# Tehnike oblikovanja umjetne inteligencije u videoigrama

**Arbanas, Karlo**

SVEUČILIŠTE U DUBROVNIKU
ODJEL ZA ELEKTROTEHNIKU I RAČUNARSTVO

KARLO ARBANAS

# TEHNIKE OBLIKOVANJA UMJETNE INTELIGENCIJE U VIDEOIGRAMA

DIPLOMSKI RAD

Dubrovnik, rujan 2021.

SVEUČILIŠTE U DUBROVNIKU
ODJEL ZA ELEKTROTEHNIKU I RAČUNARSTVO

# TEHNIKE OBLIKOVANJA UMJETNE INTELIGENCIJE U VIDEOIGRAMA

DIPLOMSKI RAD

Studij:Diplomski studij Primijenjeno/poslovno računarstvo

Mentor: prof.dr.sc. Vedran Batoš

Komentor: Ivan Grbavac, dipl.ing.

Student: Karlo Arbanas

Dubrovnik, rujan 2021.

# SADRŽAJ

# SAŽETAK

U ovom radu objašnjene su tehnike oblikovanja umjetne inteligencije uz pomoć Unreal Enginea. Unreal Engine je programski alat za razvoj video igara, simulacija i arhitektskih projekata u kojem je moguće oblikovatifunkcionalnosti koristeći C++ jezik i *Blueprints* - sustav za vizualno skriptiranje. Koristeći stabla ponašanja ili eksperimentalni sustav za ispitivanje okoliša (eng. *Enviroment Query System* - EQS) moguće je proizvesti računalno upravljane likove (eng. Non-Playable Character - NPC). Ti NPC-jevi mogu služiti kao prepreke ili pomagala igraču. U ovom projektu izrađena su 4 NPC-a pomoću stabla ponašanja, dok je peti izrađen pomoću EQS-a. Prvi NPC naziva se *Wanderer*. Taj NPC traži nasumičnu lokaciju na mapi i pomiče se prema toj lokaciji. Sljedeći NPC je nazvan *Guard*. *Guard* prati određenu putanju, a kada vidi igrača, lovi ga i pokuša ga ozlijediti. Igraču je dodana mogučnost skrećanja pažnje NPC-u kako bi se igrač lakše mogao sakriti. *Healer* je treći razvijeni NPC koji ima funkciju praćenja igračeve energije i liječenja igrača ako je ozlijeđen. NPC također prati svoju spremljenu energiju te traži najbližu stanicu za punjenje ako je prazan. Zadnji NPC, izrađen pomoću stabla ponašanja, je *Stalker*. Taj NPC očitava da li ga igrač može vidjeti te se kreće prema njemu ako se ne nalazi unutar igračevog vida, a inače je zamrznut. NPC razvijen pomoću EQS-a ispituje, pomoću testova lokacije, okruženje oko sebe tražeći najbolju lokaciju za skrivanje od igrače te se kreće prema toj lokaciji.

Ključne riječi: Unreal Engine, umjetna inteligencija, Blueprint, NPC, stablo ponašanja, EQS

# SUMMARY

This graduate thesis describes techniques of designing artificial intelligence by using Unreal Engine. Unreal Engine is a software used in the development of video games, simulations and arhitectual projects. Functions within the engine can be develped using C++ language or the Blueprint visual scripting system. Using the Behavior Trees or the experimental EQS it is possible to develop Non-Playable Characters - NPC. NPC can be used as obstacles or assistance to the player. 4 NPC were developed in this project using the Behavior Tree, while the fifth was developed using EQS. The first NPC is called Wanderer. This NPC finds a random location on the map and then moves to that location. The next NPC is called Guard. Guard follows a pre-determined path, but when he sees the player, he runs after and tries to hurt them. The player is given an ability to distract the NPC so that the player can hide more easily. Healer is the third NPC developed with the ability to monitor player's energy and healing the player if they are hurt. NPC also monitors it's own energy and looks for the nearest recharge station when it's energy is depleted. The last NPC developed using the Behavior Tree is called Stalker. This NPC detects if the player can see it and follows the player when it's outside the player's line of sight, otherwise it is frozen. NPC developed using the EQS tests it's surroundings and looks for a place best suited for hiding from the player. When it finds the location, it moves towards it.



Keywords: Unreal Engine, artificial intelligence, Blueprint, NPC, behavior tree, EQS

# 1. UVOD

Umjetna inteligencija u video igrama se primarno koristi za stvaranje računalno upravljanih likova koji imaju sposobnost reagiranja i prilagođavanja na događaje u video igri (često na događaje koje igrač sam proizvede). Umjetna inteligencija, koju navedeni likovi koriste,najčešće se izrađuje pomoću stabla ponašanja koje sadrži provjere određenih stanja te ovisno o tim stanjima odlučuje koji će se zadatak ili niz zadataka obaviti. Cilj umjetne inteligencije jest da simulira ljudsko ponašanje i donosi odluke s minimalnom ljudskom podrškom.

U ovom radu istražuje se stvaranje umjetne inteligencije unutar Unreal Enginea pomoću stabla ponašanja. U prvom poglavlju se opisuju povijest i svojstva Unreal Enginea, te sustav za vizualno skriptiranje zvan *Blueprint***.** U sljedećim poglavljima se opisuje stvaranje (koristeći *Blueprint* sustav i C++) 4 računalno upravljana lika koji koriste stablo ponašanja kako bi obavili svoje funkcionalnosti. Na temelju njih se opisuje proces stvaranja stabla ponašanja i prikazujudosezi koje je moguće postići sa stablima ponašanja. Na kraju rada se prikazuje izrada računalno upravljanog lika koristeći eksperimentalni EQS sustav.

# 2. UNREAL ENGINE

## 2.1. Povijest

Unreal Engine je softver za razvoj video igara kojeg je razvio Tim Sweeney, osnivač Epic Games kompanije. Rad na Unreal Engineu je započeo 1995. godine, a prva video igra koja je bila razvijena na njemu je Unreal, pucačina u prvom licu. (1)

Prva verzija Unreal Enginea sadržavala je sustav za otkrivanje kolizije (što je omogućuje simulaciju fizike), mogućnost iscrtavanja obojenog svijetla i ograničeno filtriranje tekstura. Unreal Engine također je sadržavao integrirani uređivač mape zvan UnrealEd, koji je omogućavao programerima jednostavno pravljenje novih i izmjenu već postojećih mapa u video igrama (koje su pravljenje u Unreal Engineu), te UnrealScript koji je programerima omogućavao jednostavnu izmjenu mnogih dijelova video igre (npr. kako će se igrač kretati, s kojim elementima video igre će moći manipulirati i sl.) bez duboke izmjene samog Unreal Enginea. Zbog toga je Unreal Engine postao veoma popularan, te su ga mnogi programeri i kompanije počeli koristiti za razvoj vlastitih projekata. (1)

2002. godine objavljena je druga verzija Unreal Enginea (zvana Unreal Engine 2). U usporedbi sa svojim prethodnikom, Unreal Engine 2 donosi poboljšanja u iscrtavanju objekata na ekranu: implementiran je *Particle System* (sustav pomoću kojeg se mogu simulirati specijalni efekti kao npr. vatra, dim, iskre električne struje i sl.), poboljšanja u simulaciji fizike te mogućnost stvaranja animacija pomoću kostura. Osim tih poboljšanja također su ažurirani i alati Unreal Enginea (UnrealEd je prepisan u C++ kodu pomoću wxWidgetsa). (1)

Nakon 18 mjeseci razvijanja, treća verzija Unreal Enginea (Unreal Engine 3) jepredstavljena u srpnju 2004. godine. Podržavala je mnoge napredne tehnike kao što su HDRR, dinamične sjene, računanje svih osvjetljenja i sjena se radilo po pikselu (za razliku od prijašnjih verzija Unreal Enginea koja su računanje osvjetljenja i sjena obavljali po verteksu), te znantno poboljšanje u iscrtavanju, fizici, zvuku i alatima Unreal Enginea. Osim za izradu video igara, Unreal Engine 3 se koristio za izradu projekata u konstrukciji, dizajnu, simulaciji vožnje, simulaciji za treniranje, filmovima i sl. Iako je Unreal Engine 3 bio otvoren svim programerima za uporabu, izdavanje i prodaja projekata je bilo ograničeno licencom Unreal Enginea. U studenom 2009. godine Epic Games je izdao besplatnu verziju SDK-a (eng. *Software Development Kit*) od Unreal Engine 3 zvanu UDK (eng. *Unreal Development Kit*) koja je bila dostupna javnosti. (2)

Unreal Engine 4 je trenutno aktivna verzija. Od 2003. godine Tim Sweeney je sam radio na četvrtoj verziji Unreal Enginea, a sredinom 2008. godine pridružuje mu se ostatak tima koji je bio odgovoran za Unreal Engine 3. Uz poboljšano iscrtavanje sjena, efekata, te poboljšanje sustava za fiziku, jedna od najvažnijih promjena od prijašnjih verzija jest zamjena UnrealScript-a sa "*Blueprint*" sustavom za vizualno skriptiranje, koji omogućuje dizajnerima jednostavan

pristup gotovo svim alatima koji su inače na raspolaganju samo programerima. *Blueprint* sustav omogućuje laku izmjenu elemenata projekata koji se brzo primjenjuju na projekt te se odmah mogu testirati. (1)

### 2.2. *Blueprint* sustav za vizualno skriptiranje

*Blueprint* visualni sustav za skriptiranje je sustav za skriptiranje baziran na konceptu *nodeova* pomoću kojeg se skriptiraju elementi projekta. Njime se definiraju objektno orijentirane klase i objekti u Unreal Engineu. *Nodeovi* su zadatci, događaji, pozivi funkcija, varijable i sl. kojima se definira funkcionalnost *Blueprinta*. Pomoću *Blueprint* sustava dizajneri imaju pristup gotovo svim alatima inače dostupni samo programerima, dok se programerima omogućuje stvaranje osnovnih funkcija koje dizajneri mogu proširiti. (3)

Iako je moguće sve funkcionalnosti izraditi u C++ kodu, u nekim je slučajevima, kao na primjer stvaranje vrlo jednostavnih funkcija, jednostavnije funkciju stvoriti unutar *Blueprinta*.

Svaki *node* sadrži igle (eng. *pin)* koji se mogu nalaziti na lijevoj (*input*) ili na desnoj (*output*) strani *nodea*. *Pinovi* služe za unošenje podataka u *node* (ako se nalaze na *input* strani) ili za uzimanje podataka (*output* strana) iz *nodea*. Boja *pina* ovisi o tipu podatka koje prima/nosi, npr. string *pinovi* su ružičaste boje, float zelene i sl. Navedeni *pinovi* su data *pinovi*. Također postoji *Execution pin* (bijeli trokut na vrhu *nodea*) koji služi za označavanje redosljeda obavljanja *nodeova*.

Postoje različite klase*Blueprintsa* pomoću kojih je moguće stvoriti nove elemente unutar Unreal Engine projekta, mijenjati karakteristike tih elemenata i stvoriti*macrose* (funkcije) koji se mogu koristiti u ostalim *Blueprintsima*:

***Blueprint* klasa** omogućuje dodavanje funkcija nad već postojećim klasama. Stvaraju se vizualno unutar Unreal Enginea bez kodiranja i definiraju nove klase ili vrste *Actora* koje se zatim mogu postaviti na mapi kao instance.

***Data-Only Blueprint*** je *Blueprint* klasa koja sadrži samo kod u obliku grafa *nodeova*, varijabli i komponenata naslijeđenih od roditelja. Ti se elementi mogu mijenjati, ali dodavanje novih elemenata nije moguće.

***Level Blueprint*** djeluje kao globalni graf događaja na razini cijele mape u projektu. Svaka mapa u projektu sadrži vlastiti *Level Blueprint*, a novi *Level Blueprinti* se ne mogu stvoriti bez stvaranja nove mape. Unutar *Level Blueprinta* se mogu definirati događaji nad određenim instancama *Actora* unutar mape ili nad cijelom mapom.

***Blueprint Macro Library*** sadrži kolekciju *Macrosa* koje je moguće postaviti u ostale *Blueprinte* kao *nodeove*. *Macros* su funkcionalnosti koje su korisnici *Unreal Enginea* sami napravili te su sklopljene u jedan *node*. To omogućuje implementaciju često korištenih funkcija bez njihovog ponovnog pisanja.

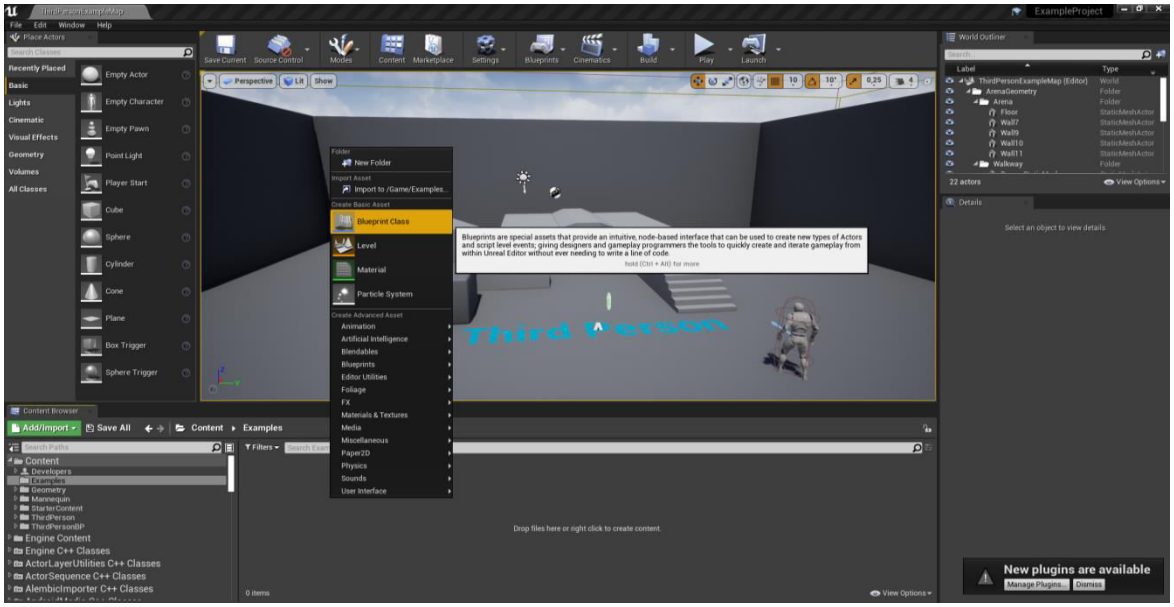
***Blueprint Interface*** je kolekcija funkcija bez implementacije koje se dodaju u druge *Blueprintse*. Svaki *Blueprint* koji sadrži *Blueprint Interface* će također sadržavati funkcije unutar *Interfacea*. Tim se funkcijama u svakom *Blueprintu* može dati funkcionalnost. *Blueprint Intefacei* imaju određene limite, tj. ne mogu dodavati nove varijable i komponente.

### 2.3. Primjer *Blueprint* sustava

U ovom poglavlju prikazat će se primjeri izrade nekih funkcionalnosti pomoću *Blueprint* sustava za vizualno skriptiranje. Koristi se Unreal Engine 4.26.2 verzija. Za izradu primjera koristi se novi projekt sa "*third-person*" predloškom i uvedenim početnim sadržajem. U projektu se automatski stvara početna mapa sa igračem kojeg je moguće kontrolirati.

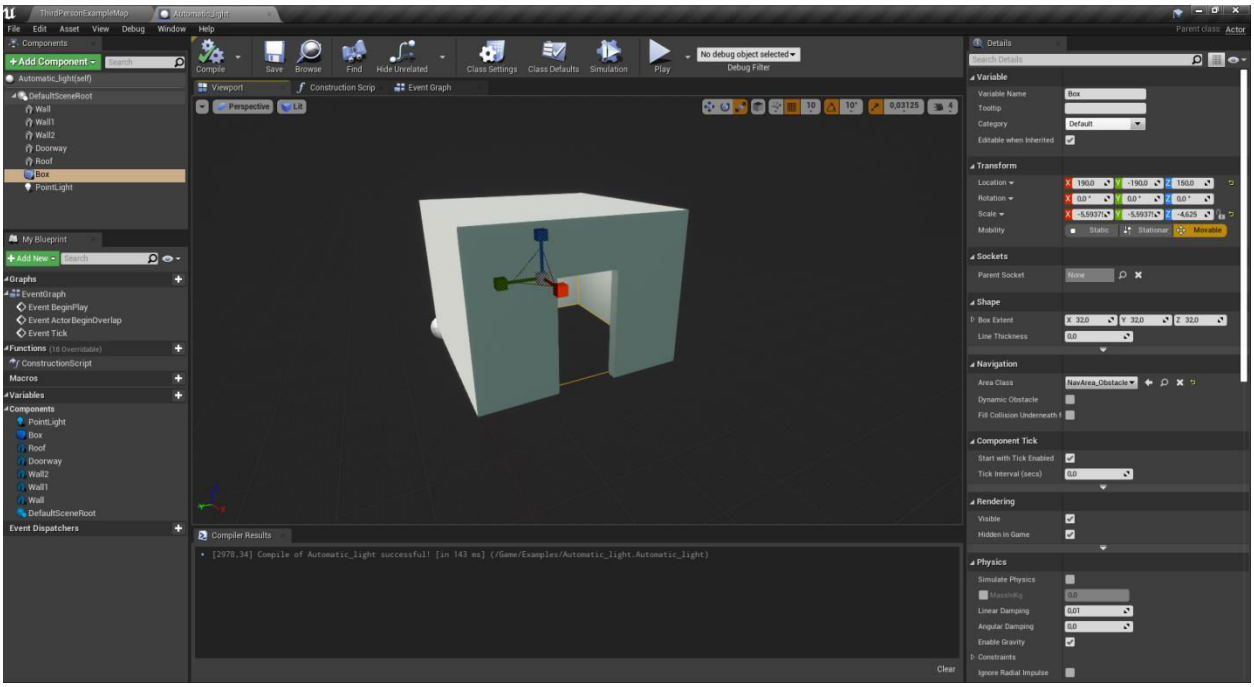
Cilj poglavlja je upoznati čitaća sa *Blueprint* sustavom za vizualno skriptiranje i sa postupkom dodavanja komponenti i funkcionalnosti *Actorima* kako bi čitać u sljedećim poglavljima lakše mogao razumjeti postupke stvaranja potrebnih *Blueprint* klasa.

Prvi primjer je izrada kuće sa svjetlom koje se upali kada igrač uđe u kuću, a izgasi nakon što igrač izađe iz kuće. Stvaranje novog *Blueprinta* se postiže desnim klikom na Content Browser i odabirom "*Blueprint Class*" kao što je prikazano na slici 1. Zatim je potrebno odabrati vrstu *Blueprint* klase. Za ovaj primjer koristi se *Actor* klasa.




**Slika 1.** Stvaranje nove *Blueprint* klase

Novi prozor koji je otvoren koristi se za dodavanje komponenti u *Blueprint* (u ovom primjeru te komponente će biti zidovi kuće, svjetlo i nevidljiva kutija za koliziju koja će se koristiti za otkrivanje kada je igrač ušao u kuću i kada će se svjetlo upaliti/ugasiti) i izrade funkcionalnosti (programiranje svjetla da se upali/ugasi kada igrač uđe/izađe iz kuće). Dodavanje novih komponenti se postiže pomoću "*Add Components*" gumba u gornjem desnom kutu prozora i odabirom vrste komponente. Za zidove se koristi *Static Mesh* komponenta, za svjetlo *Point Light*, a za nevidljivu kutiju za koliziju se koristi *Box Collision* komponenta. Sve komponente je potrebno pomaknuti, rotirati i skalirati kako bi se napravila kuća. Kutija za koliziju treba ispuniti unutrašnjost kuće kako bi se pravilno otkrilo da li je igrač ušao ili izašao iz kuće. Slika 2. prikazuje konačni izgled kuće.



**Slika 2.** Izgled kuće sa automatskim svjetlom

Izrada funkcionalnosti se obavlja u *Event Graph* prozoru koji se nalazi poviše glavnog prozora u kojemu su prikazane sve komponente, u ovom slučaju gdje je prikazana kuća. Programiranje funkcionalnosti se odvija pomoću *nodeova* koji se stvaraju desnim klikom miša na glavni prozor*Event Grapha* te odabirom željenog *nodea* u ponuđenoj listi. Ako je ijedna komponenta *Blueprinta* označena prilikom stvaranja *nodea*, tada će na početku liste biti ponuđeni svi *nodeovi* (tj. sve operacije) koje se mogu primijeniti na komponentu ili koja koriste taj komponent.

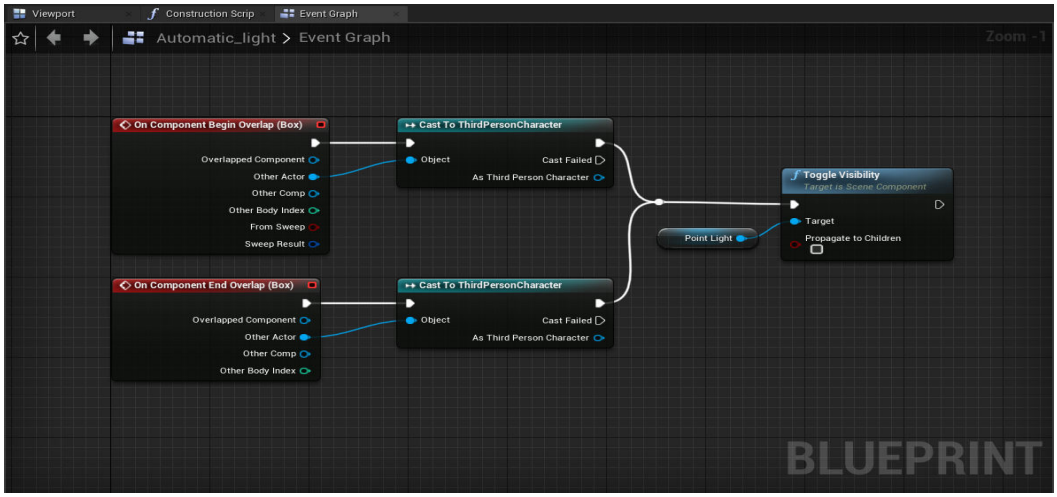
Za ovaj primjer potrebno je kliknuti na *Box* komponentu te u *Event Graphu* stvoriti *node* "*On Component Begin Overlap*". Time će se stvoriti event *node* koji se pokreće kada se otkrije da se *Actor* preklapa sa kutijom za otkrivanje kolizije. Taj event *node* će biti potreban za paljenje svijetla kada igrač uđe u kuću. Zatim je ispod "*On Component Begin Overlap*" *nodea* potrebno

stvoriti "*On Component End Overlap*" *node* pomoću iste metode. Taj *node* otkriva kada prestaje preklapanje sa kutijom za otkrivanje kolizije i kada će se svijetlo ugasiti.

Stvoreni *nodeovi* će se pokrenuti kada bilo koji *Actor* započne ili završi preklapanje sa kutijom, što znači da bilo koji *Actor* bi mogao paliti/gasiti svijetlo nakon završetka *Blueprinta*. Kako bi se osiguralo da samo igrač može djelovati na svjetlo, potrebno je provjeriti koji *Actor* preklapa kutiju. To se postiže povlačenjem od "*Execution Pina*" (bijeli trokut na vrhu) i odabirom "*Cast to ThirdPersonCharacter*". Taj *node* provjerava da li primljeni *Actor* pripada određenoj klasi (u ovom primjeru da li *Actor* koji preklapa kutiju za koliziju pripada klasi igrača, tj. *Third-person Character* klasi). "*Cast to*" *nodeovi* na input strani primaju objekt koji provjeravaju. Taj objekt, u ovom primjeru, će biti *Actor* koji preklapa kutiju za koliziju. Unos preklopljenog objekta obavlja se povlačenjem od *OtherActorpina* na output strani "*On Begin Overlap*" *nodea* do *Object pina* na input strani "*Cast to ThirdPersonCharacter*" *nodea*. Isti postupak potrebno je obaviti za "*On End Overlap*" *node* kako bi se provjerilo da li *Actor* koji izlazi iz kutije za koliziju pripada klasi igrača.

Zadnji korak jest programiranje paljenje/gasenje svjetla. Pri stvaranju komponenta za svjetlo (*point light*), svijetlo je već upaljeno. Kako bi osigurali da je svijetlo započinje ugašeno potrebno je označiti point light komponentu, te u *Details* prozoru na desnoj strani pronaći i odznačiti *visible* postavku (postavke je moguće pronaći pomoću pretraživaća na vrhu *Details* prozora). Zatim, dok je *point light* još označen, potrebno je stvoriti "*Toggle Visibility (Point Light)*" *node*, te oba *execution pina* od "*Cast to*" *nodeova* spojiti na ulazni execution pin od "*Toggle Visibility*" *nodea*. "*Toggle Visibility*" *node* pri pokretanju promjenjuje stanje vidljivosti svjetla, tj. ako je svjetlo bilo ugašeno, pri pokretanju *nodea* ga pali i obrnuto. Pri preklapanju igrača sa kutijom za koliziju prije ugašeno svjetlo će se upaliti, a pri prestanku preklapanja upaljeno svjetlo će se ugasiti.

Konačni graf *Blueprinta* prikazan je na slici 3.




**Slika 3.***Blueprint* graf za paljenje/gašenje svjetla

# 3. Izrada umjetne inteligencije pomoću stabla ponašanja

Izrada umjetne inteligencije u Unreal Engine 4 postiže se pomoću ***Behavior Tree*** (stablo ponašanja) modela. Stablo ponašanja je matematički model koji opisuje modularno prebacivanje između konačnog skupa zadataka. Koristi se u robotici, računalnoj znanosti, sistemima za kontrolu i video igrama. Pomoću stabala ponašanja obavljaju se provjere stanja i varijabli te se odlučuje koji će se zadatak obaviti. (4)

U Unreal Engineu NPC (eng. *Non-Playable Character* - računalno upravljan lik) koristi *Behavior Tree* kako bi donio odluke o zadatku koji će obaviti (npr. ako vidi igrača onda će trčati za njim, inače će hodati u nasumičnom smjeru). NPC je lik u video igri kojega igrač ne kontrolira, nego sadrži AI (eng. *Artificial Intelligence* - Umjetnu Inteligenciju) pomoću koje obavlja određene zadatke.

## 3.1. *Behavior Tree* u Unreal Engine 4

Za izradu umjetne inteligencije u Unreal Engine 4 potrebne su 3 komponente: *Behavior Tree, Blackboard i AI Controller*.

***Behavior Tree*** se koristi kako bi se obavljale provjere raznih stanja pomoću kojih se odlučuje koji zadatak će NPC obaviti. *Behavior Tree* izgleda kao obrnuto stablo sa *nodeovi*ma koji provjeravaju određena stanja. *Behavior Tree* se obavlja po prioritetu, od krajnje lijevog *nodea* prema desnom. Ako provjera u *nodeu* vrati FALSE (npr. NPC provjerava da li može vidjeti igrača i zaključuje da ga ne može vidjeti) onda se prebacuje na sljedeći *node* desno. Pri izradi *Behavior Treea* važno je paziti na prioritet *nodeova*.

***Blackboard*** sadrži ključeve, podatke koje *Behavior Tree* koristi kako bi obavljao provjere. Pri stvaranju novog ključa potrebno je definirati tip podatka koji će spremiti. U postavkama *Behavior Treea* potrebno je postaviti koji *Blackboard* će koristiti.

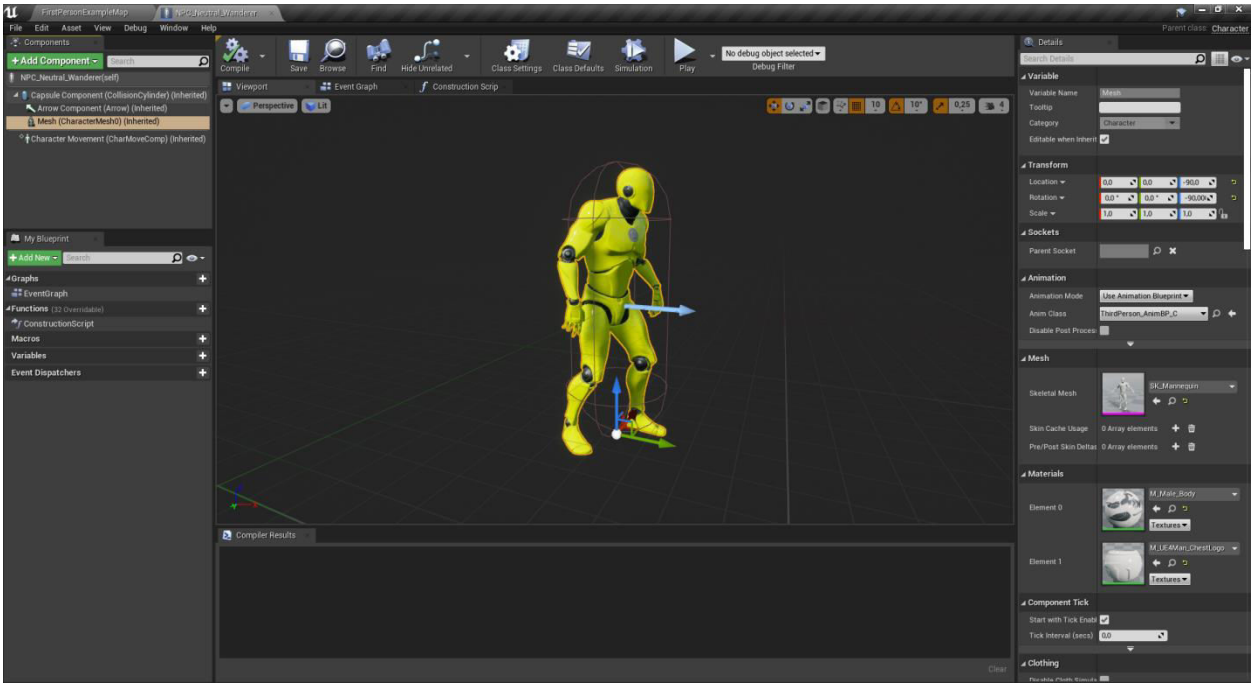
***AI Controller*** je *Blueprint* klasa koja ima sličnu ulogu kao igrač koji kontrolira igračeg *Actora*, tj. opaža svijet oko sebe i kontrolira *Actora*. Unutar *Blueprinta* od NPC-a potrebno je postaviti koji će *AI Controller* koristiti, a unutar *AI ControllerBlueprinta* potrebno je postaviti naredbu koji će *Behavior Tree* pokrenuti kada se pokrene projekt.

Sve 3 potrebne komponente stvaraju se istim postupkom kao kod stvaranja nove *Blueprint* klase. U sljedećim poglavljima opisuje se postupak stvaranja NPC-jeva ovog projekta. Prvi NPC je izrađen samo u *Blueprintu* dok su ostali u C++.

### 3.2. Izrada *Wanderer* NPC

*Wanderer* je jednostavan NPC koji traži nasumičnu točku na mapi i dolazi na tu točku. Za izradu *Actora* koji će predstavljati NPC koristi se *Character* klasa *Blueprinta*. *Character* klasa već sadrži komponente koje su potrebne za NPC kao što su: *CharacterMovement* (koristi se za pokretanje *Actora*) i *Capsule Component* (kapsula koja se koristi za koliziju). *Capsule Component* sadrži dodatne komponente: *ArrowComponent* (pokazuje smjer prema kojemu je *Actor* okrenut) i *Mesh* (koristi se za odabir 3D modela koji će prikazati NPC u svijetu).

Prvi korak jest odabir 3D modela za NPC. Potrebno je označiti *Mesh* komponentu u *Blueprintu* od NPC-a i u *Details* prozoru pod *SkeletalMesh* odabrati 3D model. U ovom projektu korišten je *SK_Mannequin* za koji je potrebno odabrati odgovarajuću animaciju što se postiže pod *Animation* i odabirom *ThirdPerson_AnimBP_C*. 3D model je potrebno pomaknuti i rotirati tako da se nalazi unutar *Capsule Componenta* i da je usmjeren prema *ArrowComponentu*. 3D model je prikazan na slici 4.



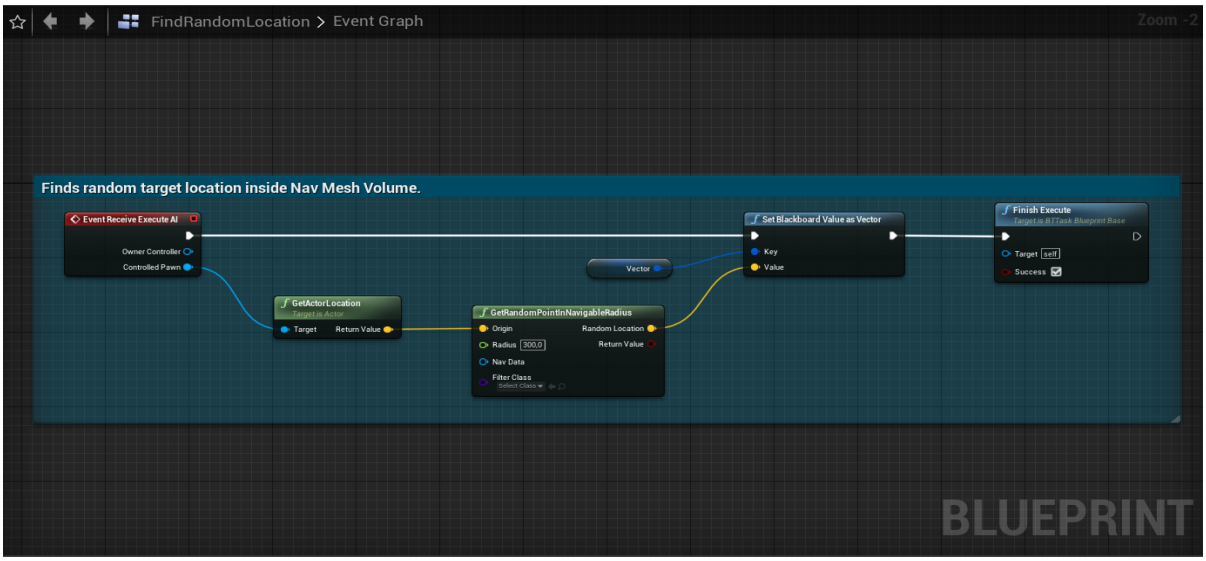
**Slika 4.** 3D model koji prikazuje Wanderer NPC-a u svijetu

Za izradu umjetne inteligencije NPC-a potrebno je stvoriti *AI Controller*, *Behavior Tree* i *Blackboard*. Unutar *Blueprinta* od NPC-a potrebno je postaviti da koristi *AI Controller*. To se postiže označujući *NPC_Wanderer (Self)* na vrhu prozora u kojem se dodaju komponente (gdje se nalazi *Mesh* i *Arrow Component*), te odabirući, unutar*Details* prozora pod *AI Controller Class*, stvoreni *AI Controller*. Sljedeće, unutar *Blueprinta* od *AI Controller*a potrebno je napraviti naredbu da se *Behavior Tree* pokrene pri pokretanju projekta. U *Event Graphu* se stvara *Event Begin Playnode* i od tog *nodea* se stvara *Run Behavior Treenode*. Pod *BTAsset* u

*Run Behavior Treenodeu* se postavlja koji se *Behavior Tree* pokreće, te je potrebno postaviti stvoren *Behavior Tree* za NPC-a.

Izrada umjetne inteligencije za NPC se obavlja unutar *Behavior Treea*. *Behavior Tree* započinje sa *Rootom* (korijenom) koji služi kao početna točka u izvođenju *Behavior Treea*. Od *Roota* se stvaraju ostali *nodeovi* pomoću kojih se obavljaju određene provjere ili zadatci koje će NPC izvršiti. Od tih *nodeova* se mogu dalje stvarati *nodeovi* ili zadatci. *Nodeovi* i zadatci se obavljaju po redosljedu od krajnje lijevog kraja *Behavior Treea* prema desno. Vrste čvrova koje su potrebne za *Wanderer* NPC su *Sequence* i *Selector*. *Sequencenode* obavlja sve *nodeove* ili zadatke koji su iz njega stvoreni po redu, dok *Selector* prekida izvođenje ako zadatak vrati uspjeh (tj. zadatak je uspješno obavljen), inače nastavlja sa sljedećim zadatkom sve dok ne dođe do uspješno obavljenog zadatka ili do kraja svog dijela stabla.

Za *Wanderer* NPC su potrebana 3 zadatka: pronalaženje nasumične lokacije na mapi, micanje NPC-a do te lokacije i čekanje određeno vrijeme prije ponovnog pokretanja *Behavior Treea*. Zadnja dva zadatka već postoje unutar Unreal Enginea dok je pronalaženje nove lokacije potrebno napraviti. Zadatci za *Behavior Tree* su *Blueprint* klase zvane *BTTask_BlueprintBase* i stvaraju se istim postupkom kao i *AIController*. Unutar *Event Grapha* od novostvorenog zadatka je potrebno stvoriti "*Event Recieve Execute AI*" *node* koji pokreće ostatak *Blueprinta* kada primi naredbu od *Behavior Treea*. Na izlazu tog *nodea* se nalazi Controlled Pawn izlaz koji služi kao referenca na *Actor* kojeg *AI Controller* kontrolira (u ovom slučaju je to referenca na *Wanderer* NPC) te je od tog izlaza potrebno stvoriti "*Get Actor Location*" *node* koji na svom izlazu vraća koordinate primljenog *Actora*. Od tog izlaza je potrebno stvoriti "*Get Random Point in Navigable Radius*" *node*. Taj *node* na ulazu prima početnu točku (u ovom slučaju to je lokacija NPC-a) i radjius unutar kojega će se tražiti nova lokacija. Radijus je u ovom projektu postavljen na 300,0 što osigurava da nova lokacija ne bude pre daleko od NPC-a. Nove koordinate je potrebno spremiti, a to se postiže stvaranjem "*Set Blackboard Value as Vector*" i *Value* ulaz je potrebno spojiti sa Random Location izlaz od "*Get Random Point in Navigable Radius*" *nodea*. Također je potreban desni klik na *Key* ulaz i odabrati *Promote to Variable*. Na kraju je potrebno stvoriti "*Finish Execute*" *node* i na *Success* ulazu postaviti kvačicu. Taj *node* vraća TRUE ili FALSE*Behavior Treeu* i prekida izvršavanje zadatka, a *Success* ulaz označava bool podatak koji se vraća *Behavior Treeu*. Krajnji izgled *Blueprinta* je prikazan na slici 5.

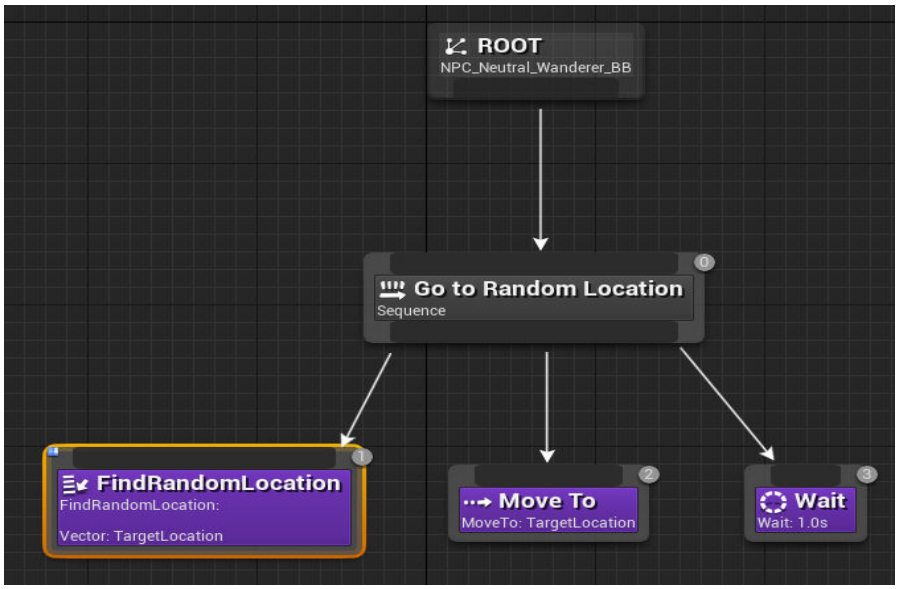



**Slika 5.** Blueprint graf za pronalaženje nove nasumične koordinate

Unutar *Blackboarda* je potrebno stvoriti varijablu koja će služiti kao spremište koordinate prema kojoj će se NPC micati. Stvaranje nove se postiže pomoću *New Key* opcije i odabirom vektor tipa varijable. U ovom projektu taj ključ je nazvan *TargetLocation*. Pri izvođenju *Behavior Treea,* unutar ključa će se spremiti koordinate nove lokacije i uzimati te koordinate kada je potrebno pomicati NPC.

Zadnji korak jest postavljanje *Behavior Treea* i *NavMesh* komponente. Iz *Rootnodea* u *Behavior Treeu* potrebno je stvoriti *Sequencenode*. To se obavlja povlačenjem iz donjeg dijela *Rootnodea* i odabirom *Sequence*. Iz *Sequencenodea* potrebno je napraviti set zadataka pomoću kojih će NPC pronaći novu lokaciju, doći do te lokacije i pričekati određeno vrijeme prije ponavljanja. Stvaranje zadatka za odabir lokacije obavlja se istim postupkom kao i stvaranje *Sequencenodea*, ali je potrebno u *Details* prozoru označiti *Blackboard* varijablu (*TargetLocation*) u kojoj će se spremiti koordinate. Zatim je potrebno istim postupkom stvoriti *Move To* zadatak i postaviti *TargetLocation* ključ, te napraviti *Wait* zadatak i postaviti u *Details* prozoru vrijeme (u projektu je postavljeno na 1 sekundu) i sačuvati promjene.

*Behavior Tree* od *Wanderer* NPC-a je prikazan na slici 6.




**Slika 6.** Behavior Tree koji pripada Wanderer NPC-u

Kada se NPC ubaci u mapu projekta i kada se projekt pokrene, on će prvo pronaći koordinate nasumične lokacije u radjiusu oko sebe, spremiti te koordinate u *Blackboard* varijablu, krenuti prema toj lokaciji i kada dođe do nje pričekati 1 sekundu i ponovno pokrenuti *Behavior Tree*.

### 3.2.1. Pregled *Wanderer* NPC-a

Ovaj NPC služi kao uvod za izradu umjetne inteligencije koristeći *Behavior Tree*. Izrada i korštenje *Behavior Tree* i *Blackboard* klase je ključna u izradi NPC-jeva koje će biti objašnjena u sljedećim poglavljima. Zadatci su rađeni pomoću *Blueprint* sustava za vizualno skriptiranje radi boljeg upoznavanja s mogućnostima tog sustava. U sljedećim poglavljima se prikazuje izrada zadataka koristeći C++, te se prikazuje više mogućnosti unutar Behavior Tree-a kao npr. provjera stanja *Blackboard* ključeva i odlučivanje koji će se zadatak izvršiti ovisno o tim ključevima.

### 3.3. *Guard* NPC

*Guard* je NPC koji je pisan u C++. On prati već određenu putanju, a kada vidi igrača onda trči za njim i pokuša ga udariti. Ako NPC izgubi igrača iz svog vida, ide na zadnju lokaciju gdje ga je vidio i istražuje kratko u nasumičnom smjeru nakon čega se vrati svojoj putanji. Igrač može pustiti zvuk na svojoj lokaciji koji će NPC istražiti na način da prekine svoje patroliranje i krene prema izvoru zvuka. Kada NPC dođe do izvora zvuka, pričekati će određeno vrijeme na toj lokaciji i vratiti se normalnom patroliranju (ako je u međuvremenu vidio igrača onda će ga loviti). Ovo se može samo obaviti ako NPC ne trči za igračem.

#### 3.3.1. *Blackboard*ključevi *Guard* NPC-a

Ovaj NPC koristiti sljedeće *Blackboard* ključeve pomoću kojih će pravilno obavljati sve zadatke i provjere:

- ***PatrolPathVector***- koordinate točke u putanji prema kojoj NPC treba ići. Pri traženju koordinata točke podatci se spremaju unutar ključa, a pri micanju NPC-a podaci se čitaju iz tog ključa.
- ***PatrolPathIndex*** - indeks trenutne točke putanje u listi. Koristi se radi pronalaženja koordinata točke u listi i radi provjere na da li je ta točka zadnja u listi.
- ***PathLooping*** - označava da li će NPC na kraju putanje ponovno krenuti od prve točke. Ako je postavljena na FALSE NPC će na kraju putanje ići unazad po točkama dok ne dođe do prve točke putanje.
- ***Direction*** - označava smjer kretanja NPC-a po putanji. Koristi se pri traženju sljedeće točke putanje i samo ako je *PathLooping*FALSE.
- ***CanSeePlayer***- bilježi da li NPC vidi igrača.
- ***TargetLocation*** - unutar ključa se spremaju koordinate igrača (ako NPC ga lovi), koordinate zadnje lokacije igrača (kada NPC izgubi igrača iz vida) i koordinate podrijetla zvuka koji NPC istražuje.
- ***PlayerIsInMeleeRange*** - označava da li je NPC dovoljno blizu igrača da ga može pokušati napasti.
- ***IsInvestigating*** - označava da li je NPC čuo zvuk te pokreće *Behavior Tree* granu za istraživanje.
- ***HasBeenChasing*** - da li je NPC lovio igrača. Provjerava se nakon što NPC izgubi igrača iz vida kako bi se pravilno pokrenula grana *Behavior Treea* zadužena za istraživanje zadnje lokacije igrača.

#### 3.3.2. Normalno patroliranje putanje

Pošto će se NPC raditi u C++, potrebno je stvoriti C++ klase od *Character*-a i *AI Controllera*. Unutar Content Browser prozora u C++ classes folderu je potrebno stvoriti novu C++ klasu i odabrati *Character* i *AI Controller* klasu, dok se *Behavior Tree* i *Blackboard* stvaraju istom

metodom kao i prije. Unutar *AI Controller* klase je potrebno inicijalizirati *Behavior Tree* i *Blackboard*, te postaviti sustav za percepciju kako bi NPC imao osjetila za vid i sluh. Sustav za percepciju radi tako da *Actor* koji sadrži taj sustav očitava stimulatore u okolini te ažurira prikladne dijelove svog sustava ovisno koji je stimulant očitao. Na drugi *Actor* je potrebno postaviti sustav za stimulaciju i tip stimulatora (vid, sluh ili sl.). Inicijalizacija unutar konstruktora klase je prikazana u primjeru 1.

```
AGuard_AI::AGuard_AI(FObjectInitializer const& object_initializer)
{
    // Initialize Behavior Tree
    static ConstructorHelpers::FObjectFinder<UBehaviorTree> obj(TEXT("BehaviorTree'/Game/NPC/Hostile/Guard/NPC_Hostile_Guard_BT.NPC_Hostile_Guard_BT'"));

    // Check cast
    if (obj.Succeeded())
    {
        btree = obj.Object;
    }

    // Access Behavior Tree and Blackboard
    behavior_tree_component = object_initializer.CreateDefaultSubobject<UBehaviorTreeComponent>(this, TEXT("BehaviorTreeComponent"));
    blackboard = object_initializer.CreateDefaultSubobject<UBlackboardComponent>(this, TEXT("BlackboardComponent"));

    // Setup Perception System
    SetupPerceptionSystem();
}

void AGuard_AI::BeginPlay()
{
    Super::BeginPlay();

    // Runs behavior tree (needs both lines)
    RunBehaviorTree(btree);
    behavior_tree_component->StartTree(*btree);
}

void AGuard_AI::OnPossess(APawn* const pawn)
{
    Super::OnPossess(pawn);

    if (blackboard)
    {
        blackboard->InitializeBlackboard(*btree->BlackboardAsset);
    }
}

UBlackboardComponent* AGuard_AI::GetBlackboard() const
{
    return blackboard;
}
```

**Primjer 1.** Inicijalizacija Behavior Treea i Blackboarda

Inicijalizacija *Behavior Treea* je spremljenja u *obj* varijabli gdje se pomoću *FObjectFinder* funkcije pronalazi ponuđena klasa objekta. Ta se inicijalizacija provjerava te, ako je uspješna, objekt sprema u *btree* varijablu. Zatim se *Behavior Tree* komponenta i *Blackboard* komponenta dohvaćaju radi kasnijeg korištenja. SetupPerceptionSystem() funkcija inicijalizira sustav za percepciju. BeginPlay() funkcija se obrađuje pri pokretanju projekta. Unutar te funkcije se pokreće *Behavior Tree*. OnPossess() funkcija se pokreće kada *Actora* posjeduje *Controller* (taj *Controller* može biti AI ili igrač). Unutar OnPossess() funkcije se inicijalizira *Blackboard*, dok se pomoću GetBlackboard() funkcije dohvaća *Blackboard*.

U primjeru 2. je prikazano postavljanje sustava za percepciju.

```cpp
// Called when Guard Perception System is updated
void AGuard_AI::OnUpdated(TArray<AActor*> const& updated_actors)
{
    for (size_t x = 0; x < updated_actors.Num(); ++x)
    {
        // Get all updated perception components
        FActorPerceptionBlueprintInfo info;
        GetPerceptionComponent()->GetActorsPerception(updated_actors[x], info);
        for (size_t k = 0; k < info.LastSensedStimuli.Num(); ++k)
        {
            // Check component type
            FAIStimulus const stim = info.LastSensedStimuli[k];
            if (stim.Tag == "Noise")
            {
                // Set blackboard Investigate value and target location
                GetBlackboard()->SetValueAsBool(bb_keys::is_investigating, stim.WasSuccessfullySensed());
                GetBlackboard()->SetValueAsVector(bb_keys::target_location, stim.StimulusLocation);
            }
            else if (stim.Type.Name == "Default__AISense_Sight")
            {
                // Set blackboard Can see Player value and Has been Chasing value
                GetBlackboard()->SetValueAsBool(bb_keys::can_see_player, stim.WasSuccessfullySensed());
                GetBlackboard()->SetValueAsBool(bb_keys::has_been_chasing, true);
            }
        }
    }
}

void AGuard_AI::SetupPerceptionSystem()
{
    // Create and initialise sight configuration object
    sight_config = CreateDefaultSubobject<UAISenseConfig_Sight>(TEXT("Sight Config"));
    if (sight_config)
    {
        SetPerceptionComponent(*CreateDefaultSubobject<UAIPerceptionComponent>(TEXT("Perception Component")));
        sight_config->SightRadius = 1500.0f;
        sight_config->LoseSightRadius = sight_config->SightRadius + 50.0f;
        sight_config->PeripheralVisionAngleDegrees = 90.0f;
        sight_config->SetMaxAge(5.0f);
        sight_config->AutoSuccessRangeFromLastSeenLocation = 100.0f;
        sight_config->DetectionByAffiliation.bDetectEnemies = true;
        sight_config->DetectionByAffiliation.bDetectFriendlies = true;
        sight_config->DetectionByAffiliation.bDetectNeutrals = true;

        // Add sight configuration component to perception component
        GetPerceptionComponent()->SetDominantSense(*sight_config->GetSenseImplementation());
        GetPerceptionComponent()->OnPerceptionUpdated.AddDynamic(this, &AGuard_AI::OnUpdated);
        GetPerceptionComponent()->ConfigureSense(*sight_config);
    }
```

**Primjer 2.** Postavke sustava za percepciju

OnUpdated() funkcija se poziva svaki put kada je sustav za percepciju ažuriran, tj. svaki put kada NPC vidi igrača, izgubi igrača iz vida ili čuje igrača. Unutar funkcije se svi ažurirani podatci sustava za percepciju provjeravaju da li pripadaju vidu ili sluhu. Ako pripada sluhu (tj. igrač je pustio zvuk na svojoj lokaciji) postavljaju se *Blackboard* ključevi koji označuju da li NPC istražuje zvuk i lokaciju odakle je zvuk čut. Inače se provjerava da li pripada vidu, te se postavljaju *Blackboard* ključevi koje označuju da li NPC vidi igrača i da li ga lovi.

Unutar SetupPerceptionSystem() funkcije se postavljaju postavke za sustav za percepciju (radijus unutar kojeg NPC može vidjeti igrača, kut perifernog vida i sl.). Unutar *Character* klase od projekta je potrebno postaviti sustav za stimulaciju kako bi NPC primao stimulatore od igrača. To je prikazano u primjeru 3.

```cpp
void ADiplomskiProjectCharacter::SetupStimulus()
{
    stimulus = CreateDefaultSubobject<UAIPerceptionStimuliSourceComponent>(TEXT("stimulus"));
    stimulus->RegisterForSense(TSubclassOf<UAISense_Sight>());
    stimulus->RegisterWithPerceptionSystem();
}
```

**Primjer 3.** Postavljanje stimulatora

Nakon postavljanja sustava za percepciju, potrebno je napraviti putanju koju će NPC pratiti kada ne vidi igrača. Potrebno je napraviti novu *Actor* C++ klasu (nazvanu PatrolPath) i postaviti kod koji je prikazan na primjeru 4.

```
UCLASS()
class DIPLOMSKIPROJECT_API APatrolPath : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    APatrolPath();

    FVector GetPatrolPoint(int const index) const;
    int num() const;

private:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "AI", meta = (MakeEditWidget = "true", AllowPrivateAccess = "true"))
        TArray<FVector> patrol_points;
};
```

**Primjer 4.** PatrolPath klasa - putanja koju će Guard NPC pratiti

UPROPRETY funkcija stvara postavke kojim se može manipulirati unutar Unreal Enginea. U primjeru je prikazano stvaranje liste koja sadrži koordinate točaka. Unutar Unreal Enginea moguće je dodavati te točke unutar mape, a njihove koordinate će se spremiti u PatrolPath *Actor*u. Funkcija GetPatrolPoint() vraća koordinate točke koja se nalazi primljenom indexu liste, dok num() funkcija vrača broj točaka u listi. Ove dvije funkcije su označene kao public kako bi se mogle pozivati unutar drugih klasa.

U sljedećim koracima će biti prikazano postavljanje *Behavior Treea* i izrada potrebnih zadataka. Prvi korak jest pravljenje zadataka pomoću kojih će NPC pronaći točke unutar PatrolPatha, doći do tih točaka po redu, te na kraju ovisno o postavkama pratiti točke unatrag ili ponovno krenuti od prve točke.

Prvo je u *Behavior Tree* potrebno postaviti provjeru da li NPC može vidjeti igrača. To se obavlja tako da se u *Selectornodeu* doda *BlackboardDecorator* koji će provjeravati *CanSeePlayerBlackboard* ključ. Ako je ključ postavljena na TRUE (NPC vidi igrača), obavljaju se zadatci. Iz tog *Selector*a potrebno je postaviti *Sequencenode* i *BlackboardDecorator* koji će provjeriti *IsInvestigating* ključ. Kako bi NPC mogao pratiti putanju, potrebno je napraviti dva zadatka: pronalaženje koordinate trenutne točke na putanji i postavljanje sljedeće točke kao trenutne, te *Service* klasu koja će se koristiti za promjenu brzine kretanja NPC-a. C++ klase koje je potrebno stvoriti za oba zadatka se zovu BTTask_*Blackboard*Base dok za *Service* klasu je potrebno stvoriti BT*Service_Blackboard*Base.

U primjeru 5. prikazan je kod unutar zadatka za pronalaženje lokacije trenutne točke.

```cpp
UFindPatrolPointCPP::UFindPatrolPointCPP(FObjectInitializer const& object_initializer)
{
    NodeName = TEXT("Find Patrol Path Point");
}

EBTNodeResult::Type UFindPatrolPointCPP::ExecuteTask(UBehaviorTreeComponent& owner_comp, uint8* node_memory)
{
    // Get AI controller and check cast
    AGuard_AI* const controller = Cast<AGuard_AI>(owner_comp.GetAIOwner());
    if (controller)
    {
        // Get current Patrol Path index from blackboard
        int const index = controller->GetBlackboard()->GetValueAsInt(bb_keys::patrol_path_index);

        // Use index to get current Patrol Path from NPC's reference to Patrol Path
        AGuard_NPC* const npc = Cast<AGuard_NPC>(controller->GetPawn());
        if (npc)
        {
            FVector const point = npc->GetPatrolPath()->GetPatrolPoint(index);

            // Transform point to global position using it's parent
            FVector const global_point = npc->GetPatrolPath()->GetActorTransform().TransformPosition(point);

            // Write global path point to blackboard
            controller->GetBlackboard()->SetValueAsVector(bb_keys::patrol_path_vector, global_point);

            // Finish with success
            FinishLatentTask(owner_comp, EBTNodeResult::Succeeded);
            return EBTNodeResult::Succeeded;
        }
    }

    // Finish with fail
    FinishLatentTask(owner_comp, EBTNodeResult::Failed);
    return EBTNodeResult::Failed;
}
```

**Primjer 5.** Zadatak za pronalaženje trenutne točke putanje

Unutar konstruktora klase je potrebno unijeti naziv *nodea*. ExecuteTask() funkcija služi za stvaranje funkcionalnosti koje će se pokrenuti pri pokretanju zadatka. Kako bi se lokacija točke putanje pronašla potrebno je preko *AI Controllera* od NPC-a dohvatiti *Blackboard*cjelobrojni ključ koji sadrži indeks trenute točke putanje. Dohvaćanje *AI Controllera* se obavlja preko *owner_comp* varijable koja u sebi sadrži *Actor* kojeg *Behavior Tree* upravlja. Nakon provjere da li je *AI Controller* uspješno dohvaćen, dohvaća se indeks trenutne točke putanje preko *Blackboarda*, te preko NPC-a se dohvaća vektor trenutačne točke. Koordinate te točke se transformiraju u globalnu poziciju i spremaju u *Blackboard* ključ patrol_path_vector. Zadatak je na kraju potrebno završiti i vratiti uspjeh (u slučaju da provjere nisu uspjele vrača se neuspjeh).

Inkrementiranje točke koje je prikazana u primjeru 6. potrebno je postaviti u *Behavior Treeu* nakon *MoveTo* naredbe kako bi se indeks postavio tek nakon što NPC dođe do točke. To će osigurati, ako NPC lovi igrača i izgubi ga iz vida, da se vrati na prijašnju putanju.

```cpp
EBTNodeResult::Type UIncrementPathIndexCPP::ExecuteTask(UBehaviorTreeComponent& owner_comp, uint8* node_memory)
{
    // Get AI controller
    AGuard_AI* const controller = Cast<AGuard_AI>(owner_comp.GetAIOwner());
    if (controller)
    {
        // Get NPC
        AGuard_NPC* const npc = Cast<AGuard_NPC>(controller->GetPawn());
        if (npc)
        {
            int const points_num = npc->GetPatrolPath()->num();
            int const max_index = points_num - 1;

            // Get current index
            int index = controller->GetBlackboard()->GetValueAsInt(bb_keys::patrol_path_index);

            // Check if looping is on and current index is last point
            if (loop && index >= max_index)
            {
                // Set next index to starting point
                controller->GetBlackboard()->SetValueAsInt(bb_keys::patrol_path_index, 0);

                //Finish with success
                FinishLatentTask(owner_comp, EBTNodeResult::Succeeded);
                return EBTNodeResult::Succeeded;
            }
            // Set Direction to Reverse if at last index (at the end of patrol path)
            else if (index >= max_index && direction == EDirectionType::Forward)
            {
                direction = EDirectionType::Reverse;
            }
            // Set Direction to Forward if at first index (at the start of patrol path)
            else if (index == 0 && direction == EDirectionType::Reverse)
            {
                direction = EDirectionType::Forward;
            }

            // Increment or decrement index based on direction and set blackboard value
            controller->GetBlackboard()->SetValueAsInt(bb_keys::patrol_path_index, (direction == EDirectionType::Forward ? std::abs(++index) : std::abs(--index)));

            // Finish with success
            FinishLatentTask(owner_comp, EBTNodeResult::Succeeded);
            return EBTNodeResult::Succeeded;
        }
    }

    // Finish with fail
    FinishLatentTask(owner_comp, EBTNodeResult::Failed);
    return EBTNodeResult::Failed;
}
```

**Primjer 6.** Zadatak za inkrementiranje trenutne točke putanje

Potrebno je dohvatiti NPC i njegov *AI Controller*, te preko njih dohvatiti veličinu PatrolPath liste, *Loop* varijablu koja označava kako se NPC miče na kraju putanje (ako je TRUE onda na zadnjoj točki se vraća na prvu, dok FALSE označava da ide točkama unatrag, od zadnje točke unatrag do prve) i *Direction* varijablu koja označava smjer u kojemu se NPC kreće po putanji. Potrebno je također dohvatiti *Blackboard* ključ za trenutni indeks i provjeriti da li je trenutni indeks jednak veličini liste (tj. da li se NPC nalazi na zadnjoj točki). Zatim je potrebno provjeriti da li je *Loop* varijabla TRUE i da li se NPC nalazi na zadnjoj točki, te po uspješnoj provjeri postaviti u *Blackboardu* index putanje na početak te završiti zadatak. Ako je *Loop*FALSE onda je potrebno provjeriti da li se NPC nalazi na početku ili na kraju putanje i postaviti *Direction* varijablu (postavlja se unatrag ako je na kraju putanje, inače se postavlja unaprijed). Time se osigurava da NPC na kraju/početku putanje okrene smjer. Na kraju je potrebno povećati indeks putanje ako je *Direction* namješten na naprijed, inače se indeks smanjuje, te završiti zadatak.

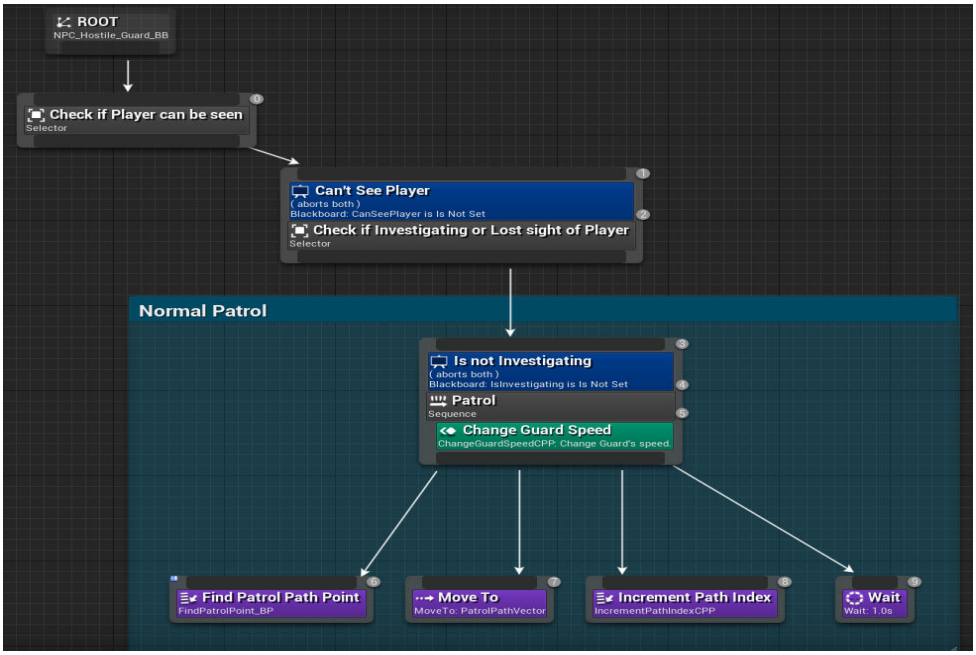
Dok se *Behavior Tree* Task obavlja tek kada se pokrene, *Service* se izvršava sve dok se grana u kojoj je smještena izvršava. (5)*Service* u ovom primjeru će postaviti brzinu kretanja NPC-a na željenu vrijednost. Unutar zaglavlja*Servicea* je potrebno napraviti UPROPERTY varijablu za brzinu što omogućuje postavljanje vrijednosti varijable unutar *Behavior Treea*. Zatim je potrebno

u OnBecomeRelevant() funkciji preko NPC klase dohvatiti *MovmentComponent* i postaviti MaxWalkSpeed(). OnBecomeRelevant() se izvršava pri izvršavanju grane unutar koje je *Service* smješten. Kod je prikazan u primjeru 7.

```cpp
void UChangeGuardSpeedCPP::OnBecomeRelevant(UBehaviorTreeComponent& owner_comp, uint8* node_memory)
{
    Super::OnBecomeRelevant(owner_comp, node_memory);

    // Get AI
    auto const controller = owner_comp.GetAIOwner();
    if (controller)
    {
        // Get NPC
        AGuard_NPC* const npc = Cast<AGuard_NPC>(controller->GetPawn());
        if (npc)
        {
            npc->GetCharacterMovement()->MaxWalkSpeed = speed;
        }
    }
}
```

**Primjer 7.** Service za promjenu brzine kretanja NPC-a

Unutar *Behavior Treea* je potrebno postaviti sve zadatke tako da se provjeri da li NPC vidi igrača i da li trenutačno istražuje zvuk, te pronađe lokaciju trenutne točke, krene prema njoj, inkrementira indeks točke putanje i pričeka 1 sekundu. Ova grana *Behavior Treea* je prikazana u primjeru 8.




**Primjer 8.** Behavior Tree grana za normalno patroliranje

### 3.3.3. Lovljenje igrača

Kada NPC vidi igrača, mora prekinuti zadatke koje trenutno obavlja, povećati brzinu, pronaći lokaciju igrača i trčati za njim. Ako se NPC nalazi blizu igrača, mora ga udariti i pri uspješnom udarcu ozljeđuje igrača.

Kako bi igrač mogao biti ozlijeđen, potrebno je igraču dodati varijablu koja će pratiti koliko energije igrač ima, te funkcije koje dohvaćaju i mijenja tu vrijednost. Unutar DiplomskiProjekt*Character* header-a je potrebno stvoriti: Private UPROPERTY varijablu koja će predstavljati igračevu trenutnu energiju (u ovom primjeru ta varijabla se naziva *PlayerHealth*), Protected UPROPERTY varijablu koja predstavlja koliko maksimalno energije igrač može imati (*MaxHealth*), te public funkcije za dohvaćanje tih varijabli (GetPlayerHealth() i GetMaxHealth()) kao i funkciju za ažuriranje *PlayerHealth* varijable (UpdatePlayerHealth()). Primjer 9. prikazuje stvorene komponente unutar *headera*, dok primjer 10. prikazuje kod unutar stvorenih funkcija.

```
protected:
    // Maximum Player Health Points
    UPROPERTY(EditAnywhere, Category = "Health", Meta = (BlueprintProtected = "true"))
        float MaxHealth;

private:
    // Current Player Health Points
    UPROPERTY(VisibleAnywhere, Category = "Health")
        float PlayerHealth;

    class UAIPerceptionStimuliSourceComponent* stimulus;
    void SetupStimulus();

public:
    // Accessor function for Maximum Player Health Points
    UFUNCTION(BlueprintPure, Category = "Health")
        float GetMaxHealth();

    // Accessor function for Current Player Health
    UFUNCTION(BlueprintPure, Category = "Health")
        float GetPlayerHealth();

    // Function for updating Player's Health Points
    UFUNCTION(BlueprintCallable, Category = "Health")
        void UpdatePlayerHealth(float HealthChange);
};
```

**Primjer 9.** Funkcije unutar headera za dohvaćanje i ažuriranje PlayerHealth varijable

```
// Returns Maximum Health Points
float ADiplomskiProjectCharacter::GetMaxHealth()
{
    return MaxHealth;
}

// Returns Player's current Health Points
float ADiplomskiProjectCharacter::GetPlayerHealth()
{
    return PlayerHealth;
}

// Updates Player's Current Health Points (called whenever the Player gains/loses Health Points)
void ADiplomskiProjectCharacter::UpdatePlayerHealth(float HealthChange)
{
    // Change Player's Current Health
    PlayerHealth += HealthChange;

    // Check if Player's Current Health Points exceed Maximum Health Points
    if (PlayerHealth > MaxHealth)
    {
        // Set Player's Current Health Points to Maximum Health Points
        PlayerHealth = MaxHealth;
    }
}
```

**Primjer 10.** Kod funkcija za dohvaćanje i ažuriranje PlayerHealth varijable

GetMaxHealth() i GetPlayerHealth() vraćaju respektivne varijable, dok UpdatePlayerHealth() prima vrijednost koja predstavlja količinu promjene *PlayerHealtha*. Zatim se *PlayerHealth* zbraja sa *HealthChange*, te ako je *PlayerHealth* veći od *MaxHealth* onda se *PlayerHealth* postavlja na istu vrijednost kao *MaxHealth*. Time se osigurava da igrač nikada nema više energije nego što je dozvoljeno.

Unutar *Behavior Treea* je potrebno postaviti *BlackboardDecorator* koji provjerava da li NPC vidi igrača i ako može vidjeti igrača, pokreće se njegova grana. Kako bi se svi zadatci prekinuli kada NPC vidi igrača, unutar *Details* prozora od stvorenog *Decoratora* je potrebno postaviti *Observer Aborts* na *Both*. Pomoću *Observer Aborts* se označava koji dijelovi *Behavior Treea* se prekidaju kada se promijeni rezultat provjere *Decoratora*. Moguće postavke su: *None* (bez prekida), *Self* (prekida izvršavanje grane *Decoratora*), *Lower Priority* (prekidaju se grane manjeg prioriteta od *Decoratora*) i *Both*. Postavljanje *Observer Aborts* na *Both* označava da kada NPC vidi igrača, sve grane manjeg prioriteta se prekidaju, a ako izgubi igrača, prekida se grana koja je odgovorna za lovljenje igrača.

Kada NPC čuje zvuk koji je igrač proizveo, postavlja se *IsInvestigating* varijabla na TRUE, te u *Behavior Treeu* se pokreće grana zadužena za istraživanje lokacije zvuka. Ako NPC vidi igrača trebao bi zaustaviti istraživanje i nakon što izgubi igrača, ne bi smio nastaviti istraživanje. Zbog

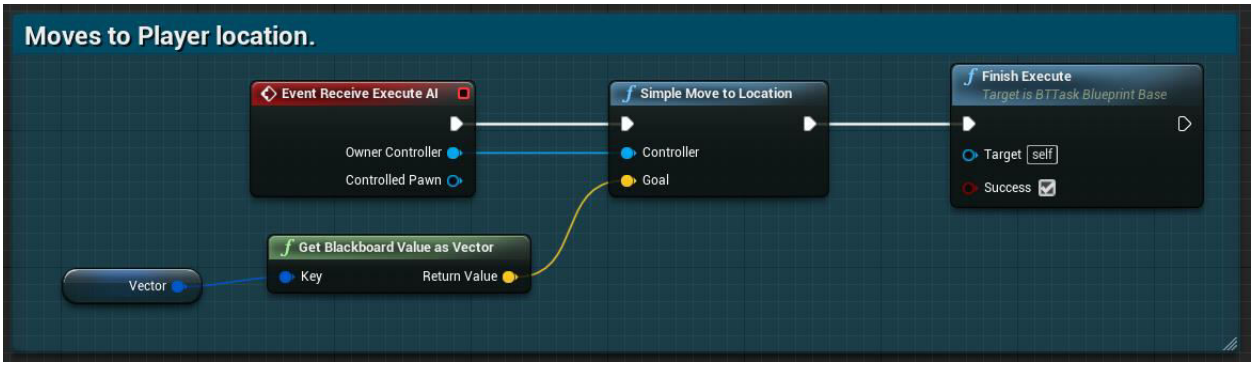
toga je potrebno napraviti jednostavan *Behavior Tree* zadatak koji će postaviti *IsInvestigating* varijablu na FALSE.

Da bi NPC mogao loviti igrača potrebno je napraviti zadatak koji vraća igračevu lokaciju. Taj zadatak će uzeti referencu na igrača i u *Blackboard* postaviti koordinate igrača unutar *TargetLocation* varijable kao što je prikazano na primjeru 11.

```cpp
// Find Player location
EBTNodeResult::Type UFindPlayerLocationCPP::ExecuteTask(UBehaviorTreeComponent& owner_comp, uint8* node_memory)
{
    // Get Player character and NPC
    ADiplomskiProjectCharacter* player = Cast<ADiplomskiProjectCharacter>(UGameplayStatics::GetPlayerCharacter(GetWorld(), 0));

    // Check cast
    if (player)
    {
        // Set blackboard Target Location value
        owner_comp.GetBlackboardComponent()->SetValueAsVector("TargetLocation", player->GetActorLocation());

        // Finish with success
        FinishLatentTask(owner_comp, EBTNodeResult::Succeeded);
        return EBTNodeResult::Succeeded;
    }

    // Finish with fail
    FinishLatentTask(owner_comp, EBTNodeResult::Failed);
    return EBTNodeResult::Failed;
}
```

**Primjer 11.** Zadatak za pronalaženje koordinata na kojima se igrač nalazi

Unutar Unreal Enginea postoji *Blueprint* funkcija "Simple *Move To*", koja pomiće *Actora* do primljene lokacije. Ako se usred micanja *Actora* promijene koordinate prema kojoj ide, *Move To* će prvo završiti micanje *Actora* do stare lokacije, a onda pomaknuti do nove dok Simple *Move To* funkcija automatski promjeni smjer kretanja *Actora* prema novoj lokaciji. Zbog toga je potrebno napraviti novi *Blueprint* zadatak sa *SimpleMove To* funkcijom kako bi NPC pravilno lovio igrača. *SimpleMove To* funkcija prima *Actora* kojeg pomiče (to će biti *Controlled Pawn* od *Event Recieve Execute AInodea*) i koordinate prema kojima se miče (te koordinate se primaju od *BlackboardTargetLocation* varijable). Ovaj zadatak je rađen u *Blueprintu* jer će se ponovno koristiti u drugom NPC-u. U ovom projektu ovaj zadatak je nazvan *Chase Player* i prikazan je u primjeru 12.




**Primjer 12.** Blueprint zadatka za pmicanje NPC-a prema primljenim koordinatama

Sljedeći korak je postavljanje funkcionalnosti za udaranje unutar C++ klase od NPC-a. Potrebno je napraviti kutiju za koliziju na desnoj šaci od NPC-a koja će služiti kako bi se otkrilo da li je NPC udario igrača i ako jest, ozlijedi igrača. Unutar *headera* od NPC-a je potrebno stvoriti UPROPERTY *UBoxComponent* varijablu (nazvana *right_fist_collision*) koja je potrebna za stvaranje i lijepljenje kutije za koliziju na ruci NPC-a. U konstruktoru unutar *Source filea* od NPC-a se za *right_fist_collision* postavlja veličina kutije i profil kolizije koji se postavlja na "*NoCollision*" profil kako bi kutija prolazila kroz sve komponente u projektu (inače kutija ne bi prolazila kroz komponente). U BeginPlay() funkciji se postavljaju pravila za kutiju, zaljepljuje se na desnu ruku NPC-a i postavlja se funkcije koje će se pozvati kada kutija započne preklapanje sa drugim komponentom projekta (tj. funkcija koja se poziva kada kutija preklapa igrača). Na primjeru 13. je prikazan kod u konstruktoru i u BeginPlay() funkciji.

```
AGuard_NPC::AGuard_NPC() :
    right_fist_collision_box(CreateDefaultSubobject<UBoxComponent>(TEXT("RightFistCollisionBox")))
{
    // Set this character to call Tick() every frame.  You can turn this off to improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

    // Smoothes NPC rotation
    bUseControllerRotationYaw = false;
    // GetCharacterMovement()->bUseControllerDesiredRotation = true;
    GetCharacterMovement()->bOrientRotationToMovement = true;

    // Setup collision box for right fist
    if (right_fist_collision_box)
    {
        FVector const extent(5.0f);
        right_fist_collision_box->SetBoxExtent(extent, false);
        right_fist_collision_box->SetCollisionProfileName("NoCollision");
    }
}

// Called when the game starts or when spawned
void AGuard_NPC::BeginPlay()
{
    Super::BeginPlay();

    // Setup for collision box on right fist
    if (right_fist_collision_box)
    {
        FAttachmentTransformRules const rules(EAttachmentRule::SnapToTarget, EAttachmentRule::SnapToTarget, EAttachmentRule::KeepWorld, false);
        right_fist_collision_box->SetRelativeLocation(FVector(-7.0f, 0.0f, 0.0f));
        right_fist_collision_box->AttachToComponent(GetMesh(), rules, "hand_rSocket");

        right_fist_collision_box->OnComponentBeginOverlap.AddDynamic(this, &AGuard_NPC::OnAttackOverlapBegin);
        right_fist_collision_box->OnComponentEndOverlap.AddDynamic(this, &AGuard_NPC::OnAttackOverlapEnd);
    }
}
```

**Primjer 13.** Postavljanje *Collision Box* komponente unutar konstruktora i BeginPlay() funkcije od *Guard* NPC-a

U sljedećem koraku je potrebno stvoriti attack funkciju koja će pri pozivu pokrenuti animaciju, te funkcija koja se poziva kada kutija za koliziju započne preklapanje. Ta funkcija će provjeriti da li je kutija preklapa igrača, te mu oduzeti energiju. Za pokretanje animacije je prvo potrebno stvoriti UPROPERTY anim varijablu pomoću koje će se u Unreal Engine projektu moći postaviti željena animacija (u ovom projektu ta varijabla je nazvana "*montage*" i animacija je već postavljena). Zatim se stvara MeleeAttack() funkcija koja pri pozivu pokreće animaciju. Za provjere kolizije se stvara OnAttackOverlapBegin() funkcija (čiji je poziv definiran u

konstruktoru). Unutar funkcije se provjerava da li preklopljeni element pripada igraču i oduzima mu energije ako jest. Zadnji korak jest stvaranje *Behavior Tree* zadatka koji pokreće MeleeAttack() funkciju preko reference na NPC. Kod unutar C++ klase od NPC-a i kod *Behavior Tree* zadatka je prikazan u primjeru 14.

```cpp
// Plays Melee Attack animation
void AGuard_NPC::MeleeAttack()
{
    // Check if montage is set
    if (montage)
    {
        // Play animation
        PlayAnimMontage(montage);
    }
}

// Called when fist collision box has overlapped an actor
void AGuard_NPC::OnAttackOverlapBegin(UPrimitiveComponent* const overlapped_component,
    AActor* const other_actor, UPrimitiveComponent* other_component, int const other_body_index,
    bool const from_sweep, FHitResult const& sweep_result)
{
    // Check if overlapped actor is the Player
    if (ADiplomskiProjectCharacter* const player = Cast<ADiplomskiProjectCharacter>(other_actor))
    {
        // Check if player wasn't already damaged during animation
        if (!has_damaged)
        {
            // Damage player and set has_damaged value to true
            player->UpdatePlayerHealth(-1 * damage);
        }
    }
}
```

```cpp
EBTNodeResult::Type UMeleeAttackCPP::ExecuteTask(UBehaviorTreeComponent& owner_comp,
    uint8* node_memory)
{
    // Get AI controller and NPC
    AAIController* const controller = owner_comp.GetAIOwner();
    AGuard_NPC* const npc = Cast<AGuard_NPC>(controller->GetPawn());

    // Check cast
    if (controller && npc)
    {
        // If the montage has finished playing, play it again (stops "jittering")
        if (MontageHasFinished(npc))
        {
            npc->MeleeAttack();
        }
    }

    // Finish with success
    FinishLatentTask(owner_comp, EBTNodeResult::Succeeded);
    return EBTNodeResult::Succeeded;
}
```

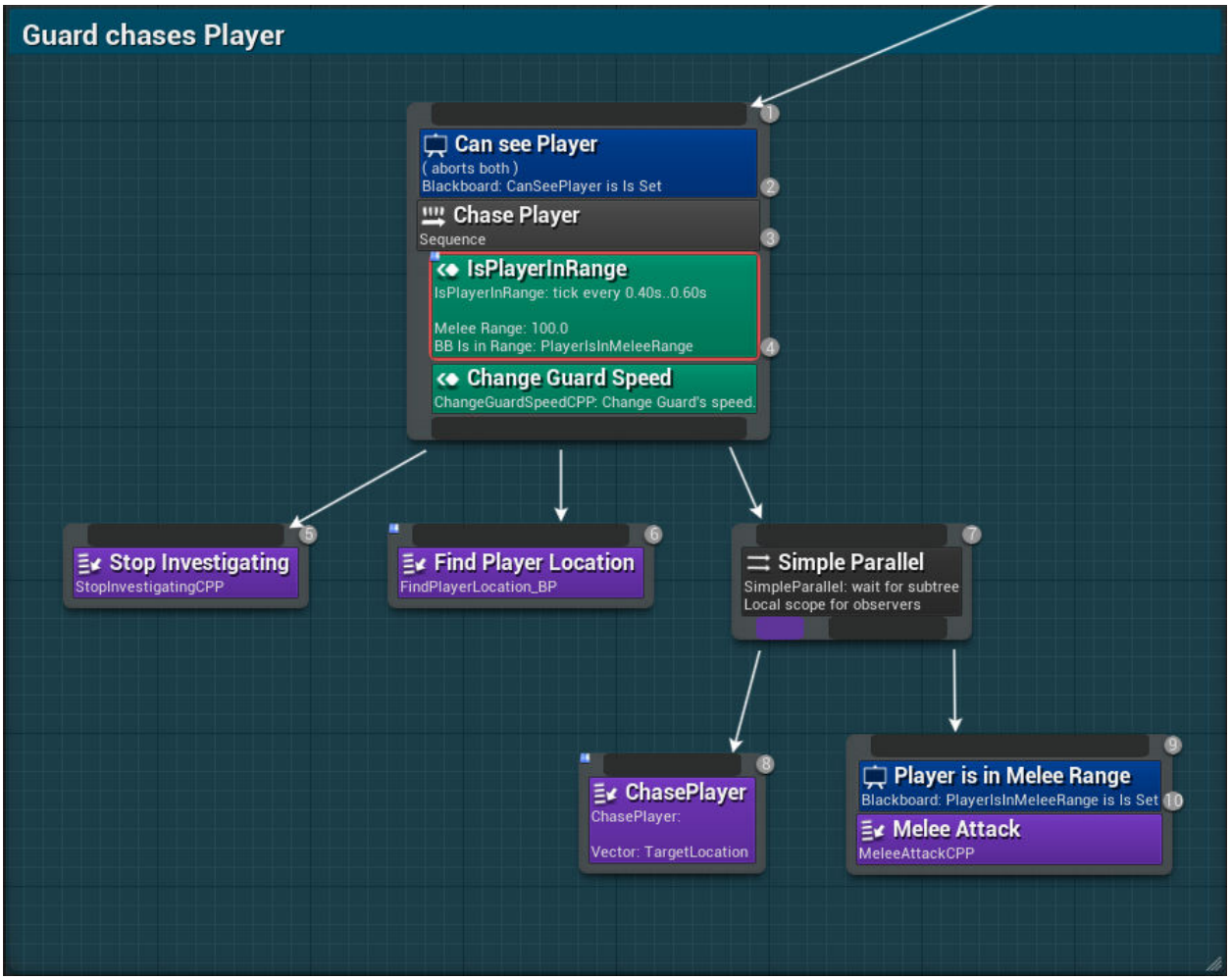
**Primjer 14.** Kod za pokretanje naimacije i ozlijeđivanje igrača (lijevo) i kod za pokretanje napada (desno)

Za promjenu brzine NPC-a i provjere udaljenosti od igrača se koriste *Service* klase. Promjena brzine se obavlja dohvaćanje *Movement* komponenta od NPC-a i promjene *MaxWalkSpeed* varijable na željenu vrijednost. Vrijednost nove brzine je stvorena kao UPROPERTY varijabla kako bi njeno postavljanje bilo jednostavnije (nema potrebe za ponovnim otvaranjem i izmjenom C++ klase svaki put kada se želi promijeniti brzina, nego se postavlja u *Behavior Treeu*).

*Service* za provjeru udaljenosti od igrača će biti korišten i za druge NPC-ove, te je iz tog razloga rađen u *Blueprintu* kako bi se izbjeglo ponovna izrada istog koda za drugi NPC. Unutar Unreal Engine postoji GetDistanceTo funkcija koja prima dva *Actora* (u ovom primjeru to su NPC i igrač), uzima njihove koordinate i vraća udaljenost između njih. Ta udaljenost se uspoređuje sa MeleeRange varijablom koja označava maksimalnu udaljenost unutar koje NPC može pokušati udariti igrača. Ako je udaljenost između igrača i NPC-a manja od Melee Range, postavlja se *Blackboard* varijabla *PlayerIsInMeleeRange* koja se koristi za pokretanje dio *Behavior Tree* grane zadužen za napadanje igrača.

Nakon izrade svih potrebnih zadataka, potrebno je zadatke i provjere u *Behavior Treeu* postaviti u pravilan redoslijed kako bi NPC pravilno lovio igrača. Unutar *Sequencenodea* koji provjerava da li NPC vidi igrača se postavljaju *Servicei* za promjenu brzine i za provjeru udaljenosti od igrača. Iz *Sequencease* stvaraju zadatci sljedećim redoslijedom: prvo je potrebno zaustaviti istraživanje zvuka kako se NPC ne bi vratio na istraživanje nakon što izgubi igrača iz vida. Zatim se traži i sprema lokacija igrača. Na kraju, kako bi NPC istovremeno trčao za igračem i (ako je blizu igrača) pokušao ga udariti, potrebno je stvoriti *Simple Parallelnode* koji omogućuje istovremeno pokretanje dva zadatka. Taj *node* sadrži dva izlaza, te se na lijevi postavlja *ChasePlayer* zadatak, dok se na desni postavlja *MeleeAttack* zadatak. Na *MeleeAttack* zadatak je

potrebno postaviti *Decorator* koji provjerava *PlayerIsInMeleeRange* varijablu. Time se osigurava da NPC trći za igračem te ga istovremeno pokušava udariti ako je blizu igrača. Grana *Behavior Treea* za lovljenje igrača je prikazana u primjeru 15.




**Primjer 15.** Behavior Tree grana za lovljenje igrača

### 3.3.4. Gubljenje igrača

Kada NPC izgubi igrača iz vida, *Blackboard* ključ *CanSeePlayer* se postavlja na FALSE, te se time prekida izvršavanje grane *Behavior Treea* zadužene za lovljenje igrača. Međutim, *Behavior Tree* pokreće sljedeću granu po prioritetu koju je moguće uspješno izvršiti, a to je grana zadužena za normalno patroliranje. Time bi se NPC, nakon što je izgubio igrača iz vida, vratio na normalnu putanju iako treba istražiti zadnju lokaciju gdje je vidio igrača. Kako bi NPC pravilno pokrenuo istraživanje potrebno je provjeriti *HasBeenChasingBlackboard* ključ. Ako je varijabla TRUE (NPC je taman završio lovljenje igrača) pokreće se istraživanje, inače se pokreće normalno patroliranje.

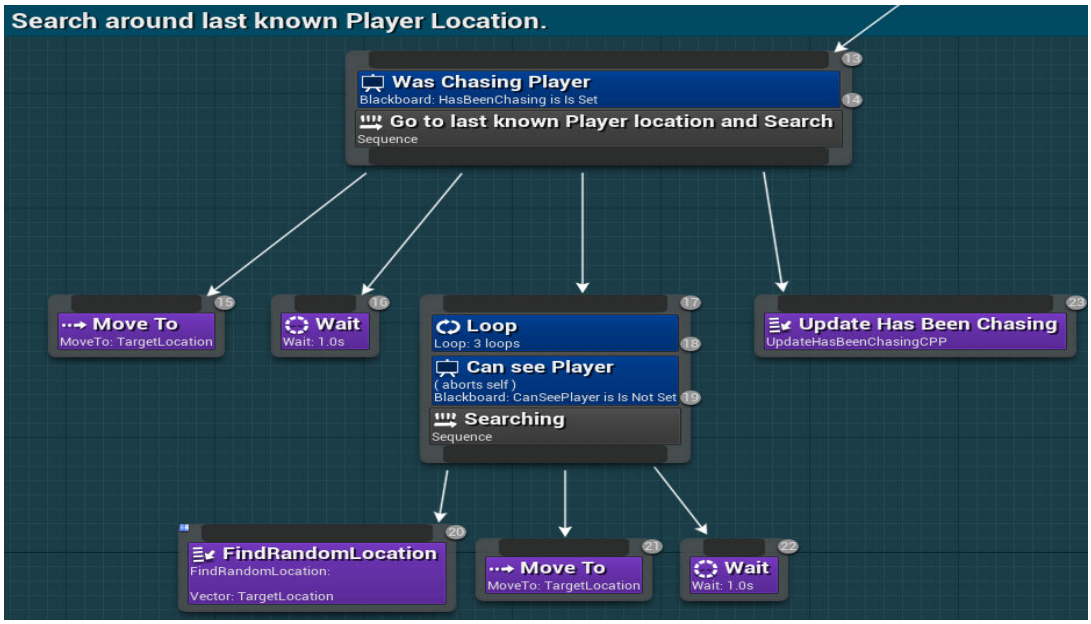
Pri istraživanju igračeve zadnje lokacije, NPC mora krenuti prema toj lokaciji, prčekati određeno vrijeme (1 sekundu u ovom projektu) i određeni broj puta krenuti prema nasumičnoj lokaciji i opet prčekati određeno vrijeme. Tijekom istraživanja se obavlja provjera da li NPC može vidjeti igrača, pri čemu se prekida obavljanje ove grane i pokreće grana za lovljenje. Nakon izvršetka grane za istraživanje bez pronalaženja igrača, *Blackboard* ključ *HasBeenChasing* se postavlja na FALSE, čime se prekida obavljanje grane za istraživanje zadnje lokacije i pokreće grana za normalno patroliranje.

Tijekom lova, igračeva lokacija se ažurira cijelo vrijeme dok NPC može vidjeti igrača. Kada se igrač uspješno sakrije u *TargetLocation* ključu će biti spremljena zadnja igračeva lokacija, te će se koristiti u *MoveTo* naredbi kako bi NPC, pri pokretanju grane za istraživanje, mogao krenuti prema tim koordinatama. Zatim se koristi *Wait* naredba koja pauzira pokretanje sljedećeg zadatka u *Behavior Treeu* određeno vrijeme.

Sljedeći korak jest ponavljanje postupka istraživanja u nasumičnim smjerovima određeni broj puta što se postiže pomoću *LoopDecoratora*. *Loop* ponavlja određeni broj puta svoju granu *Behavior Treea* nakon čega nastavlja sa sljedećim zadatcima. *Loop* ne očitava prekidanje grane u kojoj se nalazi, tj. ako tijekom istraživanja NPC vidi igrača, neće ga loviti dok potpuno ne završi *Loop*. Zbog toga je potrebno u *Nodeu* unutar kojeg je postavlje *Loop* također postaviti *Decorator* za provjeru *CanSeePlayer* ključa. Time se osigurava da NPC lovi igrača čim mu dođe unutar vida.

Za istraživanje u nasumičnom smjeru se koristi *Blueprint* zadatak koji je napravljen za *Wanderer* NPC i koji će pronaći nasumičnu lokaciju oko NPC-a i spremiti krodinate u *TargetLocation*. Zatim je potrebno pomaknuti NPC-a prema tim koordinatama i postaviti *Wait* naredbu. Time će NPC poći do lokacije, pričekati i ponoviti postupak određeni broj puta dok cijelo vrijeme provjerava da li može vidjeti igrača. Na kraju grane za istraživanje zadnje lokacije je potrebno stvoriti jednostavan C++ zadatak koji postavlja *HasBeenChasing* ključ na FALSE kako bi se NPC vratio na normalno patroliranje.

Grana *Behavior Treea* zadužena za istraživanje zadnje lokacije igrača je prikazana na primjeru 16.




**Primjer 16.** Behavior Tree grana za istraživanje igračeve zadnje poznate lokacije

### 3.3.5. Istraživanje zvuka

Osim traženja vizualnog stimulatora, NPC također prima i audio stimulator kojeg igrač može proizvesti. Kada igrač proizvede zvuk na svojoj lokaciji (tako da "zazviždi"), NPC će prekinuti sa trenutnim zadatcima koje obavlja, poći prema lokaciji, te nakon određenog vremena vratiti se na normalnu putanju.

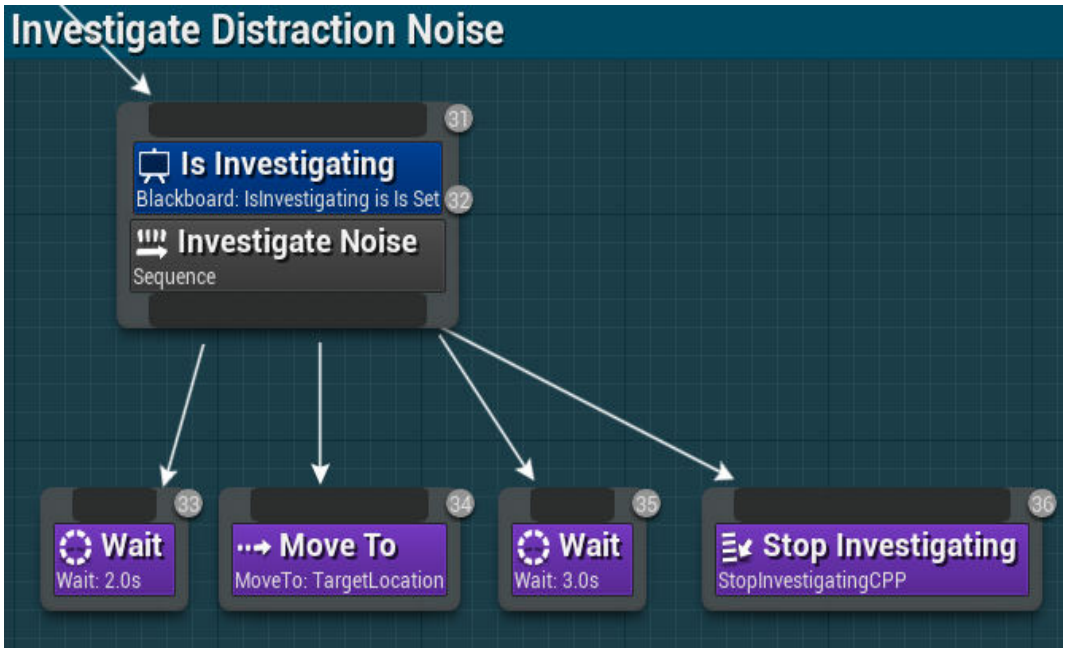
Kako bi igrač mogao proizvesti zvuk, potrebno je unutar *Blueprinta* od *FirstPersonCharacter* (to je *Actor* kojega igrač zaposjedne kada se pokrene projekt) i napraviti funkcionalnost koja pri pritiskom određene tipke (u ovom projektu ta tipka je "C") pokrene određeni zvuk i registrira taj zvuk kao stimulator. Za event *node* koji će pokrenuti ostatak *Blueprinta* je potrebno odabrati "*Keyboard Event*" i u ponuđenoj listi odabrati željenu tipku tipkovnice. Taj event će, nakon pokretanja projekta, osluškivati da li je odabrana tipka pritisnuta i ako jest, pokrenuti povezani *Blueprint*. Nakon pritiska tipke potrebno je pustiti zvuk na igračevoj lokaciji, što se postiže pomoću "*Play Sound at Location*" *nodea*. Na *Sound* ulazu od *nodea* se postavlja željeni zvuk (u projektu se koristi "*CompileSuccess*" zvuk), dok na Location ulazu se postavlja igračeva lokacija koja se dobiva sa "*GetActorLocation*" *node*om. Na kraju je taj zvuk potrebno registrirati kao stimulator kako bi ga NPC mogao čuti. Pomoću *node* "*Report Noise Event*" se stvara stimulator na primljenoj lokaciji (koja se uzima od "Get*Actor*Location" *nodea*) te će omogućiti NPC-u da čuje taj zvuk i reagira na njega.

Stvorena funkcionalnost unutar *FirstPersonCharacterBlueprinta* je prikazana na primjeru 17.


**Primjer 17.** Blueprint za puštanje zvuka na igračevoj lokaciji

Pri stvaranju sustava za percepciju u prijašnjem poglavlju je već postavljen sluh i ažuriranje *IsInvestigatingBlackboard* ključa, što ostavlja postavljanje jednostavne *Behavior Tree* grane. Jedini zadatak koji je potrebno napraviti jest postavljanje *IsInvestigating* ključa na FALSE, dok svi ostali zadatci postoje unutar Unreal Enginea.

Kada NPC čuje zvuk, treba prvo pričekati malo vremena (taj zadatak je postavlje tako da izgleda kao da NPC provjerava odakle je zvuk došao), poći prema toj lokaciji sa *MoveTo* naredbom, opet pričekati određeno vrijeme i postaviti *IsInvestigating* ključ na FALSE čime završava granu i vraća se na normalno patroliranje. Ova grana *Behavior Treea* je prikazana u primjeru 18.




**Primjer 18.** Behavior Tree grana za istraživanje lokacije na kojoj je igrač pustio zvuk

### 3.3.6. Pregled Guard NPC-a

Cilj ovog NPC-a jest simulirati ponašanje stražara koji patrolira određenu lokaciju i služi kao prepreka igraču. Igrač mora izbjeći NPC-jev vid, inače će NPC početi loviti igrača i pokušati ga ozlijediti. U tom slučaju, igrač mora pobjeći iz vida NPC-a i sakriti se dok ga NPC traži. Kako bi igrač mogao lakše izbjeći NPC-a, dodana je funkcija "zviždanja" koja omogućuje igraču da trenutno zaustavi NPC-jevo patroliranje i odmakne ga od putanje koju patrolira. Također ovaj NPC služi za upoznavanje sa izradom zadataka i funkcionalnosti u Unreal Engineu koristeći C++.

## 3.4. Healer NPC

Healer je NPC koji provjerava da li je igrač ozlijeđen i pokušava ga izliječiti. Healer u sebi može sadržavati određenu količinu energije sa kojom liječi igrača. Kada potroši tu energiju (tj. energija je ispod određene razine) NPC ide prema stanici za punjenje i obnavlja energiju na njoj. Ako na mapi postoji više stanica, onda će NPC ići prema najbližoj stanici.
Ovaj NPC sastoji se od dva dijela: *Actora* koji lijeći igrača i stanice na kojoj *Actor* puni svoje spremište energije.

### 3.4.1. *Blackboard*ključevi Healer NPC-a

*Blackboard* od NPC-a se sastoji od sljedećih ključeva važnih za obavljanje zadataka:

- ***PlayerNeedsHealing* (bool)** - ključ koji označava da li je igrač ozlijeđen (FALSE ako nije, TRUE ako jest). Koristi se za pokretanje *Behavior Tree* grane zadužene za liječenje igrača.
- ***LowCharge* (bool)** - ključ koji označava da li NPC ima malu količinu energije, ispod određene granice. Pomoću ključa se određuje kada NPC treba tražiti stanicu i napuniti energiju.
- ***TargetLocation* (vector)** - spremište koordinata igrača ili najbliže stanice, ovisno da li NPC mora liječiti igrača ili napuniti energiju.
- ***IsPlayerInRange* (bool)** - koristi se za provjeru udaljenosti od igrača. Ako je NPC blizu igrača (tj. ključ je postavljen na TRUE) onda ga može liječiti.
- ***HasNoCharge* (bool)** - ključ koji označava da li je NPC potrošio svu energiju. Koristi se kako bi NPC prekinu liječenje igrača i odmah krenuo prema stanici za punjenje.

### 3.4.2. Stanica za punjenje

Stanica za punjenje je *Actor* koji se sastoji od 3 komponente: *Mesh*, tj. 3D model koji će predstavljati stanicu u svijetu, *CollisionSphere* koja služi za otkrivanje kolizije kako bi se otkrilo da li se NPC nalazi na stanici i *ParticleSystem* u kojem će se postaviti iskre koje se pojavljuju kada se NPC puni na stanici.

Stvaranje komponenti stanice se obavlja u konstruktoru klase stanice. Funkcija CreateDefaultSubobject() stvara komponentu primljenog tipa unutar *Actora*. Potrebno je dodati *UStaticMeshComponent* za *Mesh*, *USphereComponent* za kuglu za koliziju i *UParticleSystemComponent* za dodavanja efekta iskri. Za *USphereComponent* je potrebno postaviti radius kugle, dok je za *UParticleSystemComponent* potrebno postaviti vidljivost na FALSE kako se iskre ne bi prikazale prije nego što su potrebne. *USphereComponent* i *UParticleSystemComponent* je potrebno priljepiti za *UMeshComponent* pomoću SetupAttachment() funkcije. Unutar Unreal Enginea je potrebno u *Mesh* komponenat dodati željeni 3D model koji će predstavljati stanicu u projektu i u *Particle System* dodati željeni efekt (iskre).

Kod za stvaranje komponenti stanice je prikazan u primjeru 19.

```
AMedicStation::AMedicStation()
{
    // Set this actor to call Tick() every frame.  You can turn this off to improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

    // Create static mesh component
    StationMesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("StationMesh"));
    RootComponent = StationMesh;

    // Create collision sphere
    CollisionSphere = CreateDefaultSubobject<USphereComponent>(TEXT("CollisionSphere"));
    CollisionSphere->SetupAttachment(RootComponent);
    CollisionSphere->SetSphereRadius(100.f);

    // Create particle system
    static ConstructorHelpers::FObjectFinder<UParticleSystem> Effect(TEXT("ParticleSystem'/Game/StarterContent/Particles/P_Sparks.P_Sparks'"));
    Sparks = CreateDefaultSubobject<UParticleSystemComponent>(TEXT("ParticleSystem"));
    Sparks->SetTemplate(Effect.Object);
    Sparks->SetupAttachment(RootComponent);
    Sparks->SetVisibility(false);
}
```

**Primjer 19.** Stvaranje svih komponenti stanice za punjenje

Stanica treba provjeravati da li se NPC nalazi unutar kugle za koliziju i prikazati iskre dok se NPC puni. Pošto se je ovu provjeru potrebno obavljati dok projekt radi, kod je potrebno staviti unutar Tick() funkcije. Provjera se postiže tako da se uzimaju svi *Actori* koji preklapaju kuglu za koliziju i spremaju se u listu *Actora*. Zatim se *Actori* provjeravaju da li pripadaju klasi NPC-a, te ako pripadaju, da li imaju puno spremište. Ako spremište NPC-a nije puno, vidljivost iskri se postavlja na TRUE kao što je prikazano na primjeru 20.

```cpp
void AMedicStation::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

    // Get all overlapping Actors and store them in an array
    TArray<AActor*> CollectedActors;
    CollisionSphere->GetOverlappingActors(CollectedActors);

    //For each collected actor
    for (int32 iCollected = 0; iCollected < CollectedActors.Num(); ++iCollected)
    {
        // Cast collected actor to AMedic_NPC
        AMedic_NPC* const CollectedMedic = Cast<AMedic_NPC>(CollectedActors[iCollected]);
        // Check if the cast was successful and Medic is not at Max stored health
        if (CollectedMedic && (CollectedMedic->GetStoredHealth() < CollectedMedic->GetMaxStoredHealth()))
        {
            // Show sparks
            Sparks->SetVisibility(true);
        }
        else
        {
            // Don't show sparks
            Sparks->SetVisibility(false);
        }
    }
}
```

**Primjer 20.** Funkcija za prikazivanje efekta iskri kada se NPC puni

### 3.4.3. Postavljanje spremišta NPC-a

Unutar C++ klase NPC-a je potrebno stvoriti UPROPERTY varijable za trenutnu energiju, maksimalnu količinu energije i razine energije ispod koje NPC ide na punjenje, te postaviti početne vrijednosti tim varijablama unutar konstruktora. Varijable su stvorene sa UPROPERTY kako bi se mogle mijenjati unutar Unreal Enginea. Unutar C++ klase NPC je također potrebno stvoriti *Getter* za stvorene vrijednosti i public funkciju pomoću koje će se mijenjati količina trenutne energije NPC-a i koja će se koristiti pri liječenju igrača (tako da NPC troši trenutnu energiju kako liječi igrača) i pri punjenju spremišta NPC-a.

NPC sada sadrži potrebna svojstva za energiju, te je potrebno postaviti provjere pomoću kojih očitava igračevo stanje (tj. da li je igrač ozlijeđen) i stanje spremišta NPC-a (tj. da li NPC ima puno spremište ili je energija NPC-a ispod granice za punjenje). Te se provjere postavljaju u Tick() funkciju unutar *AI Controllera* NPC-a pošto je te provjere potrebno obavljati za cijelo vrijeme pokretanja projekta. Za provjeru stanja igrača, potrebno je dohvatiti igrača u svijetu i dohvatiti njegovu trenutnu i maksimalnu energiju, te ako nisu jednake postaviti *PlayerNeedsHealing* ključ na TRUE, inače se ključ postavlja na FALSE. Za NPC je potrebno preko GetPawn() funkcije od *AI Controller*a dohvatiti NPC i provjeriti njegovu trenutnu i maksimalnu spremljenu energiju kao i granicu za punjenje. Ako je trenutna energija ispod granice, postavlja se *LowCharge* ključ na TRUE, a ako je trenutna energija jednaka maksimalnoj spremljenoj energiji *LowCharge* ključ se postavlja na FALSE.

*Behavior Tree* sadrži dvije grane: prva grana se obavlja kada spremište energije NPC-a padne ispod granice, te potom traži najbližu stanicu i miče NPC-a prema njoj, dok druga grana očitava igračevo stanje, te ako je igrač ozlijeđen miče NPC-a prema njemu i pokušava ga liječiti. Za granu za punjenje NPC-eva spremišta je potrebno stvoriti *Sequencenode* koji provjerava *LowCharge* ključ, tj. da li je spremište NPC-a ispod granice za punjenje. Zatim potrebno je potražiti lokaciju najbliže stanice, pomaknuti NPC-a prema njoj i pričekati dok se NPC ne napuni.

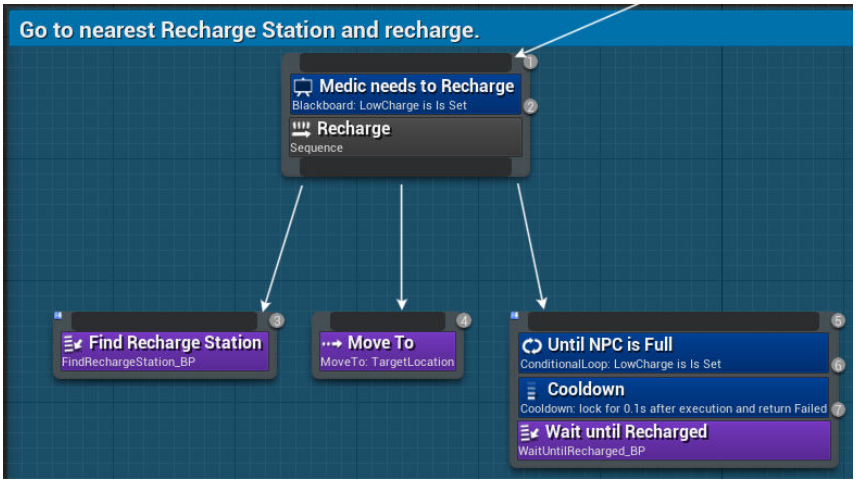
Za traženje najbliže stanice je potrebno stvoriti novi *Behavior Tree* zadatak koji će spremiti sve *Actor*e koji pripadaju klasi stanice za punjenje, te za svakoga provjeriti udaljenost od NPC-a i spremiti koordinate najbliže stanice u *TargetLocation* ključ. Dohvaćanje svih stanica se postiže pomoću funkcije *GetAllActorsOfClass* koja vraća sve *Actor*e koje pripadaju primljenoj klasi. Zatim je potrebno inicijalizirati varijable u kojima će se spremati trenutna stanica u nizu i najbliža pronađena stanica kako bi se udaljenosti stanica mogle usporediti i spremiti najbliža. Sljedeći korak jest prelaženje po listi spremljenih *Actora* i uspoređivati udaljenosti trenutne stanice od NPC-a sa spremljenom udaljenosti. Udaljenosti se dobivaju pomoću funkcije Dist() koja vraća udaljenosti između dva primljena *Actora*. Ako je nova udaljenost manja od spremljene, nova udaljenost se sprema i provjere se nastavljaju do kraja liste, nakon čega se najbliža udaljenost sprema u *TargetLocation* ključ. Ovaj kod je prikazan u primjeru 21.

```
// Get NPC controller
auto const npc_controller = Cast<AMedic_AI>(owner_comp.GetAIOwner());
// Check if cast was successful
if (npc_controller)
{
    // Get NPC
    auto const npc = npc_controller->GetPawn();

    // Get all Recharge Stations and put them in an array
    TArray<AActor*> AllStations;
    UGameplayStatics::GetAllActorsOfClass(GetWorld(), AMedicStation::StaticClass(), AllStations);

    // Initialize Station and NearestStation
    AMedicStation* NearestStation = Cast<AMedicStation>(AllStations[0]);
    AMedicStation* Station = Cast<AMedicStation>(AllStations[0]);

    // If the casts were successful
    if (Station && NearestStation)
    {
        // If Array has more than one element
        if (AllStations.Num() > 1)
        {
            // Distance between Medic and first Station
            float distance = FVector::Dist(npc->GetActorLocation(), Station->GetActorLocation());

            // For each station in array
            for (int32 iStation = 1; iStation < AllStations.Num(); ++iStation)
            {
                // Cast Actor to MedicStation
                Station = Cast<AMedicStation>(AllStations[iStation]);

                // If cast was successful and new distance is less than stored distance
                if (Station && (distance > FVector::Dist(npc->GetActorLocation(), Station->GetActorLocation())))
                {
                    // Store new closest distance
                    distance = FVector::Dist(npc->GetActorLocation(), Station->GetActorLocation());
                    // Store new nearest Station
                    NearestStation = Station;
                }
            }
        }
        // Set blackboard value for target location
        npc_controller->GetBlackboard()->SetValueAsVector(bb_keys::target_location, NearestStation->GetActorLocation());

        // Finish task with success
        FinishLatentTask(owner_comp, EBTNodeResult::Succeeded);
        return EBTNodeResult::Succeeded;
```

**Primjer 21.** Zadatak za traženje najbliže stanice za punjenje

Sljedeći zadatak je puniti NPC-evu spremljenu energiju sve dok nije napunjen. Potrebno je dohvatiti NPC-jevu klasu i pozvati UpdateStoredHealth() funkciju u koju se dodavje željena vrijednost za koju će se spremište napuniti. Pošto sada NPC nije prazan, potrebno je ostaviti NoCharge ključ na FALSE. Sljedeće je potrebno provjeriti da li je NPC pun, te ako jest postavlja se *LowCharge* ključ na FALSE. Time se osigurava da će NPC ostati na stanici sve dok se potpuno napuni, pošto ova grana *Behavior Treea* ima najveći prioritet, a *LowCharge* ključ (koji pokreće ovu granu) je FALSE za cijelo vrijeme punjenja NPC-a, te se grana za liječenje igrača neće pokrenuti ako je igrač ozlijeđen dok se NPC još puni.

Unutar *Behavior Tree* grane je potrebno prvo postaviti zadatak za pronalaženje najbliže stanice i *MoveTo* zadatak kako bi NPC krenuo prema stanici. Sljedeće je potrebno postaviti zadatak za punjenje spremišta u kojem je potrebno dodati *ConditionalLoopDecorator* koji će ponavljati zadatak dok je *Blackboard* ključ postavljen (u ovom primjeru taj ključ je *LowCharge*). Time će se NPC puniti dok ne bude potpuno napunjen. Ova *Behavior Tree* grana je prikazana u primjeru 22.




**Primjer 22.** Behavior Tree grana zadužena za punjenje NPC-a

Kako bi NPC mogao liječiti igrača potrebno je stvoriti novi *Sequencenode* i postaviti dva *Decoratora*: za provjeru *PlayerNeedsHealing* ključa i za provjeru *HasNoCharge* ključa na kojeg je potrebno postaviti *Self* u *Observer Aborts* kako bi prekinuo liječenje igrača ako se spremište isprazni. Time će NPC pokušati izlječiti igrača samo ako je igrač ozlijeđen i ako spremište energije nije prazno. Pošto NPC treba pratiti igrača i liječiti ga samo ako je dovoljno blizu, potrebno je u *Sequencenode* postaviti *IsPlayerInRangeService* koji je napravljen tijekom izrade *Guard* NPC-a kako bi se provjerilo da li je igrač dovoljno blizu NPC-a.

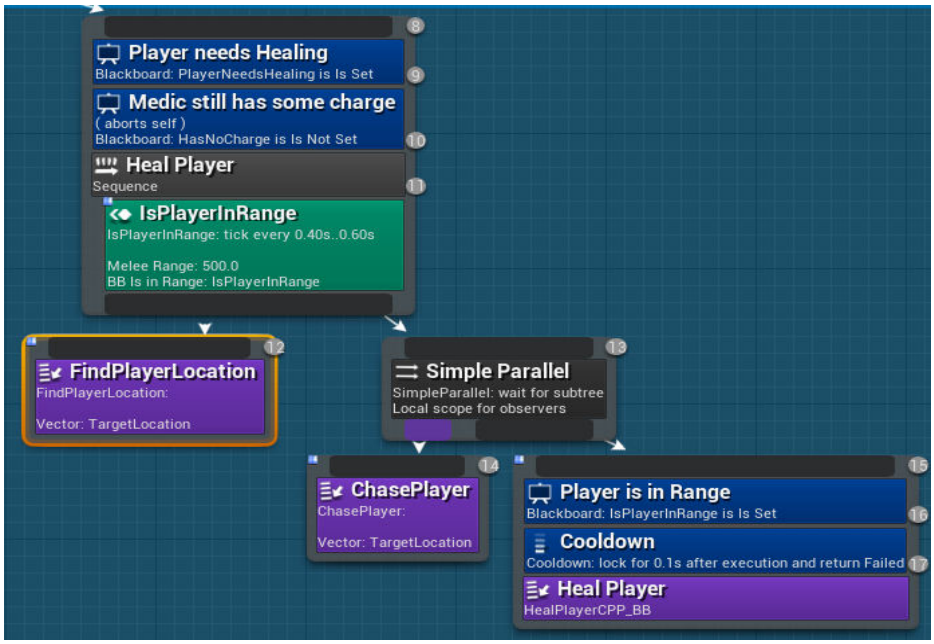
Jedini zadatak koji je potrebno stvoriti onaj jest za liječenje igrača. Potrebno je dohvatiti igrača u svijetu i pozvati UpdatePlayerHealth koja ažurira igračevu energiju i u parametar funkcije dodati željenu vrijednost za koju se želi izliječiti igrača, te pozvati UpdateStoredHealth() od NPC-a

kako bi se od spremišta oduzela ista vrijednost. Zatim se provjerava da li je igrač potpuno izliječen (pri čemu se postavlja *PlayerNeedsHealing* ključ na FALSE) i da li je spremište NPC-a prazno (*HasNoCharge* se postavlja na TRUE). Ove provjere su nužne kako NPC ne bi pokušavao nastaviti liječiti igrača ako je igrač već potpuno izliječen ili ako je NPC prazan. Ovaj kod je prikazan uprimjeru 23.

```
// Get NPC and it's controller
AMedic_AI* npc_controller = Cast<AMedic_AI>(owner_comp.GetAIOwner());
AMedic_NPC* npc = Cast<AMedic_NPC>(npc_controller->GetPawn());

// Get player character
ADiplomskiProjectCharacter* player = Cast<ADiplomskiProjectCharacter>(UGameplayStatics::GetPlayerCharacter(GetWorld(), 0));

// Check casts
if (npc_controller && npc && player)
{
    // Heal player
    player->UpdatePlayerHealth(npc->GetHealAmount());
    // Lower medic's Stored Health
    npc->UpdateStoredHealth((-1) * npc->GetHealAmount());

    // If Player is at full health
    if (player->GetPlayerHealth() == player->GetMaxHealth())
    {
        // Stop healing Player
        npc_controller->GetBlackboard()->SetValueAsBool(bb_keys::player_needs_healing, false);
    }

    // If Medic has no charge left
    if (npc->GetStoredHealth() == 0)
    {
        npc_controller->GetBlackboard()->SetValueAsBool(bb_keys::no_charge, true);
    }

    // Finish task with success
    FinishLatentTask(owner_comp, EBTNodeResult::Succeeded);
    return EBTNodeResult::Succeeded;
}
```

**Primjer 23.** Zadatak za liječenje igrača

Kada NPC otkrije da je igrač ozlijeđen treba prvo pronaći igračevu lokaciju pomoću FindPlayerLocation zadatka i pratiti ga pomoću ChasePlayer zadatka (oba zadatka su već stvorena pri izradi *Guard* NPC-a). ChasePlayer je potrebno postaviti pod lijevom stranom SimplePallarel *nodea* kako bi istovremeno pratio igrača i pokušao ga izlječiti, dok na desnu stranu *nodea* je potrebno postaviti HealPlayer zadatak. Time se završava izrada *Behavior Tree* grane zadužene za liječenje igrača koja je prikazana na primjeru 24.




**Primjer 24.** Behavior Tree grana za liječenje igrača

### 3.4.4. Pregled *Healer* NPC-a

*Healer* NPC služi kako bi se prikazala izrada NPC-a koja pomaže igraču. Pošto igrač može gubiti energiju (npr. ako igrača napadne *Guard* NPC, onda će igrač izgubiti određeni dio energije), *Healer* NPC služi kao način da igrač obnovi izgubljenu energiju. NPC-jeva mogućnost pronalaženja najbliže stanice za punjenje služi za prikazivanje načina spremanja svih *Actora* koji pripadaju određenoj klasi u jedan niz (u ovom slučaju spremanja svih *Actora* koji pripadaju klasi stanice za punjenje i pretraživanje tog niza za najbližu stanicu). Dodatna prednost pronalaženja najbliže stanice jest u slučaju kada je mapa na kojoj se NPC i igrač nalaze velika, stanice se mogu postaviti na više lokacija, što omogućava NPC-u brzu obnovu potrošene energije i brzo liječenje igrača.

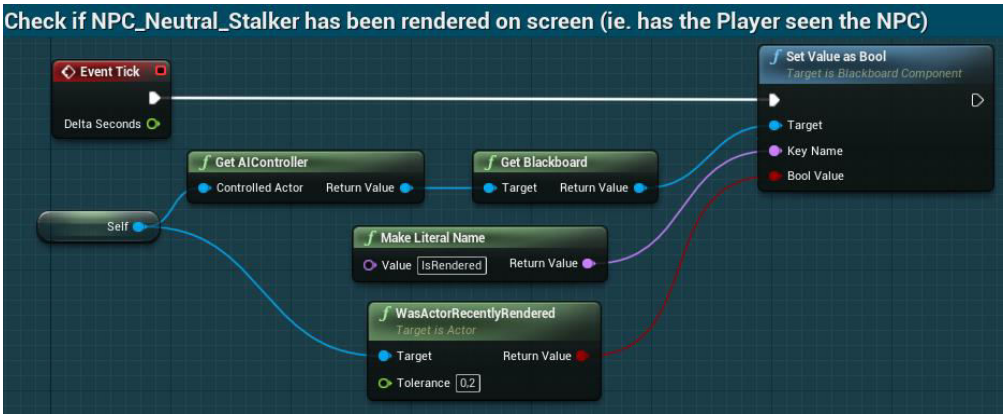
## 3.5. Stalker NPC

Stalker je jednostavan NPC koji prati igrača kada igrač ne gleda prema njemu, a ako igrač gleda prema NPC-u onda se zamrzne. NPC sadrži samo dva *Blueprint* ključa:

- ***TargetLocation***- koristi se za spremanje koordinata prema kojima će se NPC micati.
- ***IsRendered***- označava da li igrač može vidjeti NPC-a, tj. da li je NPC trenutno prikazan na monitoru.

Provjera da li igrač gleda prema NPC-u se postiže u *Blueprintu* od NPC-a pomoću ugrađene *Blueprint* funkcije "*WasActorRecentlyRendered*" koja provjerava da li je primljeni *Actor* bio

prikazan na ekranu unutar određenog vremenskog intervala. Ako jest (tj. igrač je vidio *Actora* unutar tog intervala) vraća TRUE, inače FALSE. Ako je vremenski interval mal (oko 0,2 sekundi) onda se funkcija može koristiti kako bi se otkrilo da li igrač trenutno vidi *Actora* što je potrebno za normalno funkcioniranje NPC-a tako da se vraćena Bool vrijednost od funkcije spremi u *IsRendered* ključ. *Blueprint* NPC-a je prikazan u primjeru 25.




**Primjer 25.** Blueprint za provjeru da li je NPC iscrtan na ekranu

NPC-u su potrebna samo dva zadatka: pronalaženje lokacije blizu igrača i micanje do te lokacije. Iako je zadatak za pronalaženje igračeve lokacije već stvoren, potrebno je napraviti novi zadatak koji će pronaći lokaciju koja je na određenoj udaljenosti od igrača ako nije poželjno da se NPC zabije u igrača. Udaljenost na koju će NPC biti udaljen od igrača je spremljena u UPROPERTY varijabli unutar C++ klase NPC-a.

Kako bi se pronašla odgovarajuća lokacija između NPC-a i igrača potrebno je stvoriti zadatak unutar kojega se uzima trenutna lokacija NPC-a i igrača, te se te vrijednosti oduzmu i konačna vrijednost se sprema u varijablu delta. Od delta varijable je potrebno dobiti normalu koristeći funkciju GetSafeNormal() koja vraća normalu primljenog vektora samo ako je to sigurno napraviti ovisno o duljini. (6) Zatim se ta normala zbraja sa igračevom lokacijom i množi sa dohvaćenom udaljenosti na kojoj NPC prati igrača. Ovim postupkom se pronalazi lokacija na određenoj udaljenosti od igrača prema kojoj će NPC ići. Kod je prikazan u primjeru 26.

```
// Get NPC, NPC's location and the AI controller
auto const npc_controller = Cast<AStalker_AI>(owner_comp.GetAIOwner());
if (npc_controller)
{
    auto const npc = npc_controller->GetPawn();
    auto const npc_location = npc->GetActorLocation();

    // Cast npc to stalker
    AStalker_NPC* stalker = Cast<AStalker_NPC>(npc);
    // Get Player character
    ADiplomskiProjectCharacter* player = Cast<ADiplomskiProjectCharacter>(UGameplayStatics::GetPlayerCharacter(GetWorld(), 0));

    // Check if casts were successful
    if (player && stalker)
    {
        // Find Player location
        FVector const player_location = player->GetActorLocation();

        // Find location near Player
        FVector delta = npc_location - player_location;
        delta = delta.GetSafeNormal();
        FVector target_location = stalker->GetFollowDistance() * delta + player_location;

        // Set target location and finish task
        npc_controller->GetBlackboard()->SetValueAsVector(bb_keys::target_location, target_location);
        return EBTNodeResult::Succeeded;
    }
    return EBTNodeResult::Failed;
}
return EBTNodeResult::Failed;
```

**Primjer 26.** Zadatak za pronalaženje lokacije blizu igrača

U *Behavior Tree* od NPC-a potrebno je napraviti *Sequencenode* koji sadrži *Decorator* za provjeru *IsRendered* ključa, te od *Sequencenodea* stvoriti zadatak za pronalaženje igračeve lokacije i *MoveTo* zadatak. NPC će pri pokretanju projekta pronaći lokaciju blizu igrača i krenuti prema njoj, a ako igrač gleda prema njemu prekinut će izvođenje *Behavior Treea* i zamrznuti se.

### 3.5.1. Pregled *Stalker* NPC-a

*Stalker* je veoma jednostavan NPC koji prikazuje kako igrač može utjecati na NPC. Dok igrač ne vidi NPC-a, NPC se pomiče prema igraču sve dok ne dođe do određene udaljenosti od igrača. Ta je udaljenost postavljena kako bi se spriječilo zabijanje NPC-a u igrača i izbjegle moguće greške tijekom izvođenja projekta. Ako igrač vidi NPC-a, NPC se zamrzne na mjestu. Pomoću ove funkcionalnosti moguće je stvoriti prepreke koje igrač mora lukavo riješiti (tako da odmakne NPC-a od nekog mjesta ili ga privuče prema određenoj lokaciji).

## 4. *Enviroment Query System*

EQS (eng. *Enviroment Query System* - sustav za ispitivanje okoliša) je dio sustava umjetne inteligencije u Unreal Engineu koji skuplja podatke iz okoliša mape i prema tim podatcima provodi odluke. EQS se sastoji od generatora, testa i konteksta. (7)

Generatori stvaraju nevidljive lokacije ili *Actor*e unutar mape projekta. Nad tim objektima se provode testovi te se u *Behavior Tree* vraća jedna ili skup objekata koji sadrže najbolji rezultat.

Testovi su skupovi pravila prema kojima se provode provjere nad generatorima kako bi se pronašao objekt koji najbolje zadovoljava te uvjete. Primjer testa: udaljenost *Actora* A od generiranih objekata i da li *Actor* A može vidjeti te objekte. Vraća se objekt koji je najbliži i unutar vida *Actora* A.

Kontekst se može smatrati kao referentni okvir za testove i generatore, tj. sa koje perspektive će se provoditi testovi. Koristeći gornji primjer, kontekst je bio *Actor* A. Ako se kontekst promjeni na igrača, onda će se vratiti objekt koji je najbliži i unutar vida igrača.

Pomoću EQS-a je moguće stvoriti sljedećeg NPC-a koji će pronaći lokaciju na mapi koja je skrivena od igračeva vida, a istovremeno blizu NPC-a i daleko od igrača te će poći prema toj lokaciji.

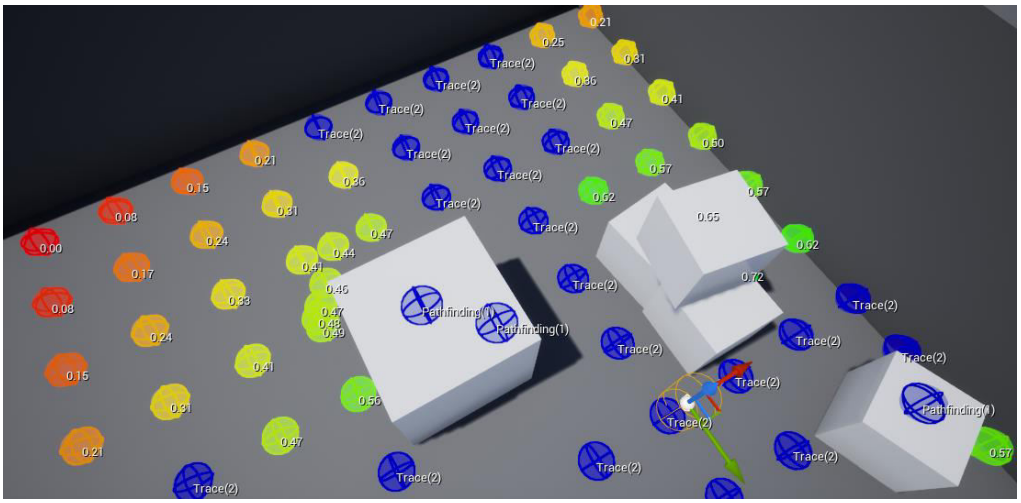
NPC se stvara kao i u prijašnjim poglavljima pomoću *Actora*, *AI Controllera*, *Behavior Treea* i *Blackboarda*. Zatim je potrebno stvoriti*Enviroment QuearyBlueprint* klasu unutar koje se stvaraju željene vrste generatora, postavljaju testovi koji će ti generatori provoditi i postavljaju konteksti. *Enviroment Queary* je sličan *Behavior Treeu*, tj. stvaranje generatora i testova u *Enviroment Quearyu* se obavlja istim postupkom kao i stvaranje zadataka i *Decoratora* unutar *Behavior Treea*. Ovaj *Enviroment Queary* će provjeravati da li NPC, koji će biti postavljen kao kontekst (tj. *Context Querier*) u opcijama na desnom prozoru, može vidjeti igrača i ako može vidjeti igrača, sprema objekt igrača u *Blackboard* ključ i uspješno završava izvedbu.

Stvara se generator tipa *All Actors of Class* i na desnom prozoru pod *Searched Actor Class* postaviti *FirstPersonCharacter* što stvara generator na svim *Actorima* na mapi koji pripadaju *FirstPersonCharacter* klasi (pošto je igrač jedini *Actor* koji pripada toj klasi, na njemu će se stvoriti generator). Na stvorenom generatoru se test dodava desnim klikom i odabirom *Add Test*, te se odabire *Trace* tip testa. *Trace* provjerava da li se može povući linija od konteksta prema generatoru bez prekidanja čime se provjerava da li je moguće vidjeti kontekst/generator.

Sljedeće je potrebno stvoriti *EnvQueryContext_BlueprintBase* klasu koja će vratiti klasu od *FirstPersonCharacter* kao kontekst kako bi se mogao koristiti u *Enviroment QueryBlueprintu*, a

to se postiže dodavajući "*Get All Actors of Class*" *node* između "*Provide Actors set*" *nodea* i *ReturnNode* koji su već stvoreni i postavljanje *ActorClass* na *FirstPersonCharacter*. Time se dohvaćaju svi *Actori* na mapi koji pripadaju klasi *FirstPersonCharacter* i predaju se *Enviroment Queryu* kao kontekst.

Nakon stvaranja funkcije koja provjerava da li NPC može vidjeti igrača i konteksta koji vraća *Actor*e koji pripadaju *FirstPersonCharacter* klasi, tj. klasi igrača, potrebno je stvoriti *Enviroment QueryBlueprint* za pronalaženje lokacije oko NPC-a koju igrač ne može vidjeti kako bi se NPC mogao sakriti. U novom *Enviroment QueryBlueprintu* se stvara generator vrste *Points: Grid*, koji na mapi oko NPC-a (koji je u ovom slučaju *Querier*, tj. *Actor* koj postavlja upite) generira matricu točaka nad kojima će se provoditi testovi. Ovisno o uspješnosti testa, svakoj točci se dodaje rezultat, te se vraća točka sa najboljim rezultatom. Na primjeru 27. je prikazana generirana matrica točaka na mapi. Svaka točka sadrži broj i odgovarajuću boju koji označavaju koliko uspješno su testovi provedeni nad točkom. Zelene točke sadrže bolji rezultat, crvene lošiji, a tamno plave točke označuju da te lokacije ne mogu zadovoljiti neki upit (npr. igrač može vidjeti te točke te ne mogu ispuniti taj upit ili se nalaze na lokaciji na koju NPC ne može doći).




**Primjer 27.** Generirana *Points: Grid* matrica na mapi

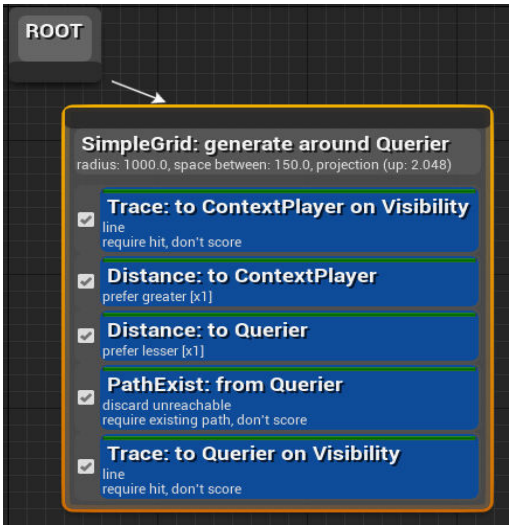
Prvi test koji je potrebno stvoriti jest *Trace* kako bi se pronašla točka koju igrač ne može vidjeti. U *Details* prozoru od *Trace* testa je potrebno postaviti kontekst na prijašnje stvoreni kontekst koji vraća sve *Actor*e klase *FirstPersonCharacter*. Zatim *Test Purpose* se postavlja na *Filter Only* koji, umjesto da daje ocjene točkama ovisno da li igrač može vidjeti te točke ili ne, odbacuje tj. filtrira sve točke koje igrač može vidjeti, te se one ne rangiraju. Bez ove opcije, NPC bi u nekim slučajevima pošao prema točci koju igrač može vidjeti jer bi ta točka imala visok rezultat zbog ostalih testova.

Sljedeća dva testa su vrste *Distnace*. Prvi test traži točke koje su daleko od igrača te je potrebno postaviti kontekst igrača, a drugi test traži točke blizu NPC-a i kontekst se postavlja na *EnvQueryContext_Querier*. Kako bi testovi pravilno ocjenjivali točke, tj. prvi test treba dati bolji rezultat točkama koje su daleko od igrača, a drugi test bolji rezultat točkama blizu NPC-a, potrebno je *Scoring Equation* postaviti na *Linear* za prvi test, a za drugi *Inverse Linear*. Bez prvog testa NPC bi trčao prema točci direktno iza igrača pošto igrač tu točku ne može vidjeti, dok bez drugog testa igrač bi trčao prema najudaljenijoj točki matrice i ne bi se efektivno skrivao.

Osim točaka koje igrač može vidjeti, također je potrebno odbaciti točke do kojih NPC ne može doći. Test *PathExist* provjerava da li postoji putanja od generirane točke do konteksta kojeg treba postaviti na NPC, te se pomoću *Filter Only* opcije odbacuju sve točke do kojih NPC ne može doći kako ne bi došlo do grešaka tijekom izvođenju projekta (npr. NPC bi trčao u zid pokušavajući doći do točke koja se nalazi na zidu.)

Zadnji test je *Trace* koji provjerava da li NPC ne može vidjeti točku kako bi se bolje skrivao. Ovaj test se stvara istim postupkom kao i prvi Trace test za igrača, s jedinom razlikom u kontekstu kojeg je potrebno postaviti na *EnvQueryContext_Querier*. Iako ovaj test nije nužan, u određenim situacijama osigurava da će NPC pronaći najbolju lokaciju za skrivanje.

Krajnji izgled *Enviroment QuearyBlueprinta* je prikazan na primjeru 28.




**Primjer 28.** Enviroment Queary Blueprint za skrivanje od igrača

Unutar *Behavior Treea* od NPC potrebno je dodati stvorene EQS zadatke, s tim da se prvo postavlja pronalaženje igrača, zatim se postavlja pronalaženje lokacije za skrivanje čije se koordinate spremaju u *TargetLocationBlackboard* ključ, a na kraju se dodava *MoveTo* zadatak kako bi NPC došao do te točke.

### 4.2. Opis NPC-a stvorenog pomoću EQS-a

Cilj ovog NPC-a jest prikaz eksperimentalnog EQS-a za izradu umjetne inteligencije koja donosi odluke ovisno o NPC-jevoj okolini. U ovom poglavlju prikazana je izrada NPC-a koji pretražuje pomoću postavljenih testova i ispitiva svoju okolicu za najbolje mjesto gdje se može sakriti od igrača, te, ovisno o rezultatu koji svaka ispitivana lokacija vrati, NPC krene prema najbolje skrivenom mjestu. Dodavanjem novih testova moguće je obrnuti funkcionalnost tako da se igrač skriva, a NPC pretražuje sva mjesta koja su najbolje skrivena od tog NPC-a i tako traži igračevu lokaciju u kojoj se skrio. Pomoću EQS-a je moguće stvoriti napredne funkcionalnosti umjetne inteligencije koja omogućuje simuliranje funkcionalnosti iz stvarnog života.

# 5. Zaključak

Umjetna inteligencija u video igrama služi za izradu računalno upravljanih likova koji simuliraju ljudsko ponašanje. Time se stvaraju prepreke koje igrač mora riješiti. U Unreal Engineu je moguće stvoriti računalno upravljane likove pomoću stabla ponašanja ili eksperimentalnog sustava za ispitivanje okoliša- EQS. Tijekom izvođenja stabla ponašanja, stanja projekta se spremajuu *Blackboard* ključeve te se ti ključevi koriste pri odlučivanju koji će se zadatak obaviti. EQS sustav koristi generatore, točke postavljene na mapi projekta i testove koji se provode nad generatorima. Generatorima se pridružuje ocjena,u ovisnosti koliko su uspješno prošli testove, te se generator sa najboljomocjenom vraća iposljedično se nastavlja s izvedbom ostalih zadataka

U ovom projektu samuspiješno razvio 5 NPC-jeva s ciljem prikaza postupka same izradete demonstracije mogućnosti stabla ponašanja i EQS-a. Pomoću *Wanderer* NPC-adan je prikazfunkcija stabla ponašanja i *Blueprint* sustava za vizualno skriptiranje. *Guard* NPC-jem je stvorena umjetna inteligencija koja prati određenu putanju, registrira osjetila vida(kada vidi igrača, počne ga loviti) i sluha (igrač može skrenuti pažnju NPC-u koji zatim privremeno napusti patroliranje). Unutar*Healer* NPC-a je stvorena umjetna inteligencija koja očitava igračevo stanje te ga liječi ako je ozlijeđen, a ako očita da mu je spremište energije prazno, traži najbližu stanicu za punjenje. Sa *Stalker* NPC-om je istraženo funkcioniranje IsRendered() metode koja očitava da li je NPC bio prikazan na monitoru te, ukoliko je uvjet istinit, isti NPC se zamrzne dok god se nalazi u igračevomvidnom polju, a u protivnom NPC prati igrača dok ne dođe do određene udaljenosti. Zadnji opisani NPCje izrađen pomoću EQS-a.On ispituje svoju okolicu kako bi pronašao najbolje mjesto gdje se može sakriti od igrača te je na taj način dobivena inteligencija koja može tražiti mjesto na kojem se najbolje može sakriti od igrača.

# Literatura


1. Unreal Engine. *Wikipedia.* [Mrežno] 2021. [Citirano: 3. lipanj 2021.] https://en.wikipedia.org/wiki/Unreal_Engine#Further_reading.

2. **Epic Games.** Unreal Engine 3. *Unreal Tehnology.* [Mrežno] Epic Games, 3. svibanj 2004. [Citirano: 7. lipanj 2021.] http://web.archive.org/web/20040607133325/http://www.unrealtechnology.com/html/technology/ue30.shtml.

3. —. Introduction to Blueprints. *Unreal Engine Documentation.* [Mrežno] Epic Games. [Citirano: 10. lipanj 2021.] https://docs.unrealengine.com/4.26/en-US/ProgrammingAndScripting/Blueprints/GettingStarted/.

4. Behavior tree (artificial intelligence, robotics and control). *Wikipedia.* [Mrežno] 10. veljača 2021. [Citirano: 19. lipanj 2021.] https://en.wikipedia.org/wiki/Behavior_tree_(artificial_intelligence,_robotics_and_control).

5. **Epic Games.** Behavior Tree Node Reference: Services. *Unreal Engine 4.26 Documentation.* [Mrežno] Epic Games. [Citirano: 5. srpanj 2021.] https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/ArtificialIntelligence/BehaviorTrees/BehaviorTreeNodeReference/BehaviorTreeNodeReferenceServices/.

6. —. FVector::GetSafeNormal. *Unreal Engine Documentation.* [Mrežno] Epic Games. [Citirano: 30. svibanj 2021.] https://docs.unrealengine.com/4.26/en-US/API/Runtime/Core/Math/FVector/GetSafeNormal/.

7. —. Enviroment Query System. *Unreal Engine Documentation.* [Mrežno] Epic Games. [Citirano: 2021. lipanj 2021.] https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/ArtificialIntelligence/EQS/.

8. **Laley, Ryan.** Unreal Engine 4 Tutorial - AI. *Youtube.* [Mrežno] 17. Siječanj 2019. [Citirano: 5. svibanj 2021.] https://www.youtube.com/watch?v=zNJEvAGiw7w&list=PL4G2bSPE_8ukuajpXPlAE47Yez7EAyKMu.

9. **MrCxx.** UE4 C++ AI Tutorial Series. *Youtube.* [Mrežno] 16. Prosinac 2019. [Citirano: 6. svibanj 2021.] https://www.youtube.com/watch?v=u8DNygzljXY&list=PLWUvrI0mg8VLrvq3n2iu45lhCo69uluDK.

# Prilozi

# IZJAVA

Izjavljujem pod punom moralnom odgovornošću da sam diplomski rad izradio samostalno, isključivo znanjem stečenim na studijima Sveučilišta u Dubrovniku, služeći se navedenim izvorima podataka i uz stručno vodstvo mentora prof. dr. sc. Vedran Batoš i komentora dipl. ing. Ivan Grbavac kojima se još jednom srdačno zahvaljujem.

Karlo Arbanas