakterističnog labirinta iz spomenute igre sa svim svojim elementima.



Slika 4.1: Originalni labirint iz igre Pac-Man. Slika preuzeta iz [15]

4.1.2. Ambijent

Utjecaj strukture labirinta na osjećaj igrača bit će ostvaren isključivo putem **mehaničkih ograničenja**. Najvažniji dio postizanja želenog ozračja je oblikovanje osvjetljenja i zvuka, koji sačinjavaju ambijent, stvarajući tako **vizualna i auditivna ograničenja**.

Osvjetljenje

U skladu s poznatijim **tehnikama osvjetljenja** korištenim u filmovima strave ili sličnim videoigramama, potrebno je osigurati **prigušeno svjetlo** [16], koje će u našem slučaju biti blago crvenkaste boje. Intenzitet svjetla ovisi isključivo o udaljenosti igrača od izvora osvjetljenja, a kako bi se postigla dodatna razina prigušenja, koristi se pozamašna količina crne magle koja također smanjuje i vidno polje igrača. Također, korištena tehnika uključuje i **visoko-kontrastne sjene** posebno istaknute na pukotinama kamenih zidova tamnice.

Kako bi se pojačao osjećaj neizvjesnosti igrača, ako se u njegovom radijusu nalazi neprijatelj, razina osvjetljenja počinje se ciklički smanjivati i pojačavati. Takvo

treperenje svjetla simbolizira igraču da se opasnost nalazi u njegovojo neposrednoj blizini i pridonosi osjećaju napetosti.

Zvuk

Neizostavno je spomenuti utjecaj **zvuka** na ostvarenje ambijenta. Zvuk je u igri organiziran u dvije razine: **glazba** i **zvučni efekti**. Glazba je komponirana od nekoliko jednostavnih i dubokih tonova basa. Isprepliće se nekoliko zvučnih efekata poput kapanja vode i blagih šumova koji su globalizirani odnosno imaju jednak intenzitet neovisno o dijelu terena na kojem se igrač nalazi. Najvažniji zvučni efekt dodijeljen je neprijateljima, aktivira se prilikom ulaska u radijus igrača, a predstavlja govor kao izravnu poruku igraču da ga neprijatelj u tom trenutku vidi ("I see you"). Intenzitet govora neprijatelja reguliran je **Dopplerovim efektom** što znači da će ga igrač čuti glasnije ili tiše ovisno o udaljenosti od izvora zvuka.

4.2. Mehanika igre

S obzirom na manji broj raznovrsnih interaktivnih elemenata i ciljeve unutar igre, mehanika igre je vrlo jednostavna. Igrač upravlja glavnim likom unutar prvog lica. Omogućeno mu je kretanje (hodanje, trčanje i skakanje), rukovanje oružjem te sakupljanje novčića i municije. Mehanika neprijatelja počiva na sustavu za traženje puteva uz primjenu algoritma A* i metode RVO.

4.2.1. FPS Mehanika

Unutar FPS igara, središnji fokus smješta se na aktivnog agenta (glavnog lika), a između navigacije, pucanja i izbjegavanja opasnosti, postoje i drugi elementi koji održavaju igrača u stanju uzbuđenja [17]. S obzirom na to da će kompleksnost tih elemenata biti svedena na minimum, ono na što se posebno potrebno osvrnuti je upravljanje oružjem. Jedan od elementarnih ciljeva igre je razviti dotjeranu mehaniku pucanja s uglađenim animacijama. U tu svrhu, igrač tijekom cijele igre nosi pušku s ograničenim brojem metaka. Dodatni metci se periodički pojavljuju na nasumičnim mjestima labirinta, a u svakom trenutku je moguće imati maksimalno 4 metka. Nakon svakog ispaljenog metka, izvršava se brza animacija koja simbolizira trzaj puške i repetiranje, a prilikom izvršavanja animacije pucanje je onemogućeno.

Ograničena količina metaka predstavlja dodatni izazov za igrača pri čemu se od njega očekuje da pažljivo odabire kada i na koji način će koristiti oružje. Sudar metaka

vidljiv je na svim dijelovima terena, a posebno je istaknut (zvučni efekt i sustav čestica) prilikom pogotka neprijatelja što i je glavna svrha korištenja oružja. Tri pogotka neprijatelja označavaju njegovu smrt, odnosno povratak na početnu poziciju patroliranja no to će još biti detaljnije objašnjeno u narednim poglavljima. Igrač ne treba manualno repetirati oružje već se to obavlja automatski prilikom sakupljanja municije. Ako je broj metaka u tom trenutku manji od maksimuma, repetiranje se izvršava, a u protivnom sudsar igača s municijom se ignorira.

4.2.2. Interaktivni elementi

Dodavanje mogućnosti sakupljanja pojedinih elemenata kako bi se time ostvario neki krajnji ili sporedni cilj, uobičajena je praksa prilikom razvoja ove vrste digitalnih igara. Ova igra sadrži samo dva takva elementa: **novčići** i **municija**. Simbolika takvog odabira elemenata, kao i kod konstrukcije labirinta, počiva na već spomenutoj igri **Pac-Man**. Sakupljanje novčića pojavljuje se i u velikom broju drugih igara no tamo oni uglavnom predstavljaju kompenzacijsko sredstvo za nadogradnju glavnog lika. Novčići su ravnomjerno raspoređeni po svakom prohodnom dijelu labirinta pa se od igača očekuje da prođe čitavi labirint i sakupi sve novčice kako bi uspješno završio igru.

Druga simbolika leži u istovjetnosti municije u pokaznoj igri i trešnji u Pac-Manu. Oba elementa služe kako bi igaču dali prednost nad neprijateljima odnosno kako bi im bilo omogućeno da neprijatelje vrate u stanje u kojem su bili kad je igra tek započela.

4.3. Izazovi

Odluke igača u velikoj mjeri utječu na uspješnost završavanja igre. Do sad su u prethodnim poglavljima opisani neki načini na koje se diskretno utječe na igača, kako bi se umanjila sposobnost donošenja racionalnih odluka i tako otežalo ispunjavanje krajnjeg cilja igre. Korištene tehnike vezane su uz vizualne i auditivne podražaje, a uključuju i mehaničke restrikcije na igača. Međutim, kako bi takvi izazovi u tijeku igre uopće imali osnovu, potrebno je jasnije definirati neprijatelje i njihovo ponašanje.

4.3.1. Cilj igre

Igra ima samo jedan krajnji cilj: sakupiti sve novčiće iz labirinta. Također, sakupljanje municije istovjetno je sporednom cilju te ono nije ni na koji način krucijalno niti potrebno za završavanje igre. Točnije, s obzirom na to da se novčići sakupljaju izravnim dodirom s igračem, za prolazak igre dovoljna je samo dobra strategija kretanja labirintom. Sve odluke koje igrač unutar labirinta donosi, motivirane su izbjegavanjem susreta s neprijateljima.

4.3.2. Vrste neprijatelja

Posljednja referenca na Pac-Mana bit će korištenje **duhova** kao figure neprijatelja, koji pretražuju labirint, u svrhu sprečavanja igrača u ostvarenju cilja igre. Duhovi su zapravo inteligentni agenti koji, uz pomoć algoritma za traženje puteva, patroliraju labirintom kako bi pronašli igrača. U slučaju da se igrač nalazi u njihovom vidnom polju, patroliranje je privremeno obustavljeno i duhovi love igrača sve dok on ponovno ne nestane iz njihovog vidnog polja. Ako se igrač ne nalazi u vidnom polju, ali je sve-jedno u neposrednoj blizini duhova, prisutne su indikacije na opasnost manipulacijom osvjetljenja i zvuka.

Na početku igre, svi duhovi imaju istu polazišnu točku na sredini mape i svi se kreću jednakim brzinama. Ono po čemu se duhovi međusobno razlikuju je smjer u kojem pretražuju labirint. Ostvarenje labirinta unutar trodimenzionalnog prostora podrazumijeva zauzeće većeg dijela terena, a kako bi se optimizirala izazovnost igre, potrebno je odabrati **veći broj duhova** koji pretražuju **manje dijelove terena**. Mapom patrolira ukupno 8 duhova od kojih polovica pripada vrsti "*izviđač*" (engl. scout), a druga polovica "*branič*" (engl. defender). Četvorica izviđača pretražuju dijelove udaljenije od središta mape dok braniči patroliraju samim središtem i "brane" svoju polazišnu točku. Svaki duh ima tri "života" od kojih se, sa svakom kolizijom s metkom, oduzima po jedan. Ako pojedinom duhu nije preostao niti jedan "život", vraća se na svoju polazišnu točku s početka igre (središte mape) i ponovno započinje svoje patroliranje. Izravna kolizija igrača s duhom uzrokovat će neuspjeh i kraj igre.

4.4. Tijek igre

Igra započinje pojavljivanjem igrača u krajnjem kutu labirinta. Strategija prolaska labirintom ovisi o igraču i prilikom svakog pokretanja igre može biti drugačija. Da bi igrač završio igru, potrebno je da prođe svakim prohodnim putem labirinta kako bi sakupio sve novčiće. Korištenje mehanike pucanja nije nužno, ali u velikoj mjeri olakšava potencijalne sukobe s duhovima. Parametri patroliranja optimizirani su kako bi se postigla neizbjegna interakcija s duhovima, a zadaća igrača je donijeti što prikladniju odluku o izbjegavanju ili suočavanju s opasnostima.

5. Razvoj pokazne videoigre

U ovom poglavlju bit će opisana konkretna implementacija opisanih koncepata pokazne videoigre unutar razvojne okoline Unity. Za svaki element, koji pojedino potpoglavlje opisuje, priložene su pripadajuće skripte pisane u programskom jeziku C# također kao i postavke objekata unutar sučelja. Svi korišteni 3D modeli i grafički elementi preuzeti su iz *Unity Asset Storea*.

5.1. Razvojna okolina Unity

Unity je više-platformska razvojna okolina koja omogućuje izradu dvodimenzionalnih i trodimenzionalnih digitalnih igara. Osim za razvoj računalnih igara, postoji podrška i za konzole, proširenu stvarnost te mobilne igre i simulacije. Sadrži dodatno sučelje za programiranje temeljeno na programskim jezicima Java i C#, a omogućen je i uvoz dodatnih integriranih razvojnih okruženja koja se koriste za oblikovanje koda (npr. *Microsoft Visual Studio*). Unity omogućava i olakšava uvoz grafičkih elemenata te njihovu pretvorbu u objekte koji se kao takvi mogu koristiti u kontekstu generiranog dvodimenzionalnog ili trodimenzionalnog svijeta. Korištenje 3D predloška daje razne mogućnosti prilagođavanja modela osvjetljenja i sjenčanja u grafičkom prototičnom sustavu (engl. *Rendering Pipeline*), a općenito je olakšano i dodavanje fizičkih svojstava, umreženosti, zvuka i animacija. U ovom radu korištena je inačica Unityja 2020.3.18f1.

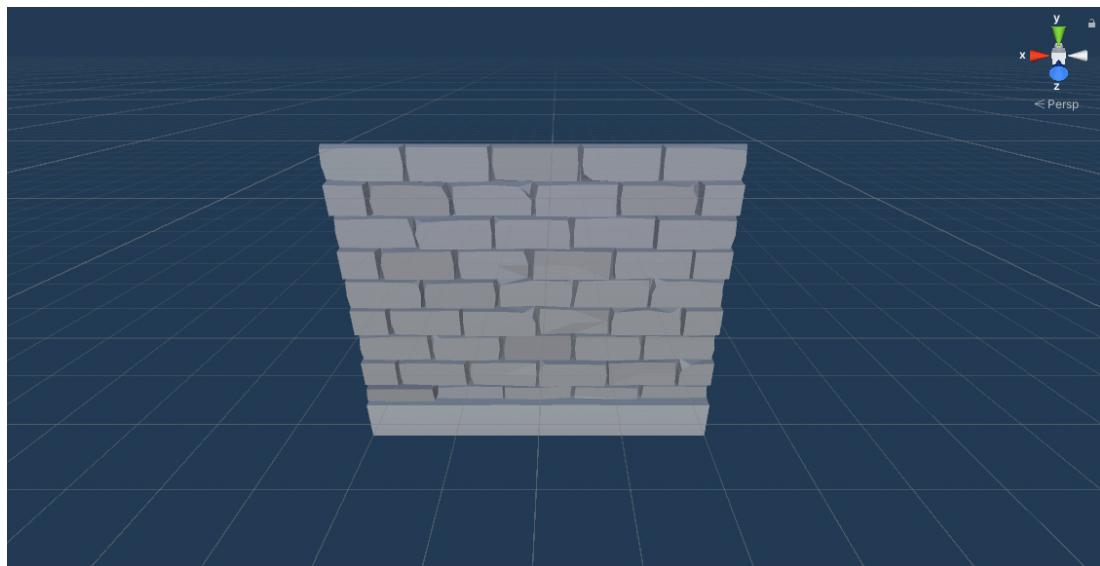
5.2. Postavljanje scene

Prije dodavanja igrača i njegovih mehaničkih svojstava na scenu, potrebno je unutar 3D prostora konstruirati teren kojim se igrač može kretati. U ovom slučaju, cilj je oformiti prohodan labirint koji se sastoji od dvije izdužene plohe (pod i krov) te većeg broja zidova koji definiraju puteve unutar labirinta. Nakon što labirint zadovoljava

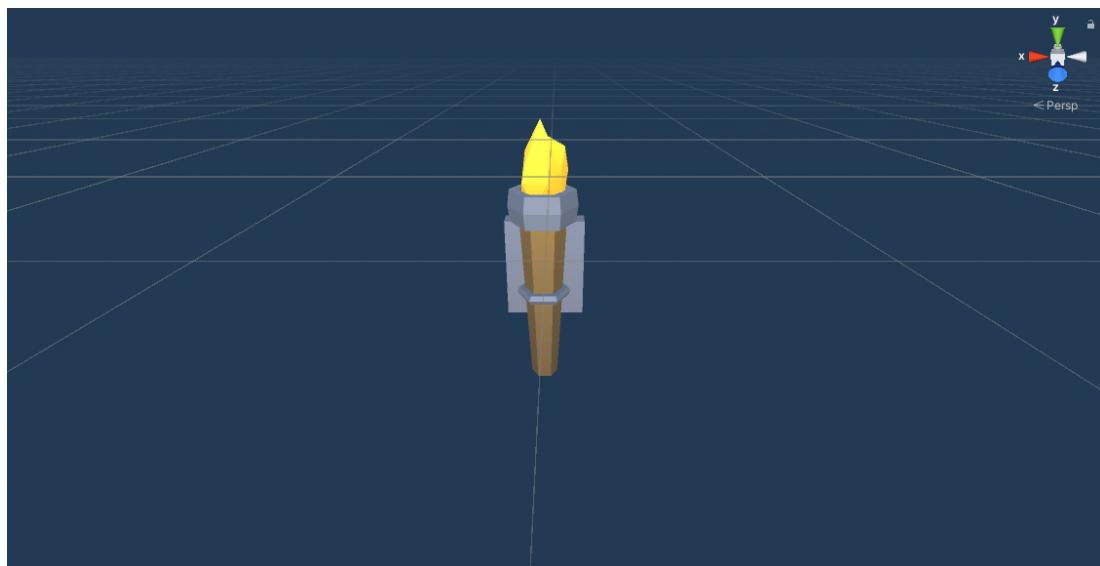
sve preduvjete koji bi osigurali funkcionalnost kretanja igrača terenom, postavljamo putne točke koje će se kasnije koristiti za implementaciju navigacijskog sustava.

5.2.1. Izrada terena

Preuzet je skup modela kojima je moguće modelirati čitavu tamnicu no jedini modeli koje koristimo su zidovi i baklje (Slike 5.1 i 5.2).

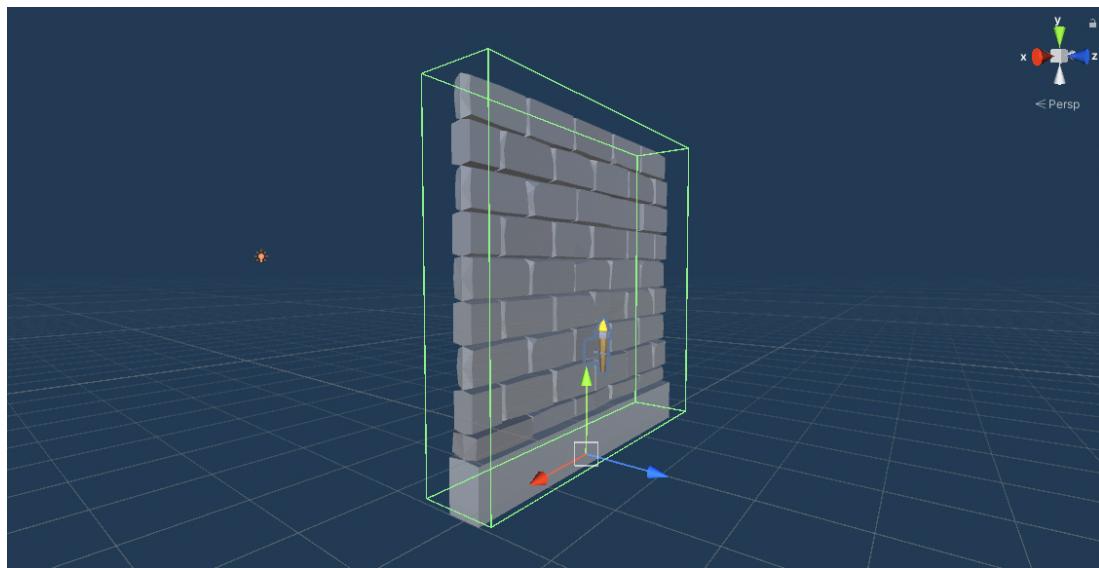


Slika 5.1: Model zida preuzet iz paketa "Low Poly Dungeons Lite"

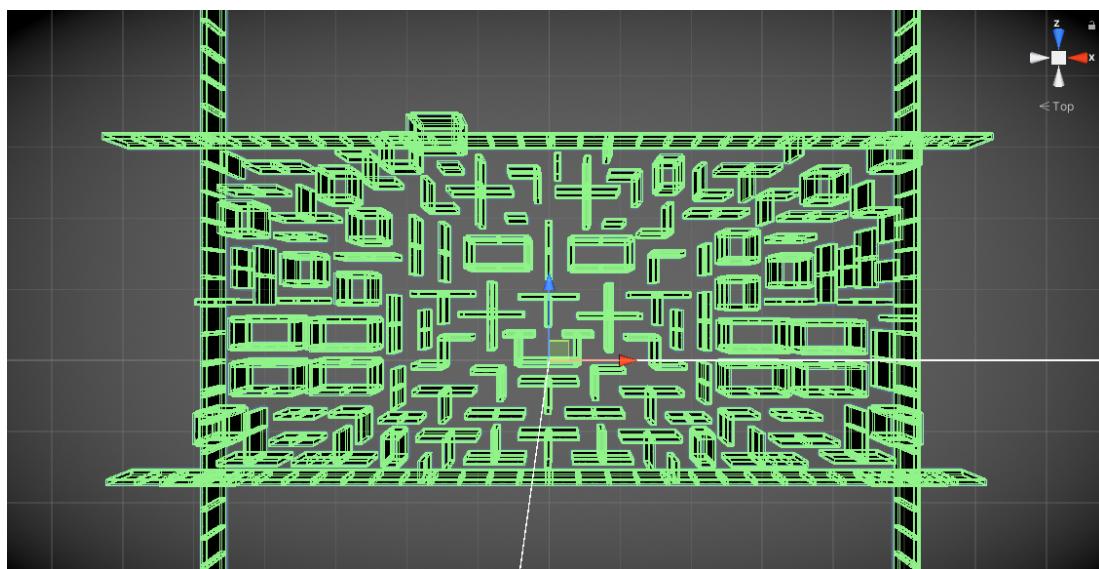


Slika 5.2: Model baklje preuzet iz paketa "Low Poly Dungeons Lite"

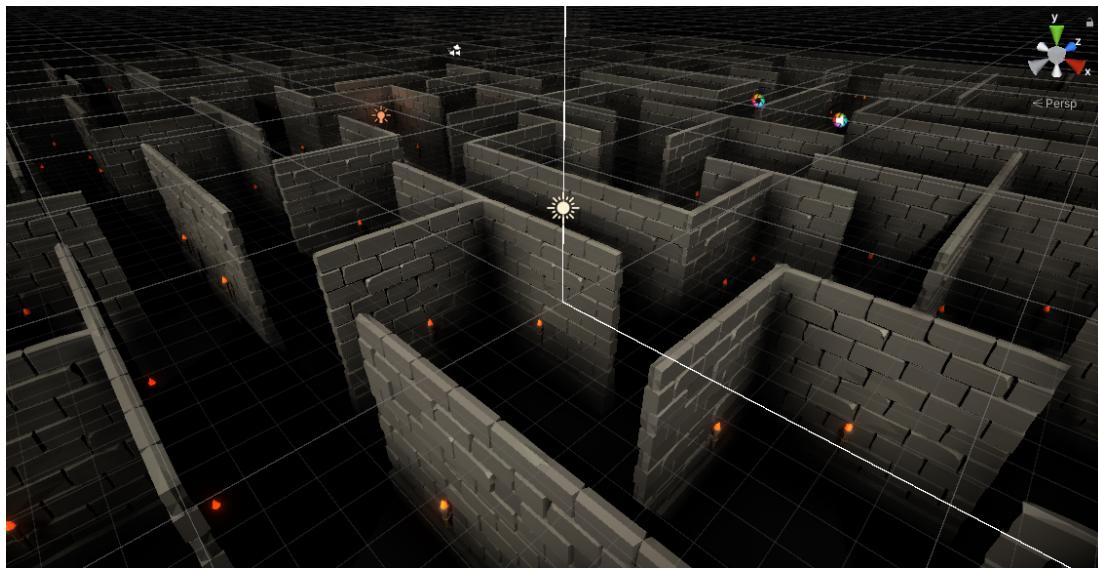
Kako bi mogli jednostavnije kreirati prohodne puteve, potrebno je duplicirati model zida i spojiti ga u jedinstveni predložak (engl. *prefab*) dvostranog zida koji će se postavljati čitavim labirintom. Takvom predlošku, sa svake strane dodajemo još i baklju za koju želimo da bude točkasti izvor osvjetljenja labirinta (Slika 5.3). Konačno, strukturi dodjeljujemo komponentu *Box Collider* kako bi se jasno mogla detektirati kolizija između igrača i zida. Predložak je višestruko duplikiran i spajan u nekoliko različitih varijacija, koje tada, pažljivo organizirane unutar vanjskih zidova tavnice, tvore završnu verziju labirinta (Slika 5.4).



Slika 5.3: Kreirani predložak dvostranog zida s bakljama kao izvorom osvjetljenja

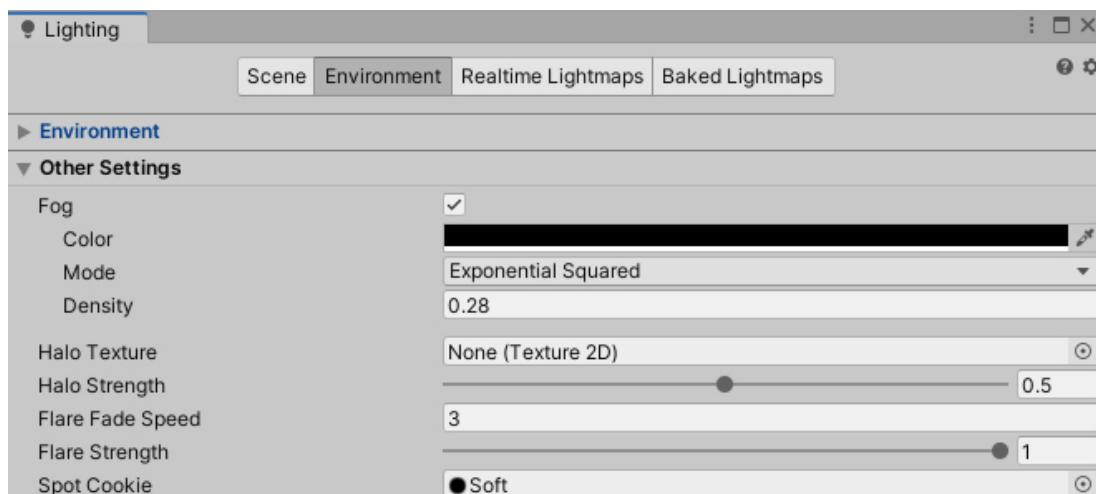


Slika 5.4: Tlocrt finalne strukture labirinta



Slika 5.5: Trodimenzionalni prikaz unutrašnjosti labirinta

Dodavanjem jednostavne crne izdužene plohe, koja predstavlja pod tamnice, završava modeliranje arhitekture labirinta i preostaje samo podešiti osvjetljenje kako bi se postigao željeni ugodaj. Smanjuje se vidno polje igrača tako da se u postavkama osvjetljenja isključuju komponente *Skybox* i *Sun Source*, a uključuje komponentu *Fog* koja će unutar cijelog labirinta stvoriti maglu željene boje i gustoće.



Slika 5.6: Prikaz dodatnih postavki osvjetljenja unutar sučelja

5.2.2. Putne točke

S obzirom na to da nam predstoji izrada navigacijskog sustava, a labirint je u potpunosti oblikovan, ispunit ćemo svaki prohodni put labirinta putnim točkama (engl. *Waypoints*). Putne točke su jednostavni prazni objekti pomoću kojih će duhovi navigirati labirintom. Svaki objekt koji predstavlja putnu točku sadrži skriptu za iscrtavanje obojane sfere što nam omogućuje bolju vidljivost putnih točaka unutar scene. Važno je napomenuti kako putne točke nisu vidljive za vrijeme igre već samo unutar scenskog prikaza u sučelju.

```
1  using UnityEngine;
2  public class Waypoint : MonoBehaviour
3  {
4      public float radius = 0.8f;
5      private void OnDrawGizmos()
6      {
7          Gizmos.color = Color.green;
8          Gizmos.DrawWireSphere(transform.position, radius);
9      }
10 }
11 }
```

Programski isječak 5.7: Skripta za iscrtavanje sfere putnih točaka

Svakom će duhu biti dodijeljeno nekoliko putnih točaka koje on ciklički obilazi. Prema tome, potrebno ih je smisleno postaviti unutar labirinta kako bi se osiguralo da duhovi patroliraju čitavim terenom. Grupirat ćemo putne točke za svakog od 8 duhova, ovisno o dijelu terena kojim taj duh treba patrolirati. Primjerice, za duha koji će obilaziti gornji lijevi dio labirinta, postavljamo putne točke kao što je prikazano na slici (5.8).

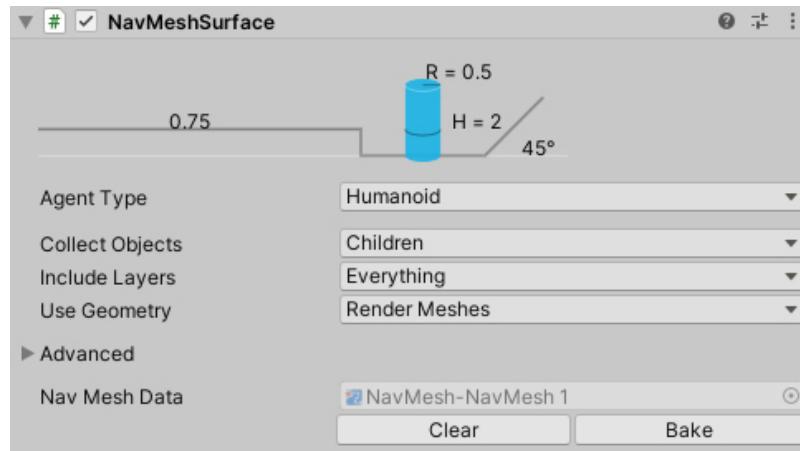


Slika 5.8: Prikaz jedne grupe postavljenih putnih točaka

5.3. Izrada navigacijske mreže

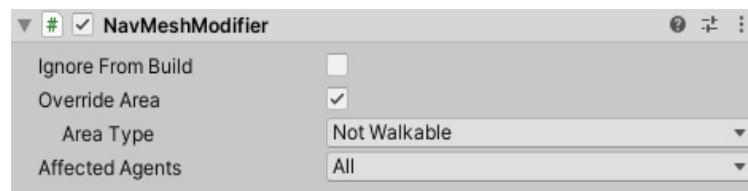
Proces izrade navigacijske mreže, odnosno "pečenje" (engl. *baking*), podrazumijeva odabir objekata u hijerarhiji od kojih će se mreža sastojati te podešavanje komponenti za odabrane objekte unutar kartice *Navigation*. Umjesto da konstruiramo običnu navigacijsku mrežu, konstruirat ćemo **dinamičku** čije komponente nisu dio standarde distribucije sustava Unity pa su dohvaćene sa službenog GitHub repozitorija *Unity-Technologies*.

Kreiran je prazni objekt imena "NavMesh" čija su djeca krovne i podne plohe kao i svi zidovi labirinta. Također, dodijeljena mu je komponenta *NavMeshSurface* koja, definirano postavkama komponente, dohvaća djecu objekta kako bi se izgradila mreža (Slika 5.9).



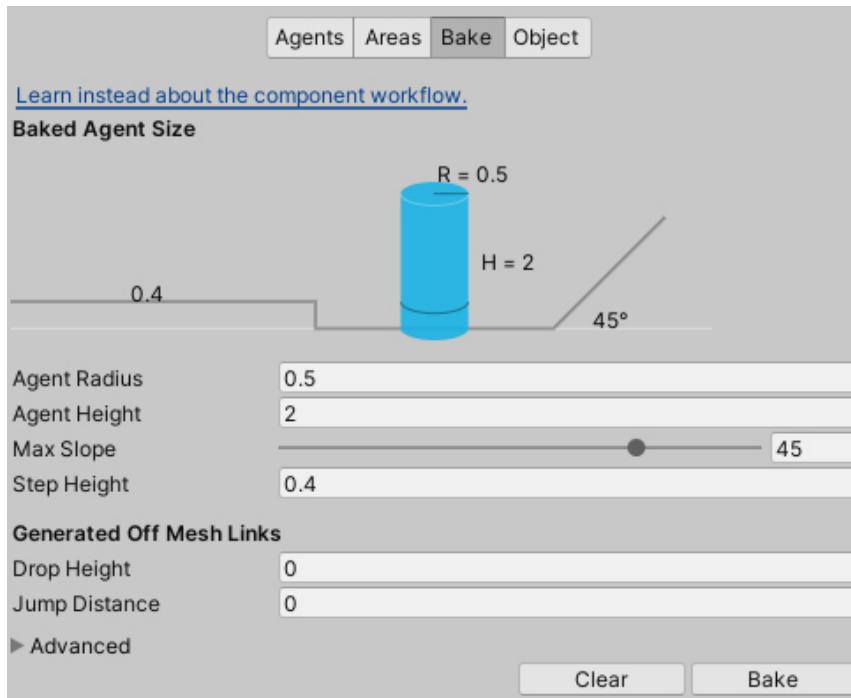
Slika 5.9: Postavke komponente *NavMeshSurface*

Svakom zidu dodaje se komponenta *NavMeshModifier* da bi se označili kao neprohodni (engl. *Not Walkable*) (Slika 5.10).



Slika 5.10: Postavke komponente *NavMeshModifier*

Na posljetku, unutar kartice *Navigation* postavljaju se finalne postavke vezane uz agenta koji se kreće navigacijskom mrežom (Slika 5.11) te se pritiskom na *Bake* stvara navigacijska mreža sačinjena od djece objekta "NavMesh".



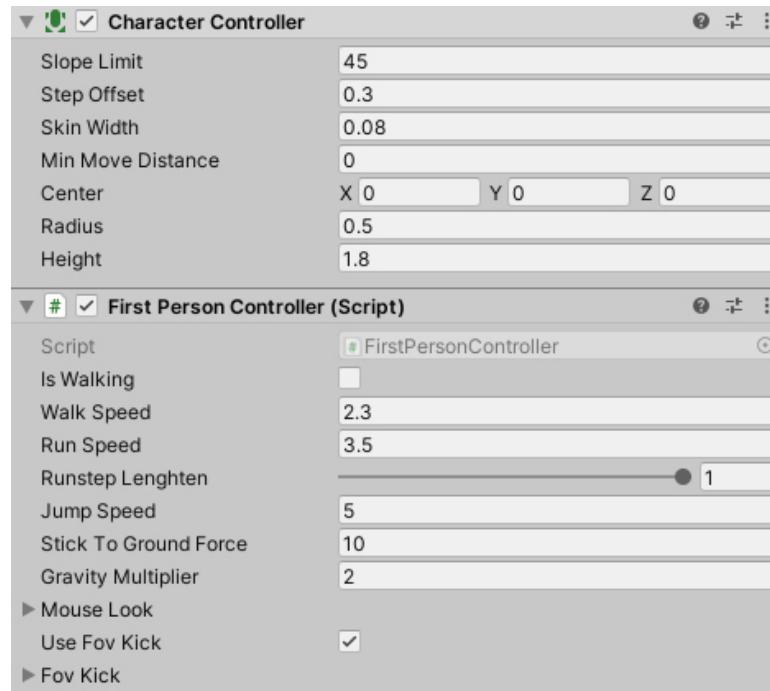
Slika 5.11: Postavke agenta unutar kartice *Navigation* neposredno prije izgradnje mreže

5.4. Igrač

Kreiran je novi objekt imena i oznake "Player". Kako bi ostvarili da kamera prati igrača i da svijet vidimo iz prvog lica, objekt koji sadrži kameru postavlja se kao dijete objekta igrača. Također, kao drugo dijete kreiramo dodatnu putnu točku čiji će položaj uvijek biti jednak položaju glavnog lika. To će nam kasnije omogućiti da duhovi privremeno zaustave svoj ciklus obilaska unaprijed definiranih putnih točaka i da slijede samo onu koja se nalazi na igraču.

5.4.1. Mehanika kretanja

Zbog jednostavnosti dodajemo unaprijed definiranu komponentu *Character Controller* i skriptu *First Person Controller*. Navedene komponente omogućuju nam da bez dodatnog pisanja koda ostvarimo kretanje u prvom licu s proizvoljnim specifikacijama. Izravno unutar sučelja, odmah po dodavanju komponenti, podešavaju se brzine kretanja i hodanja, duljina koraka, utjecaj gravitacije na skok, efekt pomicanja glave (kamere) prilikom hodanja i slično. Nakon pokretanja igre, moguće se kretati labirintom pomoću standardiziranog unosa (tipke W, A, S, D te pomicanje miša), a dodatnom optimizacijom parametara skripte dolazimo do odgovarajućeg osjećaja kontroliranja igrača.



Slika 5.12: Konačne postavke komponenti za kontrolu igrača

5.4.2. Mehanika pucanja

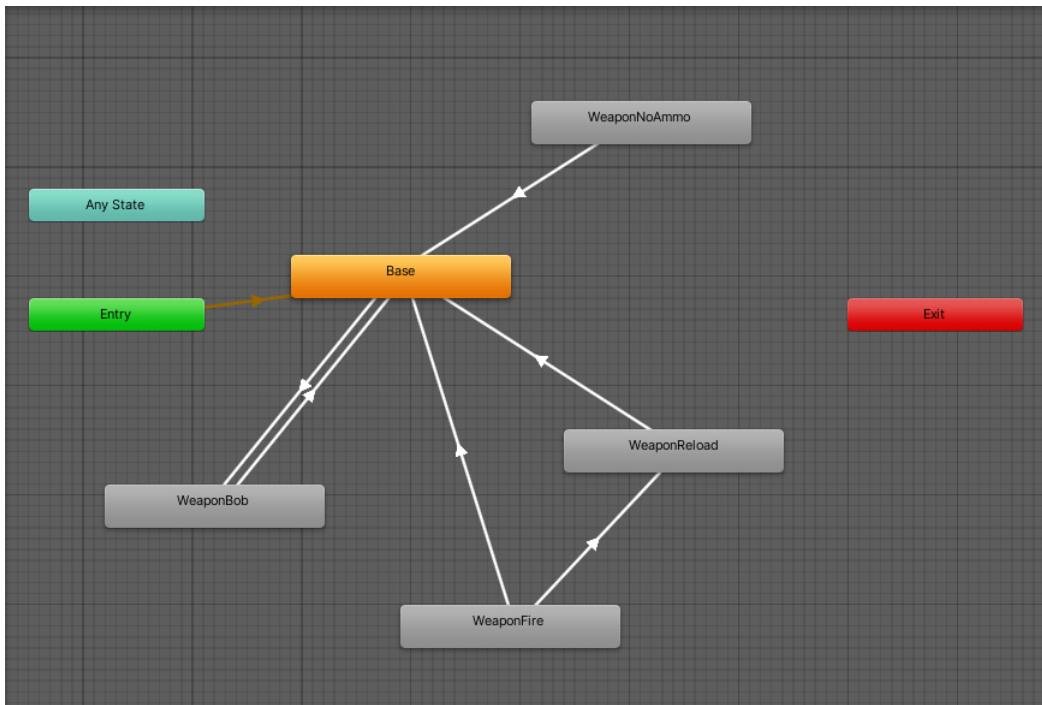
Iz paketa "Shotgun Set" preuzet je model oružja koji je, u obliku predloška, sastavljen od više zasebnih dijelova. To nam omogućuje da svaki od tih dijelova možemo animirati kako bi ostvarili željeno ponašanje oružja prilikom pucanja, punjenja ili kretanja kroz labirint. Unutar objekta koji sadrži kameru, kreirana su dva nova objekta: "WeaponPivot" i "Shotgun" koji predstavljaju model oružja. Dodatno, nišan je prikazan u obliku dvodimenzionalne grafike usidrene na središtu ekrana.



Slika 5.13: Model oružja iz pogleda igrača

"WeaponPivot" će služiti kao točka težišta oružju, oko koje se oružje rotira prilikom neke od međusobno povezanih animiranih akcija. Fizikalna svojstva puške prikazana su skupinom od četiri animacije:

1. "**WeaponBob**" je mala ciklička translacija i rotacija oružja koja je prisutna kad god je igrač u pokretu. Služi dočaravanju težine oružja jednako kao i pomicanja ruke lika prilikom hodanja ili trčanja.
2. "**WeaponNoAmmo**" aktivira se prilikom pokušaja pucanja iz prazne puške i daje naznaku igraču da trenutno nema metaka.
3. "**WeaponReload**" se izvodi nakon svakog ispaljenog metka ili prilikom skupljanja municije. Predstavlja akciju repetiranja oružja.
4. "**WeaponFire**" započinje istovremeno po ispaljivanju metka i predstavlja trzaj oružja.



Slika 5.14: Skupina međusobno povezanih animacija oružja unutar komponente *Animator*

Mehanika pucanja kontrolirana je skriptom "Shotgun" koja se nalazi na istoimenom objektu. Kolizija metka izvodi se mehanizmom *Raycast* pri čemu se stvara i pušta zraka u smjeru obilježenom pritiskom lijeve tipke miša. Dodatno, stvorena su dva sustava čestica "Muzzle Flash" i "Impact Effect" koji se aktiviraju kako bi dočarali bljesak prilikom pucanja te sudar metka i nekog objekata scene.

Iz skripte se izdvajaju pojedini dijelovi zaduženi za provjeru unosa i količine metaka te mehaniku detekcije kolizije metka s ostalim objektima. Svim značajnijim dijelovima koda, zbog boljeg razumijevanja, priloženi su odgovarajući komentari.

```

1 void Update()
2 {
3     //Ako je pritisnut lijevi klik misa i omoguceno je pucanje
4     if (Input.GetButtonDown("Fire1") && !cantShoot)
5     {
6         if (bullets > 0)
7         {
8             //Ako je preostao samo jedan metak promijeni boolean vrijednost unutar
9             animatora
10            if (bullets == 1)
11            {
12                shootingAnimator.SetBool("lastBullet", true);
13            }
14            Shoot();
15        }
16        //Ako vise nema metaka pokreni no ammo animaciju i zvuk
17        else
18        {
19            shootingAnimator.Play("WeaponNoAmmo", -1, 0f);
20            asc.PlayOneShot(noAmmo);
21        }
22    }
23    ...

```

Programski isječak 5.15: Dio skripte "Shotgun" zadužen za provjeru unosa mišem

```

1 void Shoot()
2 {
3     //Onemoguci pucanje sve dok puska nije repetirana
4     cantShoot = true;
5     //Sviraj zvuk pucnjave
6     asc.PlayOneShot(shot);
7     //Smanji broj metaka i azuriraj broj metaka unutar korisnickog sucelja
8     bullets--;
9     UpdateShellImages();
10    //Pokreni animaciju za trzaj oruzja
11    shootingAnimator.Play("WeaponFire", -1, 0f);
12    //Pokreni sustav cestica "MuzzleFlash"
13    MuzzleFlash.Play();
14    //Ostvari pucnjavu pomocu Raycast sistema
15    RaycastHit hit;
16    if (Physics.Raycast(fpsCam.transform.position, fpsCam.transform.forward, out
17        hit, shotgunRange))
18    {
19        //Zabiljezi je li zraka udarila u duha i ako jest nacini mu stetu
20        Target target = hit.transform.GetComponent<Target>();
21        if (target != null)
22        {
23            target.TakeDamage(shotgunDamage);
24            hit.transform.GetComponent<AudioSource>().PlayOneShot(ghostShot);
25        }
26        //Na mjestu u koje je zraka udarila stvori predlozak koji predstavlja efekt
27        //udarca metka u neki predmet
28        GameObject impactGo = Instantiate(ImpactEffect, hit.point, Quaternion.
29        LookRotation(hit.normal));
30        Destroy(impactGo, 2f);
31    }
32    //Nakon odredjenog vremena, repetiraj pusku i omoguci pucanje
33    Invoke("ReloadSound", 0.3f);
    Invoke("EnableShooting", 1.0f);
}

```

Programski isječak 5.16: Funkcija "Shoot" iz skripte "Shotgun" koja kontrolira koliziju metka i ostalih objekata na sceni



Slika 5.17: Aktivacija sustava čestica bljeska prilikom trzaja i ispaljivanja metka



Slika 5.18: Sustav čestica aktiviran kolizijom metka i duha

5.4.3. Detekcija kolizije

Na objektu "Player" nalazi se skripta imena "PlayerController" koja je zadužena za detekciju kolizije između igrača i svih elemenata za sakupljanje kao i sa *Collider* komponentama koje se nalaze na duhovima. Također unutar ove skripte, prati se napredak prema ciljnog stanju pa se ovisno o igračevom uspjehu ili neuspjehu izvršava niz operacija koje rezultiraju informacijama o završetku igre.

U ovom dijelu bit će priložen samo dio koda koji se odnosi na detekciju kolizije i neke od akcija koje se povodom tih kolizija izvršavaju.

```
1 void OnCollisionEnter(Collision col)
2 {
3     //Prilikom kolizije sa tijelom duha sviraj odgovarajuci zvuk i onemoguci
4     //daljnje akcije igraca
5     if (col.gameObject.tag == "Ghost")
6     {
7         playerSource.PlayOneShot(death);
8         DisableFunctions(col);
9     }
10 }
11 }
```

Programski isječak 5.19: Kolizija s običnim *Colliderima*

```

1 void OnTriggerEnter(Collider col)
2 {
3     //Ako se duh nalazi u radijusu igrača aktiviraj manipulaciju svjetlosti i zvuka
4     //kao naznake
5     if (col.gameObject.tag == "GhostRange")
6     {
7         LightSwitching.startSwitching = true;
8         playerSource.PlayOneShot(iSeeYou);
9     }
10    //Ako je sakupljen novcic sviraj odgovarajući zvuk, uništi objekt novcica i
11    //azuriraj brojac
12    if (col.gameObject.tag == "Coin")
13    {
14        playerSource.PlayOneShot(collectCoin);
15        UpdateCoinCounter();
16        Destroy(col.gameObject);
17    }
18    //Ako je sakupljena municija, ako broj metaka vec nije maksimiziran, uništi
19    //objekt municije i daj signal da je sakupljena
20    if (col.gameObject.tag == "Ammo")
21    {
22        if (Shotgun.bullets != 3)
23        {
24            collectAmmo = true;
25            Destroy(col.gameObject);
26        }
27    }
28 }
```

Programski isječak 5.20: Kolizija s Colliderima tipa *Trigger*

Ako je detektirana kolizija s tijelom duha, zaustavljaju se daljnje mogućnosti kontroliranja igrača te će cijela scena biti ponovno pokrenuta nakon izvršavanja završnih animacija.

```

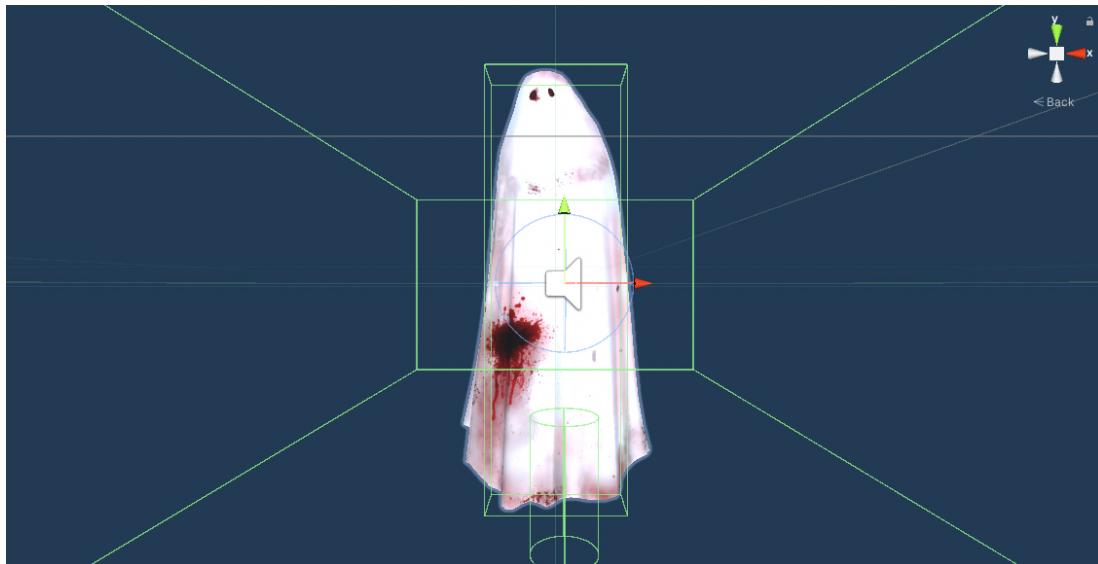
1 void DisableFunctions(Collision col)
2 {
3     //Zamrzavanje igraca i duha sa kojim je zabiljezena kolizija
4     ghostRb.constraints = RigidbodyConstraints.FreezeAll;
5     playerRb.constraints = RigidbodyConstraints.FreezeAll;
6     //Iskljucivanje komponenti zaduzenih za kretanje
7     playerFirstPersonController.enabled = false;
8     playerCharacterController.enabled = false;
9     Destroy(col.gameObject);
10    ...
11    //Iskljucivanje elemenata korisnickog sucelja i oruzja
12    GameObject.Find("HudCanvas").gameObject.SetActive(false);
13    GameObject.Find("WeaponPivot").gameObject.SetActive(false);
14    //Pokretanje zavrsnih animacija
15    GameObject.Find("PlayerCamera").gameObject.GetComponent<Animator>().Play("DeathCamera", -1, 0f);
16    Invoke("ShowGhost", 1f);
17
18}
19 void ShowGhost()
20{
21    scaryGhost.SetActive(true);
22    //Pozivanje funkcije koja ispisuje "Game Over" i ponovno pokreće scenu
23    Invoke("showGameOverCanvas", 3f);
24}
25

```

Programski isječak 5.21: Funkcija koja zaustavlja mogućnost kretanja i pucanja te poziva završne animacije

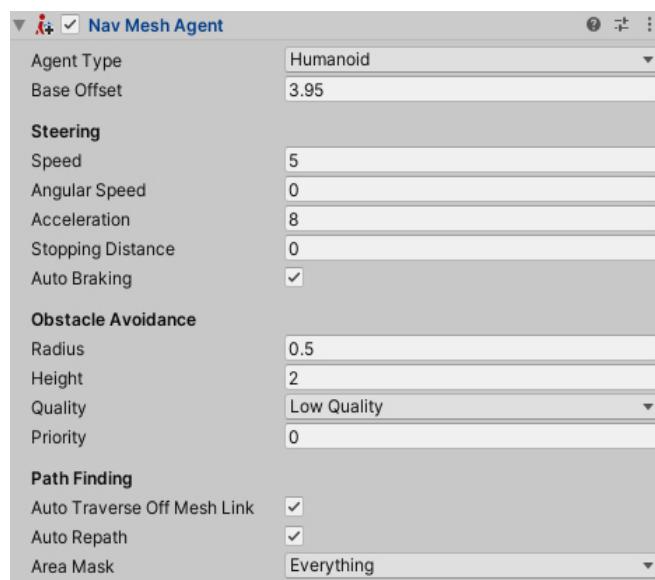
5.5. Oblikovanje neprijatelja

Iz paketa "Ghost low poly" uvezen je model duha sa svim pripadajućim animacijama i materijalima (Slika 5.22). Dodatno, dodijeljene su komponente *Box Collider*, *Rigidbody* i *NavMeshAgent* te skripta "Target".



Slika 5.22: Gotovi model duha iz paketa "Ghost low poly"

NavMeshAgent omogućit će objektu da bude prepoznat kao inteligentni agent koji se kreće izgrađenom navigacijskom mrežom. Unutar sučelja je moguće mijenjati parametre koji se odnose na mehanička svojstva intelligentnog agenta (Slika 5.23).



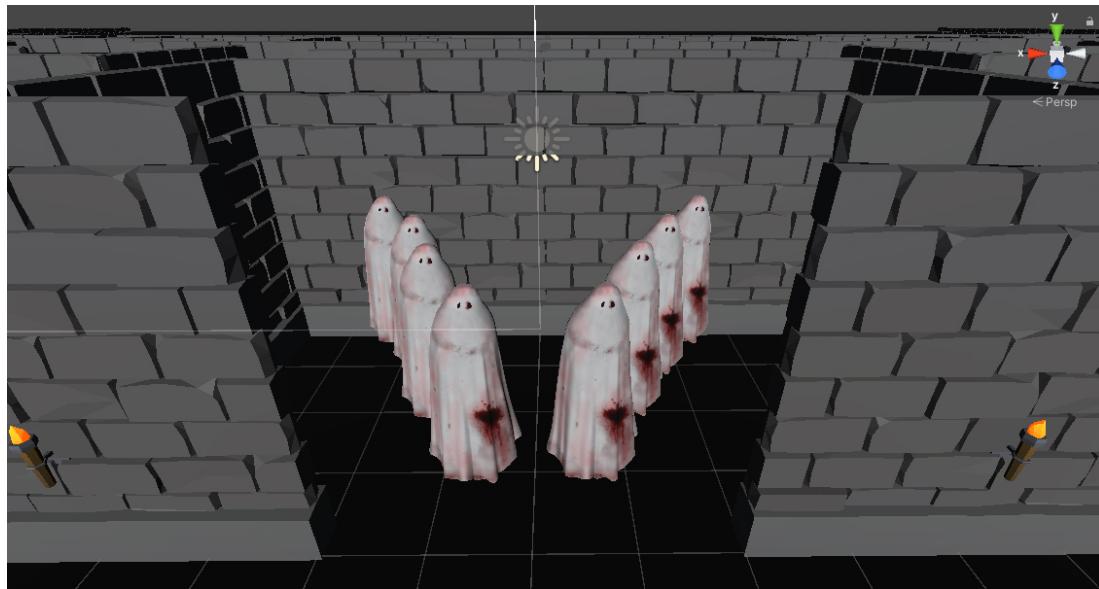
Slika 5.23: Konačni optimalni parametri intelligentnog agenta

Jednostavna skripta "Target", prilikom kolizije s metkom, ažurira štetu načinjenu na duhu. Ako su duha pogodila tri metka, njegova trenutna lokalna pozicija postaje jednaka početnoj poziciji (prije patroliranja).

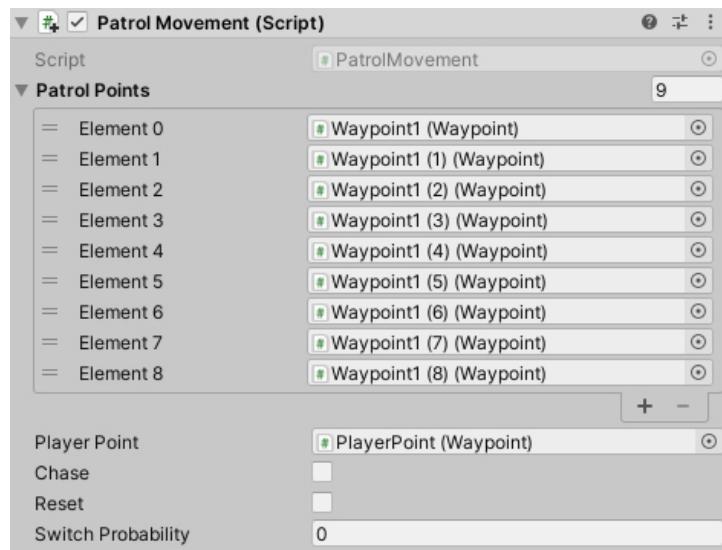
```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class Target : MonoBehaviour
6  {
7      public float health = 30f;
8
9      public void TakeDamage(float amount)
10     {
11         //Umanji varijablu health za jacinu metka i provjeri je li health trenutno 0
12         health -= amount;
13         if (health <= 0f)
14         {
15             Die();
16         }
17     }
18     //Resetiraj poziciju duha
19     void Die()
20     {
21         this.gameObject.GetComponent<PatrolMovement>().reset = true;
22         health = 30f;
23     }
24 }
```

Programski isječak 5.24: Skripta "Target" koja definira rezultat kolizije metka i duha

Dupliciranjem trenutnog predloška duha dobiva se osam zasebnih objekata koji se postavljaju na početnu poziciju unutar labirinta (Slika 5.25). Kako bi ostvarili funkcionalnost patroliranja, svakom duhu dodajemo skriptu "PatrolMovement" koja prima jedinstveno polje putnih točaka za svakog duha. Putne točke dodjeljuju se duhovima manualno unutar sučelja (Slika 5.26).



Slika 5.25: Duhovi na svojim početnim pozicijama u središnjem dijelu labirinta



Slika 5.26: Postavke skripte "PatrolMovement" za jednog od osam duhova

```

1  using System.Collections.Generic;
2  using UnityEngine;
3  using UnityEngine.AI;
4
5  public class PatrolMovement : MonoBehaviour
6  {
7      public List<Waypoint> patrolPoints;
8      public Waypoint playerPoint;
9      private Waypoint currentWaypoint;
10     public bool chase, reset, traveling, patrolForward;
11     public float changeDirectionProbabilitiy;
12     private Vector3 beginningPosition;
13     private Quaternion beginningRotation;
14     NavMeshAgent navMeshAgent;
15     int currentPatrolIndex;
16
17     void Start()
18     {
19         //Inicijalizacija pocetne pozicije te boolean vrijednosti za resetiranje
20         //pozicije i lovljenje igraca
21         beginningPosition = this.gameObject.transform.position;
22         beginningRotation = this.gameObject.transform.rotation;
23         chase = false;
24         reset = false;
25         patrolForward = true;
26         navMeshAgent = GetComponent<NavMeshAgent>();
27         currentPatrolIndex = 0; etDestination();
28     }

```

Programski isječak 5.27: Prvi dio skripte "PatrolMovement" uključuje inicijalizaciju svih potrebnih varijabli

```

1     void Update()
2     {
3         FaceTarget();
4         //Ako je potrebno, resetiraj poziciju duha na pocetnu
5         if (reset == true)
6         {
7             reset = false;
8             chase = false;
9             this.gameObject.transform.position = beginningPosition;
10            this.gameObject.transform.rotation = beginningRotation;
11            currentPatrolIndex = 0;
12            SetDestiation();
13        }
14        //Ako treba loviti igraca, trenutna putna tocka je ona koja se nalazi na
15        igracu
16        if (chase == true)
17        {
18            Vector3 targetVector = playerPoint.transform.position;
19            currentWaypoint = playerPoint;
20            navMeshAgent.SetDestination(targetVector);
21        }
22        //Ako ne treba loviti igraca, kod prispjeca na trenutnu putnu tocku, usmjeri
23        se prema iducoj
24        else if (chase == false)
25        {
26            if (currentWaypoint != playerPoint)
27            {
28                if (traveling && navMeshAgent.remainingDistance <= 1.0f)
29                {
30                    traveling = false;
31                    ChangePatrolPoint();
32                    SetDestiation();
33                }
34            }
35            else if (currentWaypoint == playerPoint)
36            {
37                traveling = false;
38                ChangePatrolPoint();
39                SetDestiation();
40            }
41        }

```

Programski isječak 5.28: Ostvarivanje osnovne funkcionalnosti skripte "PatrolMovement"

```

1 //Promjeni putnu tocku sa vjerojatnoscu "changeDirectionProbability"
2 private void ChangePatrolPoint()
3 {
4     if (UnityEngine.Random.Range(0f, 1f) <= changeDirectionProbabilitiy)
5     {
6         patrolForward = !patrolForward;
7     }
8     if (patrolForward)
9     {
10        currentPatrolIndex = (currentPatrolIndex + 1) % patrolPoints.Count;
11    }
12    else
13    {
14        if (--currentPatrolIndex < 0)
15        {
16            currentPatrolIndex = patrolPoints.Count - 1;
17        }
18    }
19 }
20 //Zadaj ciljnu tocku prema kojoj se kreće
21 private void SetDestination()
22 {
23     if (patrolPoints != null)
24     {
25         Vector3 targetVector = patrolPoints[currentPatrolIndex].transform.position;
26         currentWaypoint = patrolPoints[currentPatrolIndex];
27         navMeshAgent.SetDestination(targetVector);
28         traveling = true;
29     }
30 }
31 //Duh mora biti licem okrenut prema igraču
32 void FaceTarget()
33 {
34     var turnTowardNavSteeringTarget = navMeshAgent.steeringTarget;
35     Vector3 direction = (turnTowardNavSteeringTarget - transform.position) .
36     normalized;
37     Quaternion lookRotation = Quaternion.LookRotation(new Vector3(direction.x, 0,
38     direction.z));
39     transform.rotation = Quaternion.Slerp(transform.rotation, lookRotation, Time.
deltaTime * 5);
}

```

Programski isječak 5.29: Dodatne metode skripte "PatrolMovement"

Duhovi će sada slobodno patrolirati svojim putnim točkama sve dok u njihovom vidnom polju nema igrača. Ako se igrač nalazi u njegovom dometu, duh zanemaruje dosadašnje putne točke i prati samo onu koja se nalazi na igračevom tijelu. Kako bi ostvarili radijus unutar kojeg duh vidi igrača, implementirana je jednostavna skripta "GhostVision".

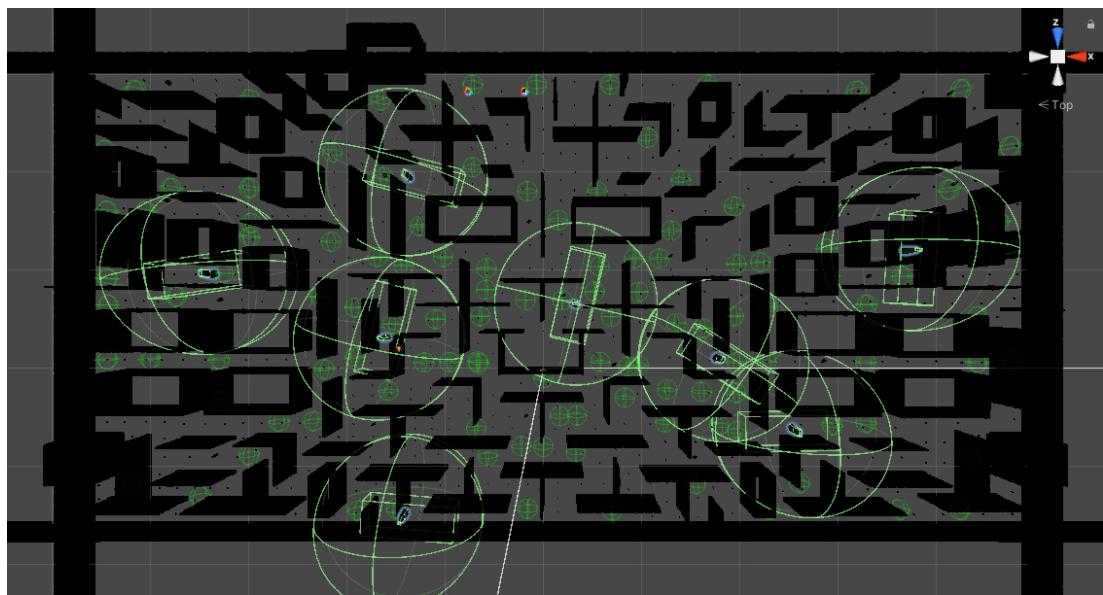
```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  public class GhostVision : MonoBehaviour
5  {
6      void OnTriggerStay(Collider col)
7      {
8          //Ako ono sto duh "vidi" ima oznaku "Player", zapocni loviti
9          if (col.gameObject.tag == "Player")
10         {
11             this.gameObject.GetComponent<PatrolMovement>().chase = true;
12         }
13     }
14     void OnTriggerExit(Collider col)
15     {
16         //Ako igrac vec 2 sekunde nije u vidnom polju duha, prestani loviti
17         if (col.gameObject.tag == "Player")
18         {
19             Invoke("StopChasing", 2f);
20         }
21     }
22     void StopChasing()
23     {
24         this.gameObject.GetComponent<PatrolMovement>().chase = false;
25     }
26 }
27
28 }
```

Programski isječak 5.30: Detekcija igrača unutar vidnog polja duha u skripti "GhostVision"

Konačan rezultat je funkcionalno i inteligentno ponašanje duhova prema željenim ishodima implementacije. Slika (5.31) prikazuje susret igrača i duha unutar labirinta odnosno trenutak u kojem duh lovi igrača, a na slici (5.32) su vidljivi položaji duhova u trenutku pretraživanja labirinta.



Slika 5.31: Prikaz trenutka u kojem se duh nalazi neposredno ispred igrača



Slika 5.32: Tlocrt scene u trenutku patroliranja pri čemu je radius duhova određen sfernim Colliderima

5.6. Elementi za sakupljanje

Iz paketa "Shotgun Set" osim modela oružja koristi se i model municije koju igrač može sakupljati. Predložak zlatnog novčića sadržan je unutar paketa "Gold Coins".

Kovanice su postavljene duž svih prohodnih puteva labirinta. Prilikom kolizije tijela igrača i novčića svira se određeni zvučni efekt, a količina dosad sakupljenog novca povećava se za jedan. U slučaju da su svi novčići sakupljeni, igra je uspješno završena i daljnje kretanje igrača je zaustavljeno.

```
1 void Update()
2 {
3     //Pracenje kolicine novca
4     if (string.Equals(coinCount.text, "617/617"))
5     {
6         //Zamrzni igraca
7         Rigidbody playerRb = this.gameObject.GetComponent< Rigidbody >();
8         playerRb.constraints = RigidbodyConstraints.FreezeAll;
9         //Ugasí skripte i komponente za kretanje
10        this.gameObject.GetComponent< FirstPersonController >().enabled = false;
11        this.gameObject.GetComponent< CharacterController >().enabled = false;
12        //Unisti objekte HUD, duh sa kojim je nastao sudar te oruzje
13        GameObject.Find("Ghosts").gameObject.SetActive(false);
14        GameObject.Find("HudCanvas").gameObject.SetActive(false);
15        GameObject.Find("WeaponPivot").gameObject.SetActive(false);
16        //Zatvori scenu nakon 1.5 sekundi
17        Invoke("CloseScreenOnSuccess", 1.5f);
18    }
19 }
20 }
```

Programski isječak 5.33: Isječak iz skripte "PlayerController" koji ostvaruje pribravanje sakupljenih novčića i praćenje stanja igre

Kreirano je nekoliko objekata, na različitim lokacijama unutar labirinta, koji će služiti kao pozicija za cikličko stvaranje predloška municije. Nakon što igrač uspješno sakupi jedan od tih predložaka, nakon određenog vremena, na istom mjestu stvorit će se novi. Stvaranje predložaka municije opisano je isječkom skripte "AmmoSpawner" (Isječak 5.34).

```

1   void Update()
2   {
3       //Ako objekt trenutno nema djece
4       if (this.gameObject.transform.childCount == 0)
5       {
6           //Ako je dozvoljeno stvaranje municije
7           if (spawnSwitch == false)
8           {
9               //Stvori predlozak sa kasnjnjem od 20 sekundi
10              Invoke("AmmoInstantiation", 20f);
11              spawnSwitch = true;
12              Invoke("ChangeSwitch", 20.5f);
13           }
14       }
15   }
16 }
17 void AmmoInstantiation()
18 {
19     GameObject newAmmo = Instantiate(ammoPrefab, this.gameObject.transform.
position, this.gameObject.transform.rotation, this.gameObject.transform);
20 }
21 void ChangeSwitch()
22 {
23     spawnSwitch = false;
24 }
25

```

Programski isječak 5.34: Skripta "Ammo Spawner" pridjeljena je svim objektima za stvaranje predložaka municije

Rezultat implementacije elemenata za sakupljanje prikazan je na slici (5.35). Broj sakupljenih elemenata svake vrste pamti se unutar skripte, a vizualno se ažurira unutar **Heads-up zaslona**.



Slika 5.35: Modeli municije i novčića unutar igre

5.7. Heads-up zaslon

Heads-up zaslon poznatiji kao **HUD** (engl. *Heads-up display*) odnosi se na prikaz podataka važnih igraču, bez potrebe da igrač skreće pogled sa svojih uobičajenih točki gledišta. U kontekstu pokazne igre, HUD označava informacije o količini sakupljenih novčića i trenutnom broju metaka koje igrač ima na raspolaganju u svakom trenutku. Kako bi se željene informacije ispisivale na zaslonu, potrebno je stvoriti objekt tipa *Canvas* čija će djeca biti tekstualni i dvodimenzionalni grafički elementi. Takvima manipulira se unutar skripte (Isječci 5.36), a promjene se u realnom vremenu prikazuju na zaslonu scene. Elementi HUD-a vidljivi su u gornjem i donjem lijevom uglu scene na slici (5.34).

```

1 //Ako je ispaljen metak, iskljuci idetu ikonu metka
2 void UpdateShellImages()
3 {
4     for (int i = 2; i > -1; i--)
5     {
6         if (shellImages[i].activeInHierarchy)
7         {
8             shellImages[i].SetActive(false);
9             break;
10        }
11    }
12 }
13

1 //Dohvacanje teksta komponente, parsiranje u tip integer i azuriranje teksta
2 void UpdateCoinCounter()
3 {
4     int parseNum;
5     string[] splitCounter = coinCount.text.Split('/');
6     string first = splitCounter[0];
7     bool parsing = int.TryParse(first, out parseNum);
8     if (parsing)
9     {
10        parseNum = parseNum + 1;
11        string result = parseNum.ToString() + "/" + "617";
12        coinCount.text = result;
13    }
14 }
15

```

Programski isječak 5.36: Isječci skripte "Shotgun" i "PlayerController" koji se odnose na ažuriranje podataka o trenutnom broju metaka i sakupljenih novčića

6. Rezultati

Kao rezultat ovog rada nastala je igra imena "617 Coins". Kroz idućih nekoliko slika prikazani su karakteristični trenutci zabilježeni za vrijeme igranja.

Slika 6.1 prikazuje početnu poziciju lika u gornjem lijevu kutu labirinta, a broj sakupljenih novčića na početku igre je 0.



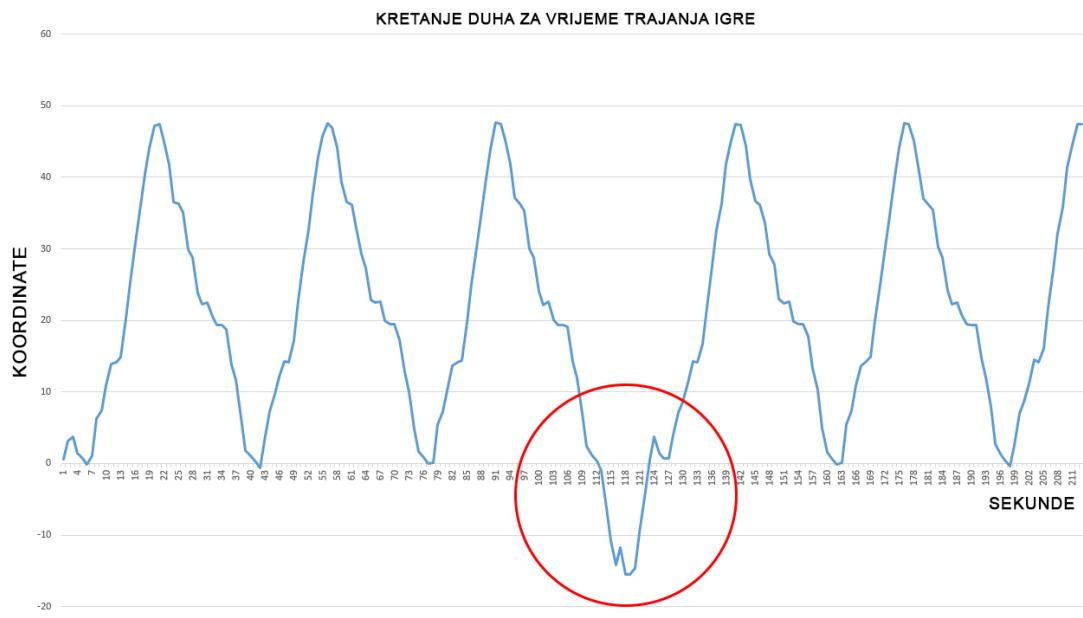
Slika 6.1: Početna pozicija igrača unutar labirinta

Na slikama 6.2 prikazan je slijed događaja prilikom neuspješnog igranja. Redom su prikazani: trenutak neposredno prije smrti igrača (uz vidljive efekte ispaljenog metka), ono što igrač vidi nakon što je ulovljen i onemogućeno mu je daljnje kretanje te ispis riječi "Death" prije ponovnog započinjanja igre. Na isti način, nakon uspješnog završetka igre i sakupljanja zadnjeg novčića, na ekranu se ispisuje "You survived".



Slike 6.2: Slijed događaja prilikom neuspjeha igrača

Kako bi pratili kretanje duha za vrijeme trajanja igre, svake dvije sekunde biježje se njegove trenutne koordinate u prostoru. Analizom podataka pomoću linijskog grafa (Slika 6.3), moguće je uočiti trenutak u kojem duh odstupa od svojeg uobičajenog uzorka kretanja (zaokruženo crvenom bojom). U tom trenutku igrač ulazi u vidno polja duha i duh ga počinje slijediti zanemarujući svoje unaprijed definirane putne točke. Nakon što je potjera završena, duh se vraća patroliranju te se ponovno iscrtava prethodni uzorak.



Slika 6.3: Linijski graf kretanja duha za vrijeme trajanja igre

7. Zaključak

Pretraživanje puteva predstavlja problem u brojnim domenama računarstva pa tako i u razvoju videoigara. Ponašanje inteligentnih agenata uvelike utječe na kvalitetu digitalnih igara pa je stoga važno dubinsko poznavanje problematike te ispravna implementacija legitimnih rješenja. U ovom radu, opisani su postupci izgradnje navigacijske mreže unutar razvojnog okruženja Unity kao i primjena ugrađenih komponenti za optimizaciju ponašanja likova vođenih algoritmom umjetne inteligencije. Rad je moguće tematski podijeliti u dva dijela pri čemu je u prvom dijelu predstavljena teorijska podloga za postizanje ciljnog rješenja. Opisane su komponente navigacijskog sustava Unityja i naveden je opći problem traženja puteva uz priložene interpretacije i česta algoritamska rješenja. Istaknuto je poglavlj o usmjerenom pretraživanju gdje se navodi algoritam A* visoke efikasnosti te njegove modifikacije. Na kraju prvog dijela, slijedi opis pokazne igre koja se razvija u svrhu implementacije rješenja spomenutih problema, a posebno se ističe odjeljak o neprijateljima te načinima na koji oni koriste umjetnu inteligenciju za ostvarivanje unaprijed definiranih ciljeva. U drugom dijelu, navode se konkretni implementacijski postupci s priloženim dijelovima koda i grafičkim prikazima scene unutar same igre.

Rezultati nam daju uvid u mogućnost korištenja Unityja, u kombinaciji s algoritmom A* i metodom RVO, za uspješnu implementaciju željenog ponašanja likova. Sustav za izgradnju navigacijske mreže u ovom je slučaju dao precizne rezultate, a metoda RVO je osigurala da međusobno izbjegavanje duhova ne utječe na konzistentnost obilaženja putnih točaka. Moguće je istražiti implementacijske metode koja su manje memorijski ili vremenski zahtjevne. Jedna od takvih zasigurno bi uključivala implicitno izračunavanje putnih točaka umjesto korištenog eksplicitnog definiranja, koje podrazumijeva ručno stvaranje i dodjeljivanje većeg broja objekata. Eksplicitna metoda predstavljala bi velik konceptualni problem na većim terenima.

LITERATURA

- [1] R. Graham, H. McCabe, and S. Sheridan, “Pathfinding in computer games,” *ITB Journal*, vol. 8, pp. 57–81, 2003.
- [2] X. Cui and H. Shi, “A*-based pathfinding in modern computer games,” *International Journal of Computer Science and Network Security*, vol. 11, no. 1, pp. 125–130, 2011.
- [3] L. Padgham and M. Winikoff, *Developing intelligent agent systems: A practical guide*. John Wiley & Sons, 2005.
- [4] “Navigation and pathfinding.” <https://docs.unity3d.com/Manual/Navigation.html>. 2018. Pristupljeno: 2022-05-09.
- [5] Z. He, M. Shi, and C. Li, “Research and application of path-finding algorithm based on Unity 3D,” in *2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*, pp. 1–4, IEEE, 2016.
- [6] B. Torun, S. Karakurt, T. B. Aydin, and Y. Altunel, “Game development on Unity,” *Turkish Online Journal of Educational Technology-TOJET*, vol. 20, no. 1, pp. 39–43, 2021.
- [7] D. Cherry, “RVO collision avoidance in Unity 3D,” *Distributed Research Experiences for Undergraduates in Computer Science and Engineering, DREU2013*, 2013.
- [8] J. Douthwaite, S. Zhao, and L. Mihaylova, “Velocity obstacle approaches for multi-agent collision avoidance,” *Unmanned Systems*, vol. 07, 01 2019.
- [9] R. E. Korf, “Artificial intelligence search algorithms,” 1999.
- [10] N. H. Barnouti, S. S. M. Al-Dabbagh, M. A. S. Naser, *et al.*, “Pathfinding in strategy games and maze solving using A* search algorithm,” *Journal of Computer and Communications*, vol. 4, no. 11, p. 15, 2016.

- [11] “Breadth-First Search - A BFS Graph Traversal Guide with 3 Leetcode Examples.” <https://www.freecodecamp.org/news/breadth-first-search-a-bfs-graph-traversal-guide-with-3-leetcodeexamples/> 2020. Pриступлено: 2022-05-15.
- [12] “What is the difference between BFS and DFS algorithms.” <https://www.freelancinggig.com/blog/2019/02/06/what-is-the-difference-between-bfs-and-dfs-algorithms/>. 2019. Pриступлено: 2022-05-15.
- [13] F. Duchoň, A. Babinec, M. Kajan, P. Beňo, M. Florek, T. Fico, and L. Jurišica, “Path planning with modified A* algorithm for a mobile robot,” *Procedia Engineering*, vol. 96, pp. 59–69, 2014.
- [14] A. Felner, U. Zahavi, R. Holte, J. Schaeffer, N. Sturtevant, and Z. Zhang, “Inconsistent heuristics in theory and practice,” *Artificial Intelligence*, vol. 175, no. 9-10, pp. 1570–1603, 2011.
- [15] “The history of Pac-Man.” <http://www.todayifoundout.com/index.php/2013/08/the-history-of-pac-man/>. 2013. Pриступлено: 2022-05-22.
- [16] S. Hantke, *Horror film: Creating and marketing fear*. Univ. Press of Mississippi, 2004.
- [17] G. A. Voorhees, J. Call, and K. Whitlock, *Guns, grenades, and grunts: First-person shooter games*. Bloomsbury Publishing USA, 2012.

PRIMJENA ALGORITAMA TRAŽENJA PUTA U PONAŠANJU RAČUNALNO-GENERIRANIH LIKOVA U VIDEOIGRI

Sažetak

Prilikom razvoja digitalnih igara, prisutan je izazov efikasne implementacije ponašanja likova vođenih algoritmom umjetne inteligencije. S obzirom na odabir razvojnog okruženja i algoritama, moguće je na različite načine ostvariti željena ponašanja. Ovaj rad daje pregled komponenti navigacijskog sustava razvojnog okruženja Unity te opis općenitog problema pretraživanja prostora uz poznatija rješenja. Poseban naglasak stavlja se na algoritam A*, a u svrhu demonstracije jednog od implementacijskih rješenja, razvijena je pokazna ambijentalna igra u prvom licu. Ishodi implementacije prvo su temeljito specificirani, a priloženi su i dijelovi koda uz grafičke prikaze scene tijekom igre.

Ključne riječi: Pogonski sustav za igre Unity, pronalazak puta, umjetna inteligencija, algoritam A*

APPLICATION OF PATHFINDING ALGORITHMS IN NON-PLAYER CHARACTERS BEHAVIOR IN A VIDEO GAME

Abstract

In video game development, there is a challenge to effectively implement the behavior of characters guided by artificial intelligence. Given the choice of various game engines and algorithms, it is possible to achieve the desired behaviors. This paper provides an overview of the components of Unity's navigation system and a description of the general pathfinding problem with well-known solutions. Special emphasis is placed on the A* algorithm and in order to demonstrate one of the implementation solutions, a demonstrative ambiental FPS game has been developed. The outcomes of the implementation were first thoroughly specified and pieces of code, as well as screenshots of the scene, were attached in this paper.

Keywords: Unity game engine, pathfinding, artificial intelligence, A* algorithm