

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
ANALIZA I RAZVOJ PROGRAMA

AIRBENDER VR
UNREAL ENGINE
Naputci za izradu

Dorian Čajko Ivan Huzjak Denis Jocković Filip Novački Luka Štefanić

VARAŽDIN

SADRŽAJ

1	UVOD U INDUSTRIJU, STVARANJE IGARA TE OKRUŽENJE	8
1.1	DIZAJN VIDEO IGARA	8
1.2	VR	10
1.3	INSTALACIJA I KORIŠTENJE OKRUŽENJA	10
1.3.1	INSTALACIJA OCULUS SOFTVERA	10
1.3.2	INSTALACIJA UNREAL ENGINEA	11
1.3.3	KORIŠTENJE UNREAL ENGINEA	11
1.4	KREIRANJE PROJEKTA	13
1.5	KONFIGURIRANJE PROJEKTA	15
1.6	ZAKLJUČAK	16
2	GLAVNI LIK	17
2.1	MOTIONCONTROLLERPAWN	17
2.1.1	POSTAVLJANJE VISINE IGRAČA TE POSTAVKE VR UREDAJA	17
2.1.2	STVARANJE I POVEZIVANJE KONTROLERA	18
2.1.3	PROCES TELEPORTACIJE	19
2.1.4	DOBIVANJE LOKACIJE I ROTACIJE ZA MJESTO TELEPORTACIJE	20
2.1.5	RUKOVANJE KONTROLERIMA	22
2.1.6	POSTAVLJANJE ROTACIJE MJESTA TELEPORTACIJE	23
2.1.7	AKTIVIRANJE VIZUALIZACIJE MJESTA TELEPORTACIJE	24
2.1.8	DEAKTIVIRANJE VIZUALIZACIJE MJESTA TELEPORTACIJE	25
2.2	BP_MOTIONCONTROLLER	26
2.2.1	MIJENJANJE SKALIRANJA	26
2.2.2	VIBRACIJE	27
2.2.3	HVATANJE RUKOM	28
2.2.4	STVARANJE LÜKA ZA VIZUALIZACIJU	29
2.2.5	DOBIVANJE NAVIGACIJSKOG SUSTAVA	32
2.2.6	TRAŽENJE MJESTA ZA STAJANJE	33
2.2.7	ŠTO AKO SE NE MOŽE NAĆI POD?	34
2.3	ZAKLJUČAK	41
3	KRETANJE	42
3.1	ROOMSCALE KRETANJE	44
3.2	MOVEMENT CHECK	45
3.3	<i>In Game Position</i>	50
3.4	ROTACIJA	50
3.5	ZAKLJUČAK	55
4	BILJEŽENJE POKRETA	56
4.1	SFERE KOLIZIJE	56
4.2	OKIDANJE DOGADAJA DODIRA SFERE	60
4.3	UNOŠENJE PODATAKA O DODIRU SFERE U POLJE	61
4.4	PRIPASIVANJE SPOSOBNOSTI	64

4.5	ZAKLJUČAK	69
5	STVARANJE PROJEKtilA	71
5.1	KREIRANJE KLASE PROJEKtilA I NJEGOVE INSTANCE	71
5.2	RAD PROJEKtilA	75
6	STVARANJE KAMENA	78
6.1	ZAKLJUČAK	80
7	NEPRIJATELJI	81
7.1	UMJETNA INTELIGENCIJA	84
7.1.1	OSNOVE STABLA PONAŠANJA	85
7.1.2	PERCEPCIJA NEPRIJATELJA	86
7.1.3	POSTAVLJANJE VARIJABLE ZA GLEDANJE IGRAČA	87
7.1.4	KADA NEPRIJATELJ VIDI IGRAČA	90
7.1.5	KADA NEPRIJATELJ NE VIDI IGRAČA	93
7.1.6	KADA NEPRIJATELJ UMRE	94
7.2	STVARANJE NEPRIJATELJA	94
7.3	SUSTAV ZA ŠTETU	96
7.3.1	FUNKCIJE APPLY DAMAGE	96
7.3.2	DOGADAJ ANYDAMAGE	97
7.3.3	VRSTE NEPRIJATELJA	99
7.4	ZAKLJUČAK	99
8	ZDRAVLJE IGRAČA	100
9	IZBORNik	102
9.1	<i>FLIPFLOP</i> I LOKACIJA IZBORNika	103
9.2	INTERAKCIJA S IZBORNIKOM	104
10	SCOREBOARD API	106
11	SPREMANJE POSTAVKI	111
12	MAPA	116
13	GOTOVI ELEMENTI I ANIMACIJE	118
13.1	ANIMACIJE	120
14	VIZUALNI EFEKTI	123
14.1	RUKA	125
14.2	PROJEKtilI	126
14.3	ENEMY	127
14.4	STVARANJE KAMENA	129
15	ZVUČNI EFEKTI	131

16 PONOVNO POKRETANJE IGRE	139
17 UVOD U IGRU	143
18 KUHANJE, PEČENJE I ZAKLJUČAK	147
18.1 STEAM	147
18.2 EPIC GAMES STORE	149
18.3 OCULUS STORE	149
18.4 PRIKAZ SVIH SPOMENUTIH TRGOVINA	152

UVOD

Ovaj priručnik za programiranje projekta namijenjen je prvenstveno studentima tehničkih područja kao nit vodilja kroz stvaranje jednog projekta. Igra koja se stvara ovim putem inspirirana je popularnom crtanom serijom Avatar, a ime, u originalu *Avatar, the Last Airbender*, vrlo se prikladno poklopio s imenom kolegija, *Analiza i razvoj programa*, iliti skraćeno AIR.

Podrazumijeva se da čitatelj ovog priručnika poznaje osnovne koncepte programiranja te se oni ovdje neće u detalje objašnjavati. Naglasak će se staviti na koncepte koji se upotrebljavaju u izradi igre, dakle oni vezani za Unreal Engine, razvoj igara itd.

Glavni tekst sadržavat će objašnjenje postupaka koji se koriste u izradi igre. Moguće je da se koraci malo promijene u određenim verzijama Unreal Engine softvera koji se koriste, no pretpostavlja se da će sve ostati slično jer se ne koriste jako opskurni koncepti.

Na kraju priručnika nalazi se kazalo pojmove kako bi se određeni pojmovi mogao lakše pronaći ukoliko je samo spomenut negdje u tekstu, a nije iz naslova jasno o kojem se pojmu točno radi.

Cijeli projekt može se naći na poveznici <https://github.com/AIR-FOI-HR/AIRBenderVR.git>. Repozitorij se može preuzeti naredbom `git pull poveznica`. Grana na kojoj su određene verzije iz faze projekta je `Example`. `Commit` na kojem se nalazi stanje repozitorija u toj fazi projekta piše na svakoj stranici u zagлавlju.

Dodatna objašnjenja mogu se vidjeti izdvojena sa strane kako bi se dodatno povezalo objašnjeno s drugim konceptima.

1 UVOD U INDUSTRIJU, STVARANJE IGARA TE OKRUŽENJE

Programiranje igara se po mnogočemu razlikuje od programiranja poslovnih aplikacija. Pristup i razmišljanje su drugačiji. Dok je za poslovne aplikacije važno usredotočiti se na podatke, u igrama je važno i misliti na tzv. *delivery*, odnosno kako će korisnik doživjeti aplikaciju. *Game engine* omogućava jednostavno uređivanje aspekata grafike, fizike, objekta i drugo. *Game engine* na taj način olakšava što su svi ti alati međusobno povezani i olakšava stvaranje kohezivne cjeline kod razvijanja softvera, odnosno omogućuje onima koji rade na softveru da se više orijentiraju na sam sadržaj nego na razvijanje tehničkog dijela.

Unreal Engine¹ je jedan od najpopularnijih rješenja za razvijanje igara te ćemo se u ovom priručniku koristiti njime.

Jedna od posebnosti Unreal Enginea je što koristi Unreal Visual Scripting, ili jednostavnije *blueprint*, alat kojim se odnosi između objekata programiraju ne kodom, nego povlačenjem odnosa između objekata koji su reprezentirani vizualno što olakšava predočavanje te dodatnu razinu apstrakcije od programskog jezika C++.

Prije nego što će biti objašnjeni detalji o Unreal Engineu objasnit će se i osnove dizajna video igara (en. *game design*), kako teče proces izrade igara te na što se sve treba pripaziti kod izrade igara.

Kao dodatna motivacija za stvaranje igara je činjenica da se ta industrija u zadnjih osam godina tržišna vrijednost igara udvostručila, a predviđa se da će se u iduće tri godine (2020.–2023.) vrijednost utrostručiti. Broj aktivnih igrača u svijetu raste velikom brzinom te se u tim podatcima prepoznaje perspektiva te industrije. Iz tog je razloga dobro poznavanje ovog sektora, ako ne zbog želje za radom u njoj, barem zbog opće kulture.

1.1 DIZAJN VIDEO IGARA

Dizajn video igara kao proces teško je definirati. Dizajn obuhvaća sve ono što se događa za vrijeme stvaranja igre, dakle počinje idejom i temom, nastavlja razvitkom i na kraju stvaranje verzije igre koja se izdaje i distribuira igračima.

Stvaranje ideje Ideja se razvija na razne načine – može doći kroz razgovor s bliskim osobama, može se razviti kroz *brainstorming*, može doći kroz

¹Unreal Engine je softver koji je već dugo razvijan i već je nekoliko većih verzija ostavio iza sebe. Važno je naglasiti da se ovaj projekt razvijao u Unreal Engineu 4. Dakle, kad god u tekstu piše Unreal Engine ili UE, zapravo se odnosi na Unreal Engine 4. Također je moguće i da se kolokvijalno čuje izraz *Unreal*, gdje se također misli na Unreal Engine, a verzija će ovisiti o kontekstu razgovora.

U Unreal Engineu moguće je u potpunosti programirati u C++, no zbog kompleksnosti jezika i pristupačnosti *blueprinta* uglavnom ćemo se baviti vizualnim skriptiranjem.

bljesak inspiracije ili na neke druge načine. Međutim, za uspjeh igre često je ideja samo preduvjet, ali ne i nužni uvjet. Marketing je uvijek zadužen za velik dio uspjeha svake igre. Iz tog razloga danas je mnogo igara koje su odlično zamišljene, ali nisu uspjele ili nisu popularne jer je marketing slab ili nikakav.

Osnovne funkcionalnosti i mehanike Mehanike igre govore kako će se igra ponašati u određenim trenutcima, odnosno na neke korisničke akcije. Mehanike se uvelike razlikuju između različitih žanrova i to ih često čini specifičnima. Primjeri mehanika su izazovi na *bossovima*, odnosno način na koji se igrač nosi s tim izazovima, način na koji igrač ima interakciju s raznim objektima u igri, kako koristi objekte, kako se objekti ponašaju prema njemu itd.

Upravo su mehanike te o kojima ovisi kvaliteta igre i završnog proizvoda jer preko njih igrač ima odnos s okolinom.

Funkcionalnosti su alati kojima se rješava neka problematika, npr. kretanje glavnog lika, obrana lika od napada, umjetna inteligencija itd.

Skica i razrada igre U ovom dijelu procesa dizajna video igre kreiraju se grube skice buduće igre i donose se razmatranja kako će izgledati pojedini element igre. Skice nisu detaljne, ali nam uvelike olakšavaju daljnji rad u nekom od alata. Tu se razvijaju likovi, razine, moći, priča itd. Priča je vrlo važan element jer je to ono na što se igrači "hvataju", odnosno to je ono što je potrebno da se užive.

Game Design Document Nakon što je uvodni dio napravljen, vrijeme je da se pojedini elementi malo detaljnije razrade. Taj dokument naziva se *Game Design Document*, ili kratko – GDD. To je detaljan dokument koji, između ostalog, sadrži:

- naziv igre
- sažetak igre
- funkcionalnosti
- mehanike
- opis likova
- dizajn razina

U sklopu ovih lekcija neće se raditi GDD, ali za bilo koji ozbiljan projekt dobro je imati taj dokument kao zamjenu za dokumentaciju kako bi se olakšalo snalaženje u projektu i kodu.

Cilj dokumenta je olakšati svima razvoj igre tako da lakše zajedno surađuju. U njemu su opisane sve glavne komponente. GDD nije uvijek dio svakog razvoja igre jer je taj dokument vrlo opsežan. Upravo zbog toga ga manji

razvojni studiji nemaju jer im nije potreban i jer je previše posla, a u velikim studijima je koristan jer svi timovi moraju biti dobro sinkronizirani.

1.2 VR

Igra-projekt kojeg će ovaj priručnik opisati bit će implementiran za VR. VR je skraćenica na engleskom od *virtual reality* što znači da se imitira stvarnost u virtualnom okruženju. Cilj VR-a je korisniku stvoriti osjećaj kao da je u stvarnom svijetu podražujući više osjetila. Glavni uređaj koji je svojevrsno obilježje VR koncepta su naočale, odnosno *headset* koji prekriva cijelo vidno polje i korisniku omogućuje da vidi sliku kao realnost tako da svako oko vidi posebnu sliku. Ovaj projekt koristit će i kontrolere čija je posebnost da otkrivaju položaj igračevih ruku u prostoru, a omogućuju i još neke funkcionalnosti koje se mogu isprogramirati prema potrebi projekta. U nekim igrama to je već iskorišteno za precizno bacanje projektila, predmeta, uzimanje predmeta itd., a u našem projektu to će biti glavni okidači za usmjereno bacanje moći.

VR može vrlo lako učiniti neiskusnog igrača omamljenog, odnosno može osjećati glavobolju, vrtoglavicu i slične simptome ukoliko nije naviknut na virtualnu stvarnost. Postoje mnoge tehnike kako se to može ublažiti. U ovom projektu će se pokušati voditi računa o tome koliko god je moguće.

1.3 INSTALACIJA I KORIŠTENJE OKRUŽENJA

Razvojno okruženje sastoji se od Oculus softvera i Unreal engine editora. Oculus softver služi kao računalni *driver* za Oculus *headset*. Kako bi *headset* mogao komunicirati sa softverom potreban je *Steam VR* koji se instalira unutar Steama.

Postavljanje okruženja odvija se u sljedećim koracima:

1. Instalacija Oculus softvera
2. Instalacija Steama
3. SteamVR
4. Epic Games
5. Instalacija Unreal enginea
6. Pokretanje ugrađenog projekta
7. Uklanjanje eventualnih problema
8. Postavljanje verzioniranja na projekt

1.3.1 INSTALACIJA OCULUS SOFTVERA

Za početak je potrebno doći do Oculusove stranice za preuzimanje softvera (oculus.com/setup/) i preuzeti softver za uređaj na kojem razvijamo. U

našem slučaju razvijamo na Oculus Rift S. Potrebno je *scrollati* dok se ne nađe uređaj i pritiskom na odgovarajući gumb preuzeti izvršnu datoteku.

1.3.2 INSTALACIJA UNREAL ENGINEA

Predkorak za instalaciju Unreal Enginea je instalacija Epic Gamesa. Na stranici [epicgames.com](https://www.epicgames.com) u gornjem desnom kutu nalazi se gumb **Get Epic Games** nakon čega se preuzima instalacijska datoteka. Po instalaciji je potrebno napraviti Epic Games račun i pomoću njega se prijaviti u Epic Games.

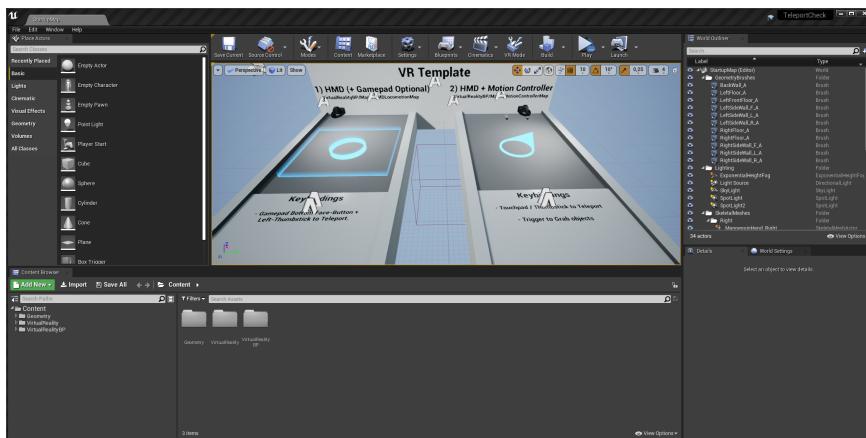
U sklopu Epic Gamesa instalirava se Unreal Engine. U lijevom izborniku potrebno je odabratи Unreal Engine. Kod instalacije treba pripaziti koja se verzija instalirava. Ovaj priručnik pisan je u verziji 4.25.4. Druge verzije moguće bi raditi, no to bi zahtijevalo nešto više angažmana za čitatelja da se pronađu iste funkcionalnosti uz rizik da neke stvari rade drugačije.

Kod odabira licence potrebno je pripaziti koja se odabire. *Publishing license* ona je koja se odabire ukoliko se proizvod namjerava prodati, a *Creators license* ukoliko se namjerava raditi nemonetiziran rad. Studenti su navedeni u obje kategorije jer se *engine* ne mora plaćati ako se koristi za projekt koji još ne stvara profit. Štoviše, Epic Games ne zahtijeva plaćanje sve dok projekt koji je napravljen Unreal Engineom ne donese poduzeću milijun dolara.

Dalje je potrebno registrirati se te slijediti uputstva kod instalacije. Unreal Engine radi na operacijskim sustavima Linux te Windows.

1.3.3 KORIŠTENJE UNREAL ENGINEA

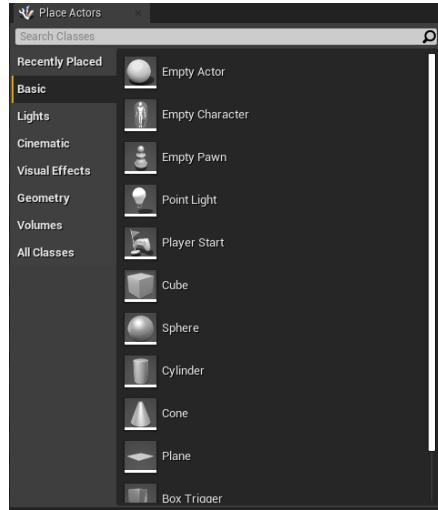
Dijelovi editora koji se nakon otvaranja projekta vide su: *Place Actors*, *Toolbar*, *Viewport*, *Content Browser*, *World Outliner* te *Details panel*.



Slika 1: Prikaz Unreal Enginea nakon otvaranja projekta

Instalacija za Linux ponešto je složenija te je potrebno kompajlirati cijeli projekt. To uzima poprilično vremena i resursa, a sadrži i nešto više koraka koji se ovdje neće opisati jer se orijentiramo na Windows operacijske sustave

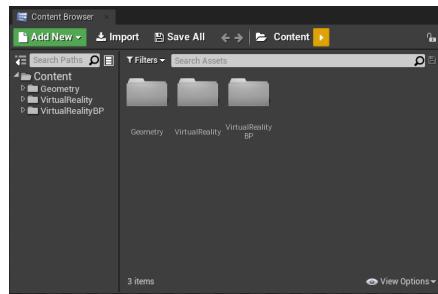
U *Place Actors* (slika 2) dijelu se nalaze *actori* koji se mogu postaviti u *Viewport* jednostavnim *drag'n'dropom*. Nakon kreiranja projekta u *Place Actors* mogu se pronaći zadani *actori* koji se nalaze u svakom projektu.



Slika 2: Prikaz *Place Actors*

Najveći dio *editora* zauzima *Viewport*. U *Viewportu* se nalazi trenutna otvorena razina (*level*, mapa). Svi elementi koji se u njoj nalaze mogu se označiti klikom miša. Označeni element može se pomicati po svim trima osima, može se rotirati po svim trima osima te se može skalirati.

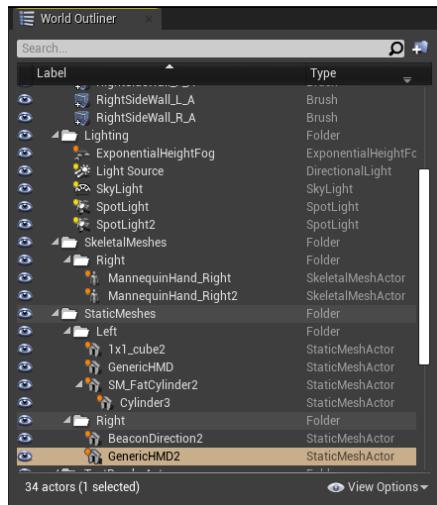
Content browser (slika 3) služi za navigaciju korisnika po raznim mapama i dokumentima. Iz njega se može pristupiti svim dokumentima koji se koriste u projektu.



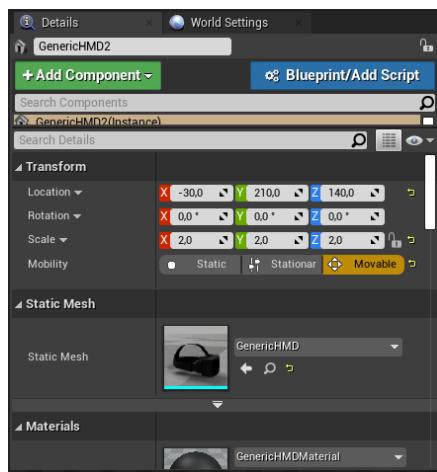
Slika 3: Prikaz *Content Browsera*

World Outliner (slika 4) je jednostavan element editora koji se nalazi u gornjem desnom kutu. U njemu se mogu pronaći svi *actori* koji su postavljeni u trenutno izabranoj razini. Ukoliko se odabere neki actor iz *World Outlinera*, on će automatski biti odabran i u *Viewportu*. Taj element će se odmah i prikazati na sredini ekrana te će se pogled skalirati tako da cijeli objekt stane.

Svaki *actor* koji se nalazi u *Viewportu* i *World Outlineru* ima svoj *Details*

Slika 4: Prikaz *World Outliner*

panel (slika 6. U ovom panelu se mogu se provjeriti i mijenjati razni podaci poput lokacije actora, rotacije, veličine.

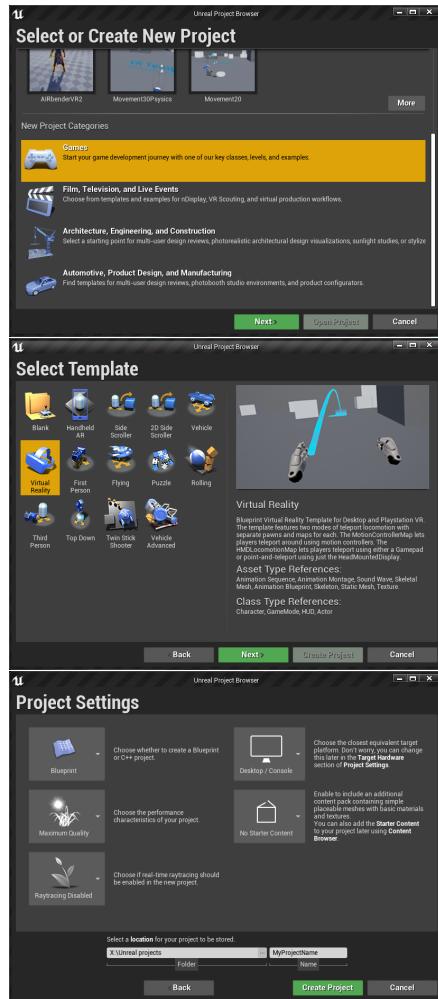
Slika 5: Prikaz *Details Panela*

1.4 KREIRANJE PROJEKTA

Kako bi testirali radi li cijelo okruženje ispravno učitati ćemo VR preset koji je ugrađen u Unreal Engine. Važno je prije pokretanja Unreal Enginea pokrenuti i Oculus softver. Steam VR pokreće se sam od sebe kad se pokrene i Unreal Engine. Pokrećemo program pritiskom na tipku "Launch" u gornjem desnom kutu.

Za stvaranje projekta odabiremo opciju "Games" što će nas odvesti na izbornik već unaprijed konfiguiranih projekata koji sadrže minimalno što je potrebno kako bi se testirale i izrađivale dodatne funkcionalnosti.

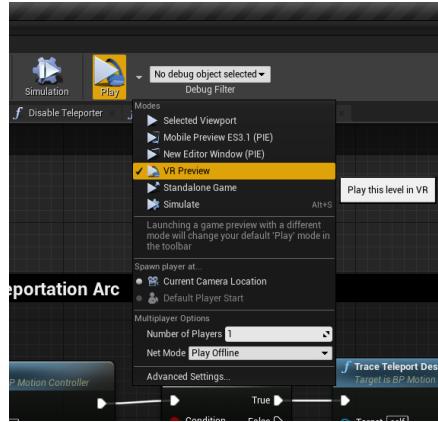
Odabiremo "Virtual Reality" preset u kojem je već implementirana softverска podrška za kontrolere, VR headset, primanje objekata u virtualnom svijetu te teleportacija. Pritisom na tipku "Create Project" projekt se stvara.



Slika 6: Prikaz Koraka za kreiranje projekta

Kad se otvori projekt moramo odabrati razinu koja je napravljena za naš tip uređaja za virtualnu stvarnost. Odabiremo desni preset i razinu pronalazimo u *content browseru* na dnu ekrana.

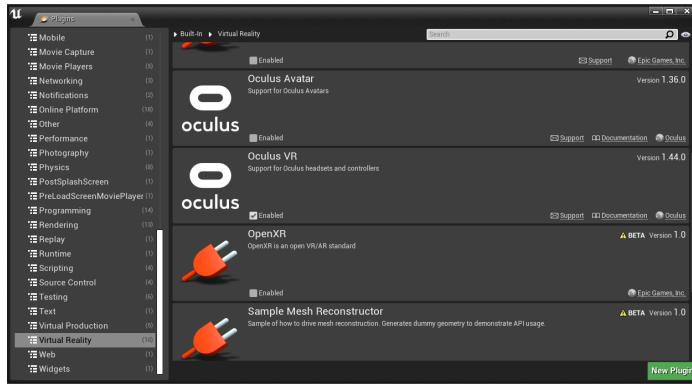
Navigiramo na lokaciju *VirtualRealityBP/Maps/MotionControllerMap* i dvostrukim klikom otvaramo tu razinu. Po otvaranju karte u gornjoj alatnoj traci odabiremo padajući izbornik pored tipke *play* i odabiremo *VR preview*.



Slika 7: Prikaz izbornika kojim se dolazi do opcije *VR preview*

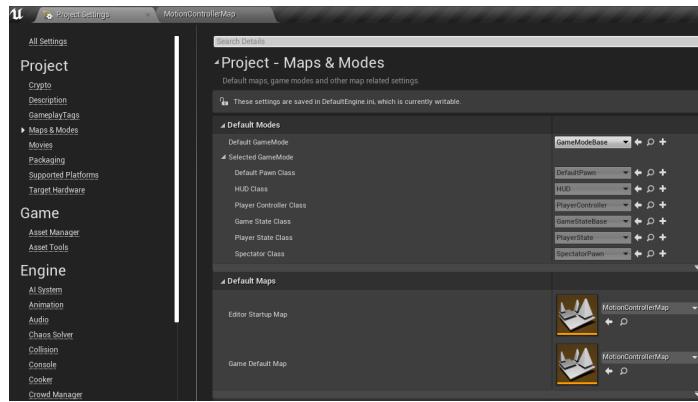
1.5 KONFIGURIRANJE PROJEKTA

U slučaju da *VR preview* ne radi, potrebno je omogućiti *plug-in OculusVR* (*Edit>Plugins*, vidi sliku 8).



Slika 8: Prikaz *Project Settings* za mijenjanje prepostavljene mape.

Osim tih postavki, potrebno je i izmijeniti početnu mapu koja će se otvarati pri otvaranju projekta. U *Edit>Project Settings* pod *MapsAndMods* *EditorStartUpMap* i *GameDefaultMap* trebaju se postaviti na *MotionControllerMap* (vidi sliku 9).



Slika 9: Prikaz *Project Settings* za mijenjanje pretpostavljene mape.

Ako je sve ispravno postavljeno, na uređaju bi se trebao prikazati VR pogled.

U Unreal Engineu se programira prvenstveno korištenjem *Blueprinta* metodom poznatom pod nazivom *Visual Scripting*. To znači da se većina koda napiše pomoću *Blueprinta* dok se određene stvari mogu napisati i u C++-u. Teoretski je moguće napisati igru i u C++-u, ali to se u pravilu ne radi jer je Unreal engine napravljen tako da se dizajn igre odvaja od programiranja dijelova klase pa se glavne funkcionalnosti uglavnom *blueprintaju*, a detalji se programiraju ukoliko su jako resursno intenzivne.

1.6 ZAKLJUČAK

U ovom uvodu instalirali smo okruženje te se upoznali s glavnim elementima Unreal Enginea. Mnogi koncepti koji vrijede u programiranju poslovnih aplikacija vrijede i u Unrealu, kao što je verzioniranje, jedino treba posebno pripaziti na velike datoteke i na `.gitignore`.

Teme i koncepti kojima se može dodatno obogatiti projekt su:

- verzioniranje (`git`, LFS...)
- detaljnije upoznavanje s elementima korisničkog sučelja u okruženju

Prostor za bilješke:

2 GLAVNI LIK

Unreal engine ima *preset* za svaku popularnu vrstu video igre kao što su *top down*, platformske igre, 2D igre itd. pa se tako populariziranjem VR igri razvio i VR *preset*. U tom *presetu* već je implementirana funkcionalnost kretanja, odnosno teleportacije, no i dalje je važno razumjeti na koji način teleportacija funkcioniра kako bi smo mogli lakše implementirati ostale funkcionalnosti u skladu s onime što već imamo.

Teleportacija je implementirana kroz dva objekta koji međusobno komuniciraju. Jedan objekt je jednostavno rečeno kamera kroz koji igrač vidi svijet, dok je drugi kontroler, odnosno kontroleri.

Pomoću ta dva objekta odrađuje se funkcionalnost teleportacije na način da se u smjeru prednjeg vektora odabranog kontrolera iscrtava luk na čijem mjestu kolizije s tlom dobivamo ciljni vektor teleportacije.

2.1 MOTIONCONTROLLERPAWN

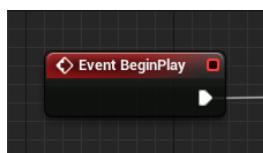
Element, odnosno **actor** iz naslova je VR uređaj u virtualnom prostoru.

Taj **actor** predstavlja VROrigin ili PlayArea u kojem se nalazi igrač, element toga je i VR uređaj kao kamera. U nastavku slijedi opis svih funkcionalnosti koje su postavljene pomoću *blueprinta* u navedenom actoru te u BP_MotionControlleru.

2.1.1 POSTAVLJANJE VISINE IGRAČA TE POSTAVKE VR UREDAJA

Događaji (**Eventi**) su objekti nalik funkcijama koji se pozivaju nakon što se neki događaj dogodi u programu, odnosno igri.

Sve što će se sada opisati okida se na događaj **BeginPlay**. Događaja u Unreal Engineu ima mnogo. Za sada će se koristiti događaji **BeginPlay**, **tick** (**EventTick**) ili *input* od strane igrača. Svaki objekt ima i svoj događaj **BeginPlay** koji se pokreće kad se objekt instancira u igri.



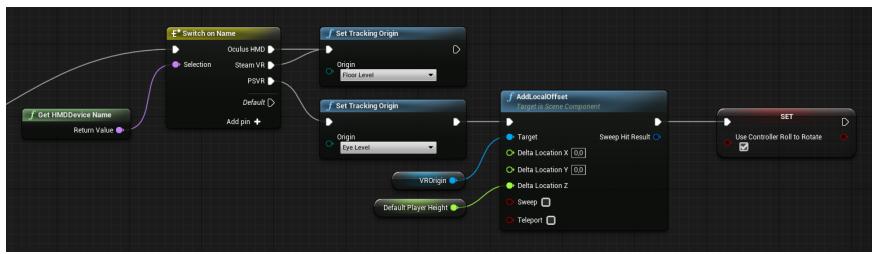
Slika 10: Događaj za početak igre za *actora*

Prvi skup čvorova koji slijedi nakon čvora početka igre služi za postavljanje visine igrača ukoliko se radi o PSVR uređaju jer PSVR ne zna gdje se nalazi pod te ne može sam odrediti visinu na kojoj se nalazi igrač. Ovisno

PSVR – Play Station Virtual Reality

o kvaliteti osvjetljenja PSVR uređaj može procijeniti visinu. Praćenje (eng. *tracking*) *headseta* i kontrolera se na VR uređajima vrši pomoću namjane dviju kamara koje onda mogu uspoređivati dvije iste scene iz različitog kuta gledanja te na taj način dobiti informacije o trodimenzionalnosti prostora kojeg pokrivaju.

Ovisno o uređaju određuje se ime uređaja koji se kasnije koristi za identifikaciju scenskog elementa kojem je taj uređaj pridružen. Drugim riječima, stvarni uređaj se veže s virtualnim te oni postaju isti element. Na taj način pokretanjem glave koja mijenja lokaciju fizičkog uređaja izaziva i mijenjanje lokacije u virtualnom prostoru čime rukuje Unreal Engine. Detalji su prikazani na slici 11.



Slika 11: Čvor koji bira offset i naziv *headseta* s kojim će raditi Unreal Engine

PSVR uređaj ima samo jednu kamenu koja prati intezivne izvore svjetlosti na PSVR uređaju te po tome može znati samo trodimenzionanu rotaciju uređaja i nema prostornu osvještenost o poziciji samog uređaja u odnosu na bilo koji drugi element u prostoru.

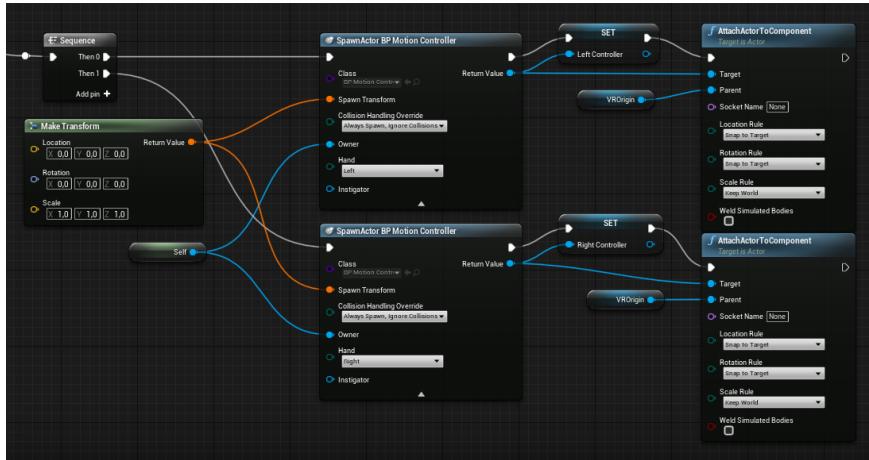
2.1.2 STVARANJE I POVEZIVANJE KONTROLERA

Kontroleri (ruke) nalaze se kao odvojeni *actor* u Unreal Engineu kao i u drugim *game engineima* jer se kreće od prepostavke da u projektu može biti i samo HMD uređaj što obično i smatramo VR uređajem bez svojih kontrolera. To su uređaji poput Samsung Gear VR, Google Cardboarda, spomenuti PSVR itd. Ukoliko uređaj ima svoje kontrolere, a najpopularniji VR uređaji poput HTC Vive, Oculus Rift S i Quest te Valve Index imaju kontrolere, onda je te kontrolere potrebno spojiti s uređajem. U nastavku (slika 12) se može vidjeti da se nakon početka igre stvaraju dvije instance kontrolera koje odgovraju lijevoj i desnoj ruci.

HMD – Head Mounted Display

Kod oba kontrolera vlasnik je **Self** što je referenca na *blueprint* u kojem se trenutno izrađuju čvorovi. Prisjetimo se da je to *blueprint* od **MotionControllerPawn**, odnosno *blueprint* od *pawn* VR *actora* te ovdje sa **Self** označavamo upravo njega. Postavljaju se vrijednosti unutar objekata **Left Controller** i **Right Controller** te se VR uređaju pridružuju reference na objekte **Left Controller** i **Right Controller**. Stvaranje ruku u VR okruženju odvija se bez obzira na kolizije koje se mogu dogoditi jer uvijek želimo da se ruke stvore iz tog razloga što su ruke na neki način tipkovnica i miš VR uređaja uz njega samog.

Na slici 12 dodaje se i osnovna funkcija `MakeTransform` koja vraća varijablu tipa `Transform` koja je neophodna za funkciju `SpawnActorFromClass`. `Set` zapisuje referencu u `MotionControllerPawn` kako bi se u *blueprintu* moglo raditi s kontrolerom kao s *actorom*. Postojanje reference omogućuje *castanje* tog objekta u `BP_MotionController`.



Slika 12: Stvaranje kontrolera kao *actora*, postavljanje referenci i pri-druživanje VROriginu kao komponente

2.1.3 PROCES TELEPORTACIJE

Teleportacija je kompleksan element pa se neće moći jednostavno prikazati i objasniti u jednom skupu čvorova. Krenut ćemo od elementa koji je najjednostavnije razumjeti jer sadrži funkciju za teleportaciju. Izvršavanje teleportacije događa se nakon provjere određenih sigurnosnih i praktičnih uvjeta. Važno je imati u vidu da se izvršavanje ovog dijela funkcionalnosti ne izvršava „odmah“ nakon korisnikovog zahtjeva za teleportacijom (po pritisku tipke), već je ovo normalan slijed operacija nakon uspješnih provjera. *Odmah* je napisano pod navodnicima jer se korisniku čini da je u tom trenutku, ali zapravo se cijeli niz događaja odradi prije teleportacije.

U samoj funkcionalnosti teleportacije postoje još dvije provjere (slika 13):

- nalazi li se već igrač u procesu teleportacije kada traži teleportaciju (`IsTeleporting`)
- je li mjesto na koje se želi teleportirati valjano (`IsValidTeleportDestination`)

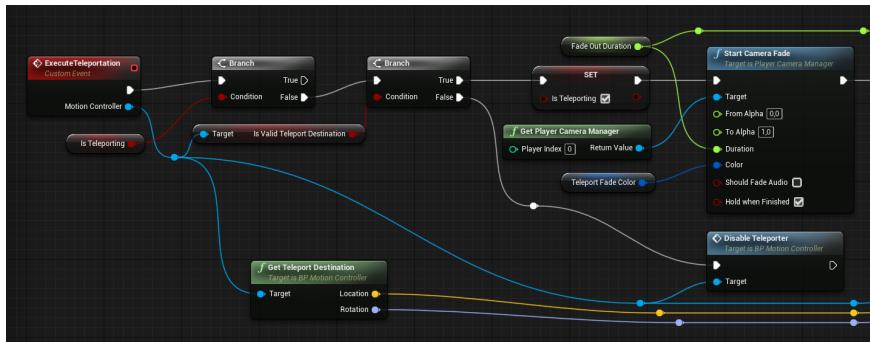
Ukoliko su zadovoljena prethodna dva uvjeta onemogućuje se prolaz novih zahtjeva za teleportaciju tako da se proces teleportacije postavlja na vrijednost `True`. Kad je prvi uvjet zadovoljen (`False` ovom slučaju), a drugi nije, briše se vizualna reprezentacija teleportacije.

Zadovoljavanje oba uvjeta pokreće funkciju koja započinje zatamnjenje (eng.

`fade out` – padanje mraka na oči

fade out) ekrana na VR uređaju. Zatamnjenje traje isto onoliko vremena koliko traje i funkcija čekanja vremena tako da se teleportacija dogodi točno nakon potpunog zatamnjenja ekrana. U drugim jezicima takve se funkcije obično zovu `sleep()`.

To zatamnjenje zapravo je crnjenje virtualne kamere tako da se stavi crna boja ispred nje i njoj se mijenja *opacity* s 0% na 100% kada zbilja to percipiramo kao apsolutno crnilo.



Slika 13: Prikaz dijela procesa teleportacije u događaju `ExecuteTeleportation`

Valjano mjesto za teleportaciju ili kretanje je mjesto kojeg objekt `NavMeshBounds` označava kao slobodnog za kretanje, a mjesto na kojeg se igrač teleportira prikazano je indikatorom mjesta teleportacije (v. sliku 14).

Prije teleportacije poziva se događaj `DisableTeleporter` iz `BP_Motion Controller`, koji postavlja vidljivost strelice, točke teleportacije i sjaja na strijelici na laž, odnosno sakriva ih (slika 15, a detaljnije opisano u 2.1.8)

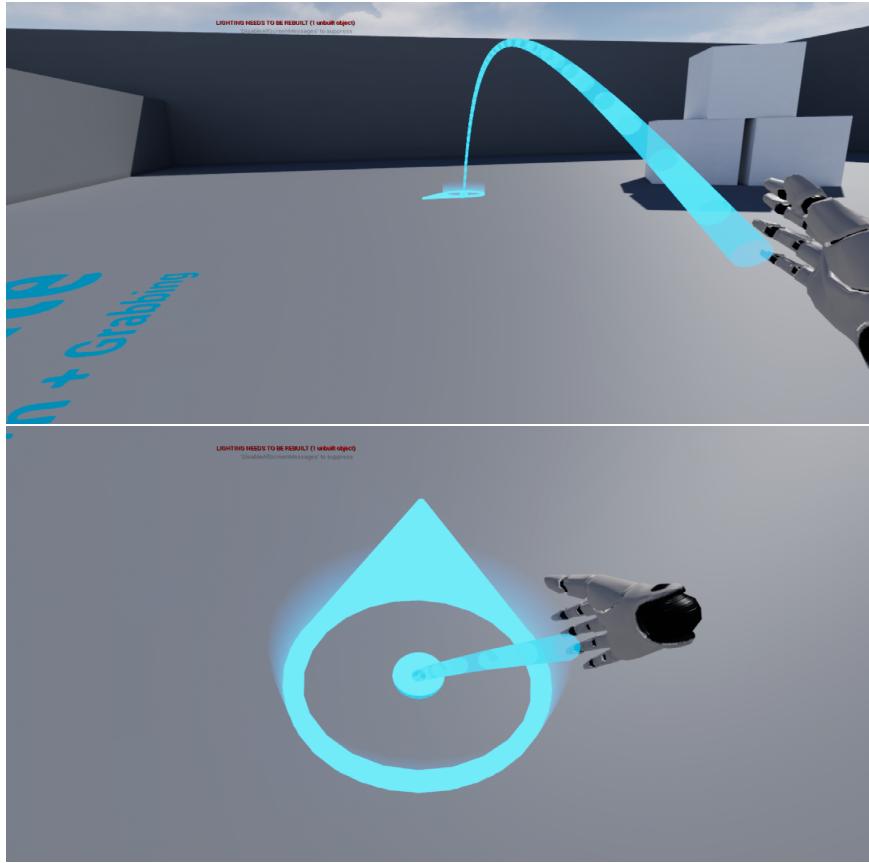
Takva zaokruženost funkcionalnosti ispitivanjem istinitosti varijable zapravo čini jedan semafor koji ne dopušta novo izvršavanje funkcionalnosti dok prethodno izvršavanje nije došlo do kraja. Funkcija teleportacije lokaciju i rotaciju teleportacije dobiva od funkcije za dobivanje željenog mesta za teleportaciju. Detaljan opis slijedi kasnije te vizualni prikaz na slici 16.

pawn – realizacija igrača u virtuelnom svijetu (ne nužno u obliku čovjeka)

U Unreal Engineu objekt slobodnog kretanja zove se `NavMeshBoundsVolume`

2.1.4 DOBIVANJE LOKACIJE I ROTACIJE ZA MJESTO TELEPORTACIJE

Lokacija HMD-a uzima se funkcijom `GetOrientationAndPosition` te se vraća objekt tipa `Vector` od čega se uzimaju samo *X* i *Y* vrijednosti jer je visina igrača nevažna, odnosno ne mijenja se. *X* i *Y* se tada rotiraju prema Rotatoru `TeleportRotation` koj se postavlja na Tick na onom kontroleru na kojem se trenutno izvršava teleportacija. To se događa u Event Graphu `MotionControllerPawna` tako da se iz funkcija `GetMotionControllerThumbLeft_Y` i `GetMotionControllerThumbLeft_X` dobivaju inputi kontrolera koji stvaraju Rotator u funkciji `GetRotationFromInput` te se onda postavlja vrijednost u `TeleportRotation`.



Slika 14: Indikator mjesto teleportacije

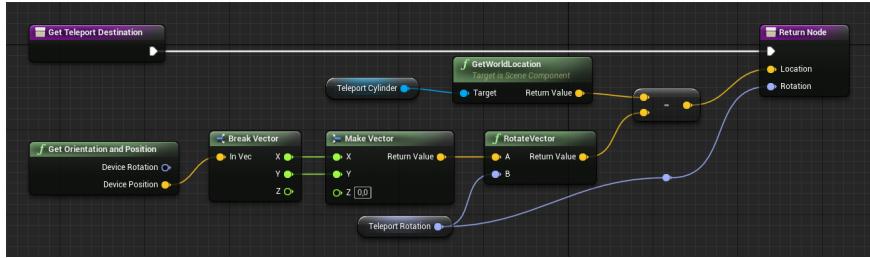


Slika 15: Funkcija DisableTeleport

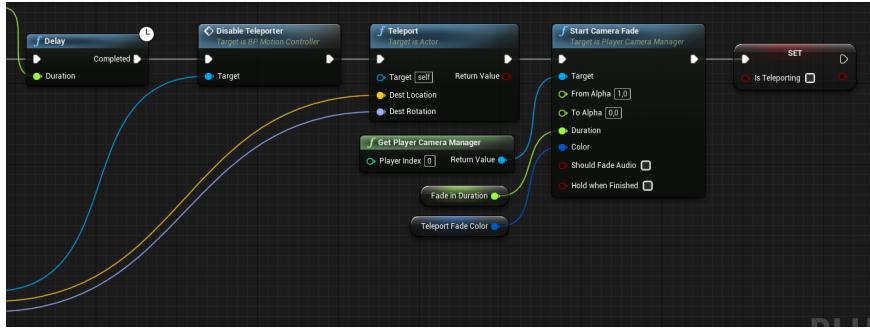
jednost `Rotator TeleportRotation`.

Tako dobiven `Vector` oduzima se od lokacije aktora `TeleportCylinder` jer se radi s `VROrigin`om i zato nije dovoljno samo postaviti `VROrigin` na lokaciju.

Nakon teleportacije započinje proces vraćanja crnog ekrana u normalan pogled (eng. *fade in*) VR uređaja. Teleportacija može bez problema raditi i bez korištenja zatamnjivanja u ove dvije spomenute funkcije i ne čini nikakvu razliku u funkcionalnosti teleportiranja, ali poboljšava korisničko iskustvo. Obična teleportacija bez korištenja zatamnjivanja ekrana djeluje neprikladno, odnosno igrač može dobiti tzv. *motion sickness*, nestručno prevedeno kao pokretnu bolest. Zatamnjivanje ovdje teleportaciju čini prirodnijom koliko god se proces teleportiranja može nazvati prirodnim. Prikaz je na slici 17.



Slika 16: Prikaz rotacije nakon teleportacije



Slika 17: Vraćanje kamere iz crnila i teleportacija

2.1.5 RUKOVANJE KONTROLERIMA

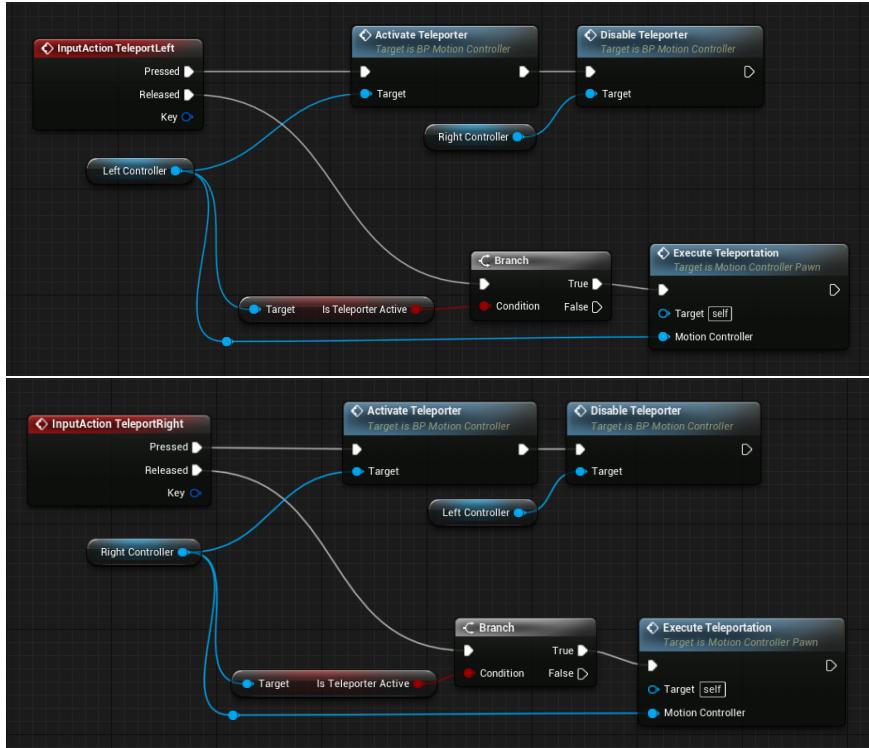
Sljedeće funkcionalnosti okidaju se na igračev unos preko kontrolera. Teleportacija je već objašnjena, a u nastavku slijedi objašnjenje i ostalih događaja. Početni događaj koji to okida može se vidjeti prikazan crvenom bojom na slici 18. Element na kojeg utječemo je objekt kojeg smo referencirali kod spajanja kontrolera s VR uređajem. Pošto je svaki kontroler odvojeni objekt moramo definirati što se događa i za jedan i za drugi kontroler pritiskom na odgovarajuću tipku. Ako i na jednom i na drugom kontroleru želimo implementirati funkcionalnost teleportacije to znači da je kod ili u ovom slučaju skup čvorova isti osim što se referenciramo na različiti kontroler. Zbog tog razloga će se u nastavku opisati slučaj za samo jedan kontroler jer je implementacija za drugi kontroler ista.

Prvo se okida događaj `InputAction TeleportLeft` koji je vezan na pritisak gljivice na lijevom kontroleru. Pritisak na tu tipku stvara se vizualna reprezentacija za mjesto teleportacije na kontroleru na kojem je pritisнутa tipka. Naziv te funkcije je `ActivateTeleporter`, a detalji se saznaju u odlomku 2.1.6. Ako se u trenutku kad je lijeva tipka za teleportaciju pritisnuta i u tom se trenutku pritisne ista tipka na desnom kontroleru, varijabla `IsTeleporterActive` za lijevi kontroler postavi na `False` što znači da se na otpuštanje lijeve tipke na kontroleru teleportacija neće izvršiti jer nisu svi uvjeti zadovoljeni.

Tako je napravljeno kako igrač ne bi mogao izazvati grešku na način da

pritisne tipku na jednom kontroleru i dok je drži pritisnutu, pritišće tipku i na drugom kontroleru. Na ovaj način to se izbjegne jer aktivacija na jednom kontroleru automatski znači deaktivaciju na drugom kontroleru. Prikaz je na slici 18

Na otpuštanje pritiska tipke provjerava se je li postavljen uvjet iz funkcije aktivacije vizualne reprezentacije teleportacije. Ukoliko je, odlazi se u funkciju teleportacije koju smo već opisali. Na slikama se usporedbom može vidjeti kako se u obje slike radi o istoj funkcionalnosti te je jedina razlika u objektu kontrolera za kojeg želimo aktivirati ili deaktivirati vizualnu reprezentaciju mesta teleportacije.



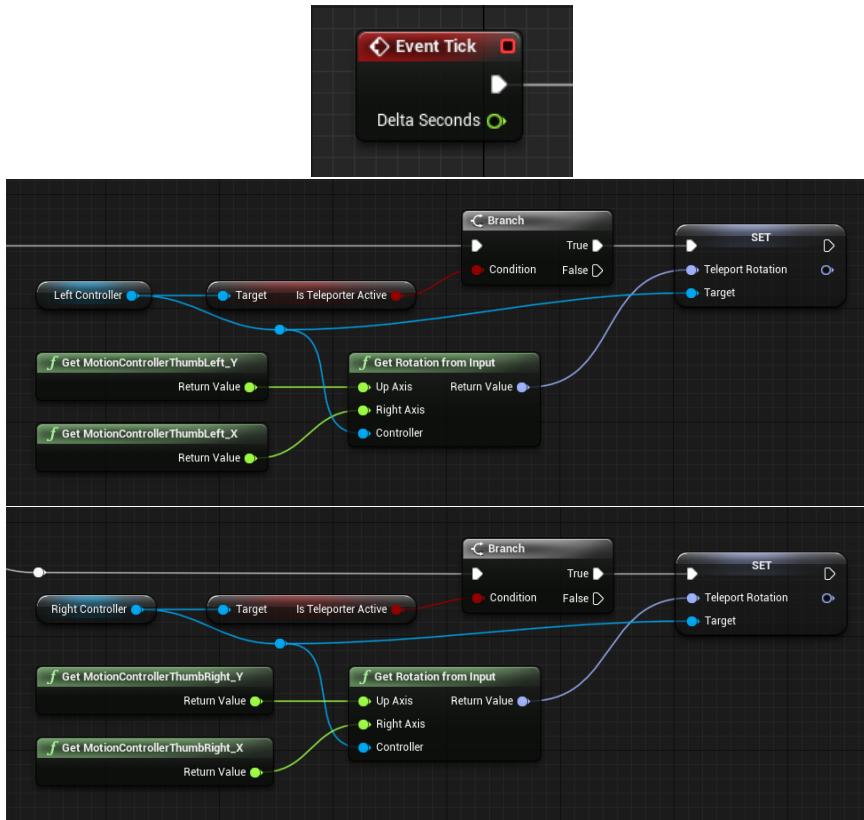
Slika 18: Događaji koji se aktiviraju pritiskom i puštanjem tipke *grip*

2.1.6 POSTAVLJANJE ROTACIJE MJESTA TELEPORTACIJE

Funkcionalnost koja slijedi izazvana je događajem `EventTick` (*tick*, slika 19) koji se neprestano okida dok je igra pokrenuta i dok je taj *actor* unutar igre. To je jedan od spomenutih događaja okidača koji je također označen crvenom bojom u *blueprintovima*. Iste funkcionalnosti vrijede i za lijevi i za desni kontroler tako da se analogijom mogu poopćiti.

Na objektu odabranog kontrolera ispituje se aktiviranost vizulane reprezentacije mesta teleportacije te ako je aktivna, postavlja se varijabla rotacije teleportacije *x* i *y* osi na gljivici na očitanu vrijednost. Prisjetimo se da se do ovog dijela funkcionalnosti dolazi svakim *tickom* što znači da se mogućnost

promjene rotacije teleportacije događa neprestano odnosno dokle god je valjan uvjet ulaska u petlju, a to je da je pritisnuta tipka za teleportaciju. Isti slučaj je i kod drugog kontrolera pa to nećemo posebno objašnjavati osim možda napomenuti očito – razlikuje se objekt nad kojim se vrši funkcionalnost.



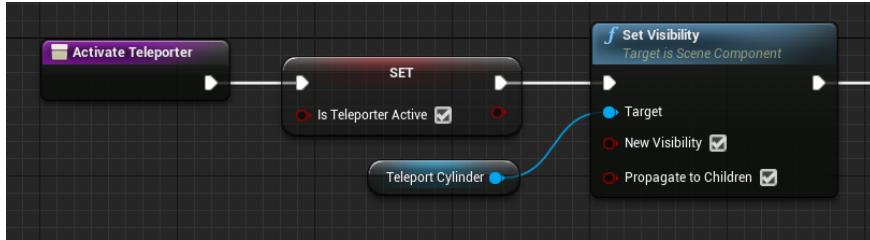
Slika 19: Slika događaja EventTick (*tick*), prikaz dobivanja lokacije i rotacija nakon teleportacije

Sada prelazimo na dio koda koje se nalazi u *blueprintu* kontrolera. Prvo ćemo krenuti od dobivanja lokacije i rotacije za mjesto teleportacije. Zbog VR uređaja koji nemaju lokacijsku osvještenost neki su detalji malo složeniji.

2.1.7 AKTIVIRANJE VIZUALIZACIJE MJESTA TELEPORTACIJE

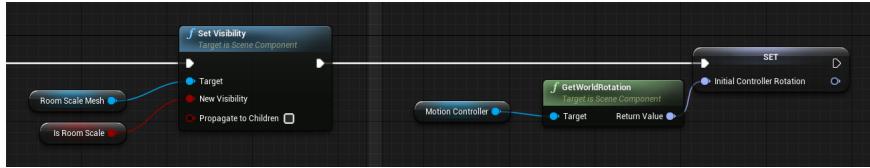
Poziv ove funkcije već smo koristili kod pritiska tipke kontrolera na *blueprintu* VR uređaja. Ova funkcija postavlja istinitost varijable aktivacije vizualizacije mjesta teleportacije na True te postalja vidljivost strelice koja vizualizira lokaciju i rotaciju na vidljivo. Jednostavnost ovog skupa čvorova može se vidjeti na slici 20.

Postoje i dodatne funkcionalnosti koje nisu potrebne za normalno funkcioniranje teleportacije na Oculus Rift S uređaju, no za neke druge uređaje je korisno. Svi VR uređaji koji koriste *light-house* praćenje mogu se poslužiti



Slika 20: Prikaz postavljanja varijable za aktivaciju teleportacije i postavljanja vidljivosti strelice (TeleportCylinder)

i vidljivošću scenske komponente koja je prostor u kojem se igra. Ovdje to ne možemo ispitati pa nećemo detaljnije ni obrađivati. Još jedna dodatna funkcionalnost je praćenje rotacije kontrolera prema rotaciji zapešća. Ova rotacija se može koristiti u igrama, ako želimo da igrač određuje rotaciju teleportacije preko rotacije kontrolera odnosno svojeg zapešća. Nekada može biti zanimljivo no više je umarajuće od rotacije preko gljivice zato što je neprirodno te ga mi nećemo koristiti. Na slici 21 se može vidjeti opisani dio.

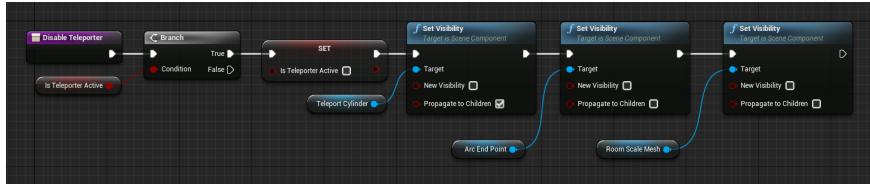


Slika 21: Postavljanje vidljivosti indikatora mesta teleportacije na vidljivo (True) postavljanje varijable rotatora za određivanje rotacije prema zapešću

2.1.8 DEAKTIVIRANJE VIZUALIZACIJE MJESTA TELEPORTACIJE

Funcionalnosti koja uklanja vizualizaciju mesta teleportacije je ona koja je također potrebna kako se vizualizacija ne bi stalno prikazivala. Kako smo vrijednost varijable aktiviranosti vizualizacije teleportacije postavili na **True**, ovdje provjeravamo tu varijablu te ju postavljamo na **False**. Na taj način provjeravamo je li igrač ušao u proces aktivacije jer nema smisla prolaziti kroz kod za postavljanje vidljivosti na nevidljivo ukoliko je objekt već sakriven. Postavlja se vidljivost indikatora lokacije i rotacije na nevidljivo isto kao i vidljivost kraja luka za teleportaciju. Na kraju se postavlja vidljivost za *room scale* element koji ponovo nemamo kod našeg VR uređaja, no zbog kompatibilnosti ćemo ga samo spomenuti.

Oculus Riftovi imaju tzv. *inside-out* praćenje što znači da za praćenje fizičke pozicije kontrolera i samog VR uređaja nisu potrebni vanjski elementi tj. *lighthouse* na dva suprotna kraja prostora za igru. VR uređaji koji koriste *inside-out* praćenje imaju kamere na samom VR uređaju koje preko izraženih točaka u stvarnom prostoru poput ruba stola, točke nagle promjene u boji tepiha i sl. određuju vlastite točke praćenja u stvarnom svijetu te na taj način određuju fizičku poziciju VR uređaja i kontrolera što se na kraju preslikava i u virtualno okruženje.



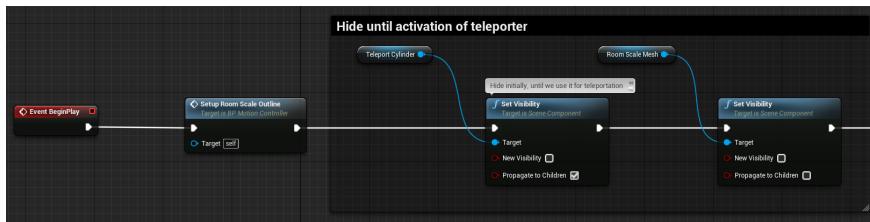
Slika 22: Deaktivacija vizualnog indikatora mjesto teleportacije

2.2 BP_MOTIONCONTROLLER

Na sam početak igre pokreće se događaj `BeginPlay` za *actora* `BP_MotionController` jer se on nalazi kao instanca u *actoru* `MotionControllerPawn` (i to čak 2 puta). Prvi događaj koji se okida događajem `BeginPlay` je događaj `SetupRoomScaleOutline`. Taj događaj za nas nije bitan jer Oculus ima vlastiti `Play Area Bounds` ugrađen u softver VR uređaja.

Jedino je važno znati da taj događaj služi za postavljanje ograničenja za igrača u smislu prostora unutar kojeg se u virtualnom prostoru može slobodno kretati bez izlaska iz vlastito postavljenog stvarnog `PlayArea`. Nakon izvršenja tog događaja postavlja se vidljivost strelice (*teleport cylinder*) i `Room Scale Mesh` na nevidljivo. Funkcija kojom se to radi za obje komponente je `SetVisibility`. `TeleportCylinder` je spojen na ruku iako se za igrača on ne vidi dok za to ne dođe vrijeme (u procesu teleportacije). `Room Scale Mesh` nam nije potreban pa njegovo postavljanje vidljivosti također ne čini nikakvu razliku za nas. Projekt će biti kompatibilan sa svim VR uređajima s kontrolerima, no detaljnije ulazimo samo u logiku iza Oculus orijentiranih funkcija jer upravo u njima radimo (slika 23).

`Play Area Bounds` od Oculusa zove se `Guardian` i jako je fleksibilan za razliku od ostalih tj. SteamVR `Play Area Bounds` jer se u `Guardian` mogu individualno ocrtati virtualni zidovi te na taj način možemo imati prilagođeni `Play Area Bounds` prostoru u kojem se nalazimo, SteamVR `Play Area Bounds` je u tom smislu ograničen jer on može imati samo četverokut, peteckut ili neki drugi pravilni mnogokut s vise kutova



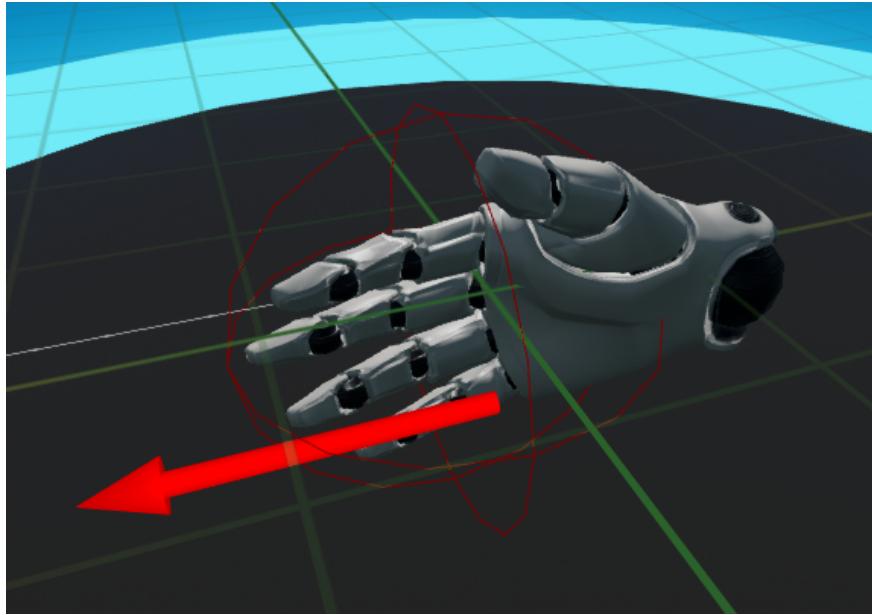
Slika 23: Postavljanje vidljivosti elemenata

2.2.1 MIJENJANJE SKALIRANJA

Dalje se „mijenja skaliranje“ *mesh-a* ruke funkcijom `SetWorldScale3D` uko-liko je ruka lijeva na 1 po X, 1 po Y i -1 po Z. To znači da se veličina *mesh-a* množi sa 1 po X i Y te ostaje ista, ali se množi sa -1 po Z što zapravo znači da veličina mijenja predznak što nije moguće pa to zapravo okreće *mesh* vertikalno. U *viewportu* *actora* `BP_MotionController` (na slici 2.2.1) možemo primjetiti da je osnovni *mesh* za ruku zapravo desni *mesh* odnosno *mesh* za

Vrsta varijable `Hand` je `EControllerHand` koje se i koristi u svrhe diferenciranja kontrolera

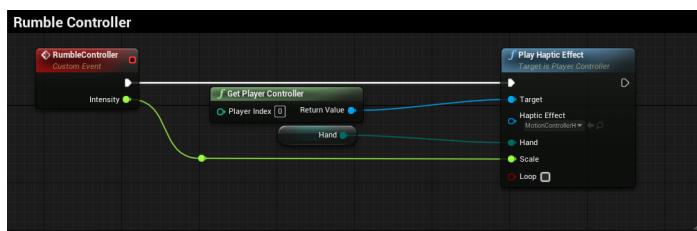
desnu ruku. Vertikalnim zaokretanjem *mesh-a* dobit ćemo njegovu zrcalnu sliku, odnosno lijevu ruku koju sada možemo i koristiti u svrhu lijeve ruke za kontroler.



Slika 24: Prikaz ruke u prostoru iz *viewporta actora*

2.2.2 VIBRACIJE

Sada će se ukratko objasniti događaj **RumbleController** koji u sebi ima samo jednu funkciju naziva **PlayHapticEffect**. Ovoj je funkciji za ulaz potreban kontroler i ruka. Kontroler se dobiva preko funkcije **GetPlayerController**, no time smo označili oba kontrolera jer je *actor* **BP_MotionController** *actor* za oba kontrolera. Zato je još potrebno staviti uvjet **Hand** čime određujemo na koji od dva kontrolera točno želimo da funkcija **PlayHapticEffect** djeliće. Mi ćemo taj događaj deaktivirati jer ne želimo dodatno trošiti baterije unutar kontrolera, a i nećemo koristiti mogućnost hvatanja objekata u kojoj se taj događaj i okida kao što nećemo implementirati da se taj događaj okida na neki pokret. Detalji prikazani na slici 25



Slika 25: Prikaz Rumble Controllera

Funkciju **PlayHapticEffect** možemo koristiti i kod običnih **Gamepad** kontrolera koji podržavaju “povratnu haptičnu informaciju” (iz engl. *force feedback*) i kod njih ne moramo paziti ili uopće dodati distinkciju ruku kao uvjet jer kontroler ima samo jedan fizički sklop koji može emitirati vibraciju. Noviji kontroleri mogu imati više emitera vibracija, no to je nesto komplikiranija tema u koju necemo ulaziti

2.2.3 HVATANJE RUKOM

Pritiskom na bilo koju od dviju *grip* tipki koje smo spojili u *inputima* na pristajućim događajima **GripLeft** i **GripRight** mijenja se i položaj u kojem se ruka nalazi. Točnije, pritiskom *grip* tipke na određenom kontroleru virtualna ruka (odnosno šaka) koja se nalazi u virtualnom prostoru na mjestu kontrolera u stvarnom prostoru se stisne. Šaka se sastoji od jedinstvenog *mesh-a* (3D modela), no njen se kostur može naći u različitim pozicijama.

Pozicija kostura u kojoj će se u određenom trenutku nalaziti model ruke provjerava se na *tick*. Prvi uvjet koji može biti zadovoljen ako će se šaka nalaziti u stanju hvatanja (*grip*) je da instanca **BP_MotionController-a** ima nekakvog *actor-a* prikvačenog (en. *attached*) na sebe što je uvjet **WantsToGrip** jer se on postavlja na **True** prilikom hvatanja i na **False** prilikom puštanja određenog *Actor-a*. To se može provjeriti u funkcijama **GrabActor** i **ReleaseActor** jer se one ovdje neće posebno objašnjavati budući da nam nisu potrebne za našu funkcionalnost, no potrebno je razumjeti kako se mijenja stanje kostura ruke, pa samim time i njenog 3D modela. Drugi uvjet je da je varijabla, odnosno objekt **Attached Actor**, različit od **null**, odnosno da u njemu postoji određeni zapis o *actor-u* kojeg držimo. To možda zvuči na prvi pogled kao da su oba uvjeta ista što bi i bio slučaj.

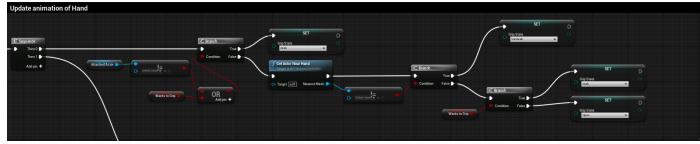
Dovoljan uvjet za obavljanje provjere bio bi samo **Bool** varijabla **WantsToGrip** budući da se ona uvijek postavlja kod hvatanja na **True** i kod puštanja na **False** bez obzira na to je li se na ruku prikvačio neki *actor* ili ne, što znaci da mi želimo sklopiti ruku bez obzira na to jesmo li nešto njom uhvatili ili ne. Po ovoj logici nema razloga za razliku je li nešto uhvaćeno ili ne što možemo staviti i u stvarnu perspektivu kako bi logika bila jasnija.

Najbanalniji primjer je da zamislimo da nam netko dobaci bombon. Ruku ćemo zatvoriti prilikom pokušaja hvatanja bombona bez obzira jesmo li bili uspješni u hvatanju ili ne. Ista logika vrijedi i ovdje. Po našem razmišljanju ne možemo doći do valjanog zaključka zašto postoji razlika između ova dva uvjeta. Bilo kako bilo, kada je bilo koji od prethodna dva uvjeta zadovoljen što uključuje i opciju kada su oba zadovoljena, **GripState** odnosno položaj u kojem se ruka (šaka) nalazi je **Grab**, odnosno zatvorena.

Ako niti jedan uvjet nije zadovoljen, poziva se funkcija **GetActorNearHand** koja izračunava i vraća najbliži 3D model ili *mesh* te, ako je on valjan, odnosno nije **null**, ruka se postavlja u stanje **CanGrab**. Ovo je stanje u kojem se u našoj igri ruka neće nalaziti pa ga nećemo ni dodatno objašnjavati. Zadnji uvjet za položaj ruke je zadovljavanje jednog od prva dva uvjeta, dakle je li pritisnuta ili puštena tipka za hvatanje. Ako je pritisnuta tipka za hvatanje tj. **WantsToGrip** je postavljen na **True**, onda će šaka biti zatvorena, ako ne, onda će šaka biti otvorena. Detalji se nalaze na slici 26.

Varijabla **GripState**, koja se postavlja u prijašnjem kôdu, tipa je **Enum**

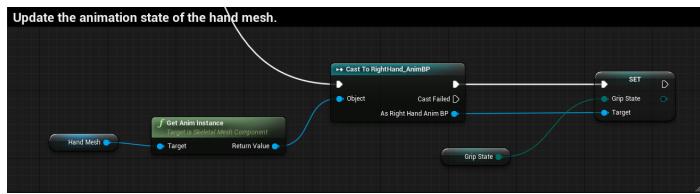
Proces okoštavajna (eng. *rigging*) je proces u kojem se na određeni 3D ili 2D model stavlja kosti i zglobovi. Potrebno je precizno postaviti kosti i zglobove kako bi se 3D model mogao dobro i realistično animirati. Unreal Engine 4 podržava sustav za okoštavanje i za animiranje, no mi nećemo u okoštavanje ni animiranje ulaziti jer nam je cilj pokazati programerski aspekt razvoja, iako je samo s programiranjem izrada video igre po današnjim standardima praktično nemoguća.



Slika 26: Prikaz hvatanja rukom

te ona sama za sebe ne mijenja ništa osim svoje vrijednosti. Promjene se događaju u animacijskom *blueprintu* za *hand mesh*. Na *tick* se na vrijednost **GripStateEnum** postavlja varijabla u animacijskom *blueprintu* **RightHand_AnimBP**.

Kako bismo mogli pristupiti *castanju* u taj *blueprint* potreban nam je objekt *castanja* kojeg dobivamo od samog 3D modela ruke koji je komponenta **Hand Mesh**. Preko funkcije **GetAnimInstance** dobivamo instancu animacijskog seta kojeg ovaj *mesh* koristi. To nam je objekt kojeg mozemo *castati* u *castanju* te kao odgovor dobiti isti objekt koji trebamo kao metu (*target*) postavljanja varijable **Grip State** u njemu samom. To je ono što će promijeniti "stisnutost" ili otpuštenost šake. Detaljan prikaz na slici 27.



Slika 27: Castanje u animation blueprintu i postavljanje Grip Statea

Svaki *animation blueprint* se spaja sa *skeletal meshom* nekog 3D modela (*mesha*), *animacijski set* se bez prilagođavanja ne može jednostavno koristiti na nekom drugom *meshu* za kojeg nije bilo originalno napravljen.

2.2.4 STVARANJE LÜKA ZA VIZUALIZACIJU

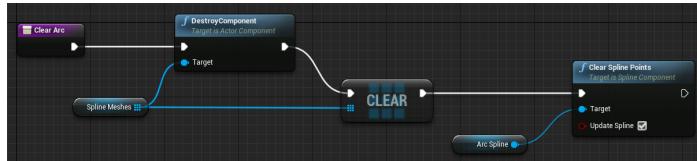
Sljedeća važna stvar koja se događa na *tick* pod uvjetom da se želimo teleportirati je stvaranje luka za vizualizaciju distance na koju se igrač teleportira. Prije te provjere pokreće se funkcija **ClearArc**. Točke koje čine luk prikazanim pritiskom na tipku teleportacije spremaju se u polje naziva **SplineMeshes**. Prvo je važno da se na svaki *tick* taj luk briše bez obzira stvara li se novi ili ne. To možda na prvi pogled zvuči neoptimalno, no to brisanje i stvaranje se odvija toliko brzo da čovjek ne može primijetiti da se luk zapravo briše. Ono što se dobije kao rezultat je da se taj luk pomiče kako igrač pomiče kontroler kako bi odredio daljinu teleportacije iako se on zapravo neprestano briše i stvara na novoj ažuriranoj lokaciji prema ulazima koje dobiva od kontrolera. Ta se akcija, ovisno od jačini grafičke podrške u računalu događa mnogo puta u sekundi. Detalji su slići 28

U funkciji **DestroyComponent** uklanja se svaka komponenta varijable **Spline Meshes** koja je tipa **Spline Mesh Component Array** te se nakon toga čisti sam *array*.

Funkcija koja slijedi nakon toga je funkcija **ClearSplinePoints** koja je

Preporuka je imati grafičku podršku kod korištenja VR uređaja koja može isertati sliku minimalno 60 puta u sekundi, a to je ovisno i o softveru koji se pokreće. Obično igrač može pretrpiti i čak i duplo manji broj slika po sekundi (30 FPS). Igre u VR-u su drugačije jer je tada VR sve što igrač u određenom trenutku vidi te igraču može pozlati ako se slika mijenja manje od šezdeset puta u sekundi.

osmišljena da čisti scenske komponente tipa **Spline**. **Arc Spline** upravo je komponenta tipa **Spline** pa se može koristiti kao ulaz u funkciji **ClearSplinePoints**. U detalje te komponente nećemo ulaziti.



Slika 28: Prikaz stvaranja i brisanja luka

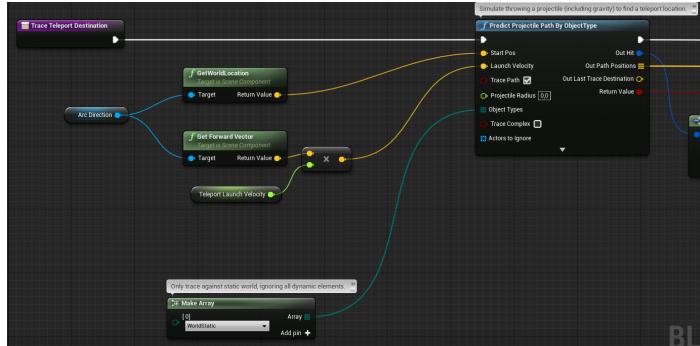
Sljedeća funkcija koja se na *tick* odvija kad je varijabla **IsTeleportActive** istinita (postavlja se na istinu na početku funkcije **ActivateTeleporter** koja se poziva na pritisak tipke za teleportaciju) stvara točke luka indikatora za teleportaciju. Prvo se koristi funkcija **Predict Projectile Path By Object Type** kojoj je potrebno kao argument dati početnu lokaciju. Početna lokacija dobiva se funkcijom **GetWorldLocation** kojoj kao argument dajemo komponentu **ArcSpline**. Spomenuta komponenta nalazi se u sredini *mesh-a* šake tako da će to biti vektor lokacija polazišne točke. Funkciji **PredictProjectilePathByObjectType** potreban je još jedan vektor kao argument, a to je **LaunchVelocity**. Taj **Vector** daje smjer i brzinu od početne točke. Dobivanjem prednjeg vektora funkcijom **GetForwardVector** od komponente **ArcDirection** dobiva se smjer koji se množi varijablom **TeleportLaunchVelocity** kako bi se dobila brzina. Vrijednost varijable **TeleportLaunchVelocity** postavljena je na 900, a tip a je **Float**. Završno je funkciji **Predict Projectile Path By Object Type** dodan argument **ObjectTypes** koji je tipa **EnumArray** i služi funkciji kako bi znala koje elemente svijeta uzima u obzir kod svoje kalkulacije. Funkcija **MakeArray** u slučaju stvaranja potrebnog argumenta za funkciju **Predict Projectile Path By Object Type** ima osnovne kolizijske kanale kao vlastiti argument. Taj argument postavljen je na **WorldStatic** što znači da će se u funkciji **Predict Projectile Path By Object Type** ignorirati svi ostali kolizijski kanali kod izračuna. Možda je još ostalo nejasno kako se od početne točke i njenog smjera i ubrzanja dobije luk, a ne pravac. To je zato što na projektilе bazično djeluje gravitacija vrijednosti 1 što označava uobičajenu zemljinu gravitaciju. Detalji prikazani na slici 29.

Dio koda koji slijedi vezan za funkciju **Project Point To Navigation**. To je jedan od elemenata koji funkcionalnost teleportacije cini dotjeranom. Uz pomoć ovog dijela kôda, a samim time i kôdom koji ga prati, postiže se normalno funkcioniranje teleportacije čak i kada nije pronađena dobra lokacija za teleportaciju. Funkcija **Project Point To Navigation** uzima u mapu postavljeni navigacijski sistem (**NavMeshBounds**) i točku koja ne mora biti unutar samog navigacijskog sustava.

Pomoću varijable **Query Extent** funkcija dobiva argument za koliko je po-

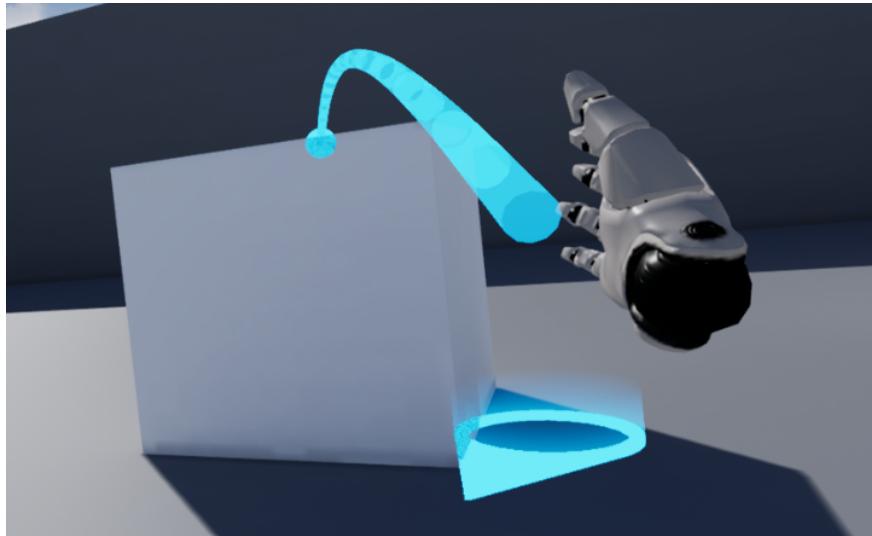
Kako je većina elemenata, tako je i gravitacija, u Unreal Engine-u 4 modularna te se ona može mijenjati za sve elemente svijeta, za pojedine elemente ili kombinirano.

Područje koje zahvaća navigacijski sustav može se lako provjeriti pritiskom na tipku P na tipkovnici dok se nalazimo u *viewportu* koja će aktivirati vizualizaciju nacigajskog sustava koji će prikazati zelenu boju na površini koju obuhvaća.



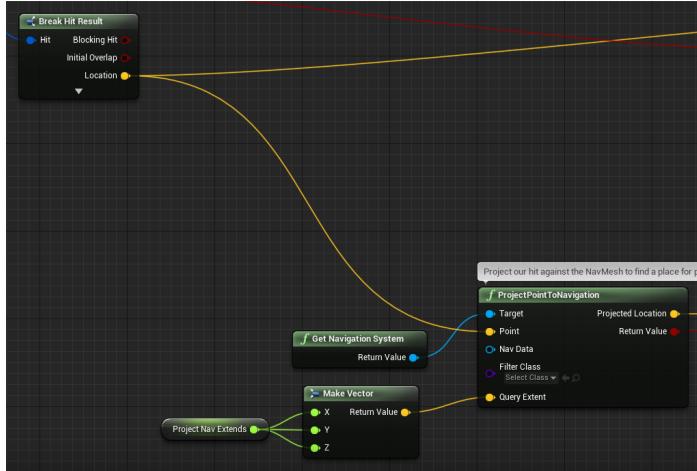
Slika 29: Prikaz izračuna puta projektila

trebno proširiti dobivenu točku u svim smjerovima kako bi dobila preklapanje s navigacijskim sustavom te vraća vektor najbliže lokacije gdje se ekstenzija zadane točke i navigacijskog sustava sijeku. Na ovaj se način osigurava da igrač, čak i ako se pokuša teleportirati u neki statički element na čijem se mjestu ne može nalaziti jer mu to ne dozvoljava navigacijski sustav, teleportacija ipak dogodi na najbliže moguće mjesto od zadanog ulaznim vrijednostima. To se može vidjeti na slici 30 koja pokazuje kamo će se igrač zapravo teleportirati iako njegov luk indikacije mesta teleportacije ne pokazuje na to mjesto.

Slika 30: Prikaz *offseta* inicijalnog mesta teleportacije

Navigacijski sustav dobiva se funkcijom `GetNavigationSystem`, točka kojoj se traži najbliža točka u navigacijskom sustavu dobiva se iz funkcije `BreakHitResult` koja nam omogućava pristup svim mogućim vrijednostima varijable `Hit`. Ta se varijabla dobiva se kao jedna od izlaznih vrijednosti prethodno opisane funkcije `Predict Projectile Path By Object Type`. Iz funkcije `BreakHitResult` uzimamo vektor lokaciju ili točku u kojoj se zamišljeni projektil sudari s podom te je to točka za koju promatramo i

tražimo najbližu ukoliko je to potrebno. Ako se točka nalazi u navigacijskom sustavu, onda će njena najbliža točka biti ta ista. Koliko daleko od zadane točke želimo tražiti navigacijski sustav postavljamo varijablu `QueryExtent` u koju unosimo vektor s vrijednostima X , Y i Z od 500 (5 metara) jer je to vrijednost zadana varijablom `ProjectNavExtends` koja se zadaje kao vrijednost svih triju argumenata u funkciji `MakeVector`.

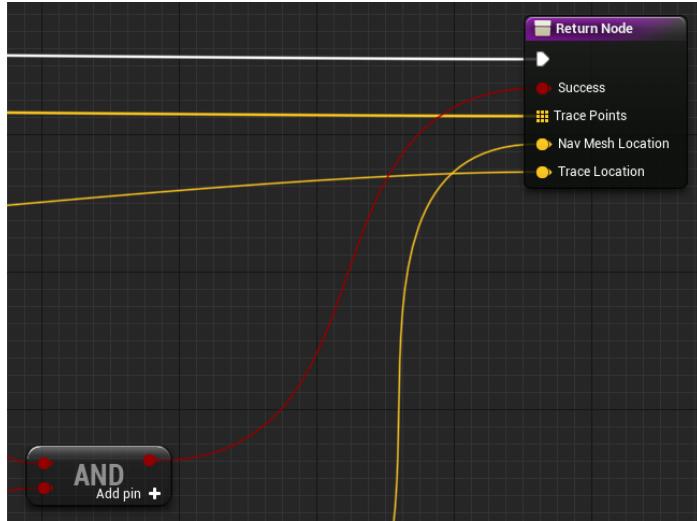


Slika 31: Postavljanje Extenta teleportacije

2.2.5 DOBIVANJE NAVIGACIJSKOG SUSTAVA

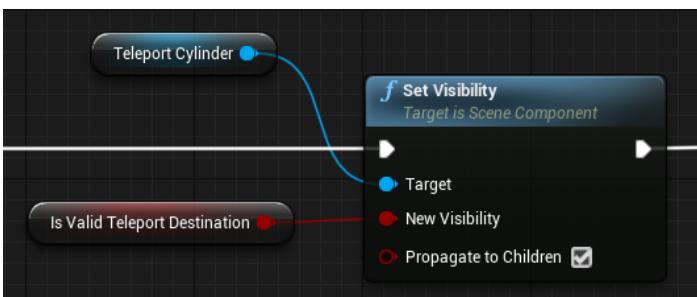
Funkcija `TraceTeleportDestination` završava sa svojim izlaznim čvorom koji sadrži vrijednosti koje implementirana funkcija vraća. Prva varijabla koju funkcija vraća je `Bool` varijabla `Success` koja je istinita ako su funkcije `PredictProjectilePathByObjectType` i `ProjectPointToNavigation` bile uspješne. Ako je bilo koja od njih bila neuspješna, varijabla `Success` biti će lažna. U slučaju da uspješnost funkcije `PredictProjectilePathByObjectType` bude lažna, automatski je lažna i uspješnost funkcije `ProjectPointToNavigation` jer ona nema lokaciju rezultata mjesto kolizije statičnih elemenata i projektila. Logika iza postavljanja istinitosti izlazne varijable `Success` implementirana je u funkciji logičkog operatatora `AND` koji prima vrijednosti o uspješnosti spomenute dvije funkcije. Sljedeća izlazna vrijednost je tipa `VectorArray` koja svoje vrijednosti dobiva izravno iz funkcije `Predict Projectile Path By Object Type` koja vraća sve vektorske pozicije puta simuliranog projektila. Vrijednost izlazne varijable `NavMeshLocation` je vektor tj. najbliža točka od tražene točke zadane korisnikovim unosom (rezultat pod nazivom `ProjectedLocation` opisane funkcije `ProjectPointToNavigation`). Zadnji element je također vektor u koji se zapisuje lokacija mesta sudara projektila i statičnog elementa. Prikaz je na slici 32.

Sada se vraćamo na `EventGraph` ili glavni *blueprint actor* `BP_MotionController` gdje nakon funkcije `TraceTeleportDestination` dolazi jednostavna funk-



Slika 32: Prikaz izlaznih varijabli funkcije TraceTeleportDestination

cija SetVisibility koja postavlja vrijednost vidljivosti TeleportCylinder (strelica) na vrijednost koju ima varijabla IsValidTeleportDestination. Prisjetimo se da je vrijednost te varijable postavljena prilikom izlaza iz funkcije TraceTeleportDestination od njenog izlaza Success što govori da vidljivost TeleportCylidera ovisi o uspješnosti pronalaska valjanog mesta za teleportaciju. To je logično jer ukoliko funkcija TraceTeleportDesetination nije pronašla valjano mjesto za teleportaciju, onda ne želimo ni vizualno osvijestiti igrača o potencijalnom mjestu njegove teleportacije budući da je teleportacija tada nemoguća. Uz pomoć nevidljivosti strelice teleportacije tjeramo igrača da proba neku drugu poziciju na koju se želi teleportirati jer je ona pozicija koju trenutno unosima zadaje nevaljana ili jednostavno nedozvoljena.



Slika 33: Prikaz izračuna puta projektila

Video igre veoma su specifične u pogledu davanja povratnih informacija korisniku ili igraču jer se ono mora učiniti na način da je jasno i razumljivo, ali da igrač i dalje bude u potpunosti uživljen u igru.

2.2.6 TRAŽENJE MJESTA ZA STAJANJE

U funkciji TraceTeleportDestination obrađena je funkcija Predict Projectile Path By Object Type koja je, između ostalog, vraćala i lokaciju sudara simuliranog projektila i statičkog svjetskog elementa (WorldStatic) na koji

se postavlja igrač ukoliko je ona unutar navigacijskog sustava. Kada ta točka nije bila unutar navigacijskog sustava, jednostavno je pronađena prva najbliža točka koja ju je zamjenila. Postavlja se pitanje: što se dogodi u situaciji kada je točka presjeka unutar navigacijskog sustava u sudaru sa zidom ili bilo kojom drugom statičkom površinom koja nije sam pod? Pogledajmo sliku 34.

Nova bi lokacija nakon teleportacije povisila poziciju igrača na onu zadalu tom lokacijom što bi napravilo neželjeni efekt u kojem bi igrač mogao stajati u zraku. Kako bismo tu pojavu spriječili potreban nam je sljedeći dio kôda koji zapravo u navigacijskom sustavu traži pod. Funkcija koja nam u ovom slučaju vraća točnu poziciju poda je funkcija `LineTraceObjects`. Njoj moramo zadati vektorski raspon u kojem želimo da pronađe elemente određenog kolizijskog tipa. Kao početna točka zadaje se ona točka u navigacijskom sustavu pronađena u funkciji `TraceTeleportDestination`, a kao krajnja točka zadaje se ista ta točka spuštena po *Z* osi (visina) za 200 (2 metra). U prostoru između zadane dvije točke sigurno se nalazi točka poda koju onda funkcija `LineTraceForObjects` pronalazi. Ono što traži zadano je poljem koje sadrži kao argument samo statičke kolizijske elemente između kojih je i pod. Kada se pronađe točka u kojoj zadani vektor dođiruje pod, ona se vraća kao `Hit` rezultat u kojem ponovo preko funkcije `BreakHitResult` uzimamo tu lokaciju tj. vektor pod nazivom `ImpactPoint`.



Slika 34: Traženje mesta za "stajanje"

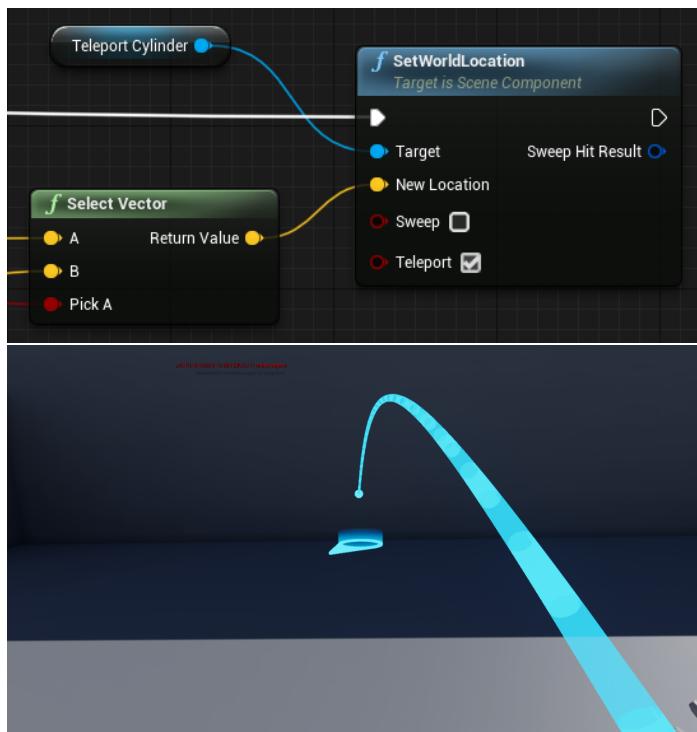
Navigacijski sustav svojim obujmom mora zahvatiti dio prostora iznad statičnog elementa mape kako bi ga označio kao valjni prostor navigacijskog sustava. Najbliža točka sudara unutar sustava je obično pri najvišoj vrijednosti visine (*Z* os) obujma navigacijskog sustava, no to nije točka poda.

2.2.7 ŠTO AKO SE NE MOŽE NAĆI POD?

Prije postavljanja lokacije `TeleportCylinder` odradjuje se još jedna funkcija `SelectVector` koja odabire vektor odnosno točku koju će funkcija `SetWorldLocation` uzeti kao argument. Funkcija `SelectVector` zapravo nije potrebna jer će se igrač uvijek moći kretati samo unutar navigacijskog sustava što znači da će funkcija `Line Trace For Objects` uvijek pronaći pod. Iz tog će razloga spomenuta funkcija uvijek vratiti određeni rezultat funkciji `BreakVector` što znači da će izlazna varijabla `BlockingHit` uvijek biti istinita. U funkciji `SelectVector` će se tada uvijek odabrati onaj vektor dobiven nakon funkcije `LineTraceForObjects`, a nikada onaj koji dolazi

direktno iz funkcije `TraceTeleportDestination`.

Zašto onda ovdje imamo funkciju `SelectVector`? Ona je sigurnosni mehanizam za slučaj kada funkcija `LineTraceForObjects` ne bi pronašla pod što je u teoriji nemoguće. Dobivena lokacija ulazi kao argument u funkciju `SetWorldLocation` koja postavlja lokaciju komponente `TeleportCylinder` na unesenu točku. Ovo je važno jer se u opisanoj funkciji `GetTeleportDestination` upravo uzima lokacija `TeleportCylinder-a` kao jedan od parametara određivanja vektora teleportacije. Detaljan prikaz na slici 35



Slika 35: Traženje poda

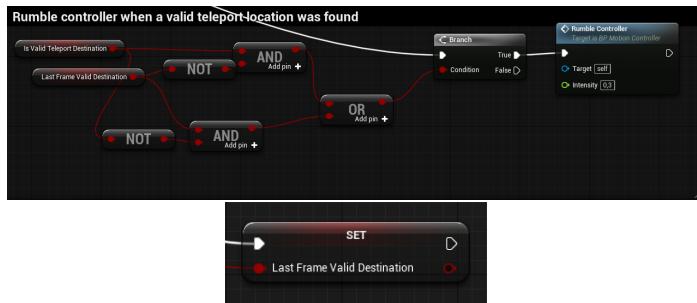
Sekvenca kôda dalje vodi na sljedeći dio kôda koji je označen komentarom *Rumble controller when a valid teleport location was found.* Ovdje se provjeravaju stanja dviju Bool varijabala od kojih je poznata `IsValidTeleportDestination`. Druga varijabla je `LastFrameValidDestination` koja nije ništa drugo nego vrijednost varijable `IsValidTeleportDestination` u prošlom *ticku*. Ona se postavlja u nastavku tako da u uvom dijelu kôda imamo uvijek ažurnu varijablu `IsValidTeleportDestination` i za *tick* zastarjelu varijablu `LastFrameValidDestination`. Ovdje se logički provjerava je li došlo do promjene vrijednosti varijable `IsValidTeleportDestination`. Ako je došlo do promjene vrijednosti, vrijednost spomenutih Bool varijabli bit će različita.

Ako je trenutno valjano mjesto teleportacije a to nije bilo prošli tren ili, ako nije valjano mjesto teleportacije a to je bilo prošli tren, pokreni funkciju `RumbleController`.

Spomenuta varijabla `LastFrameValidDestination` postavlja se kao sljedeći

element u sekvenci kôda na izlaznu vrijednost funkcije `TraceTeleportDestination`.

Postavljanje varijable `LastFrameValidDestination` može se vidjeti na slici 36.

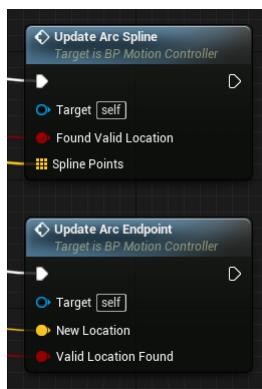


Slika 36: Vibracije na prijelazu stanja mogućnosti teleportacije

Nakon postavljanja spomenute varijable slijede dvije funkcije:

- `UpdateArcSpline` i
- `UpdateArcEndpoint`

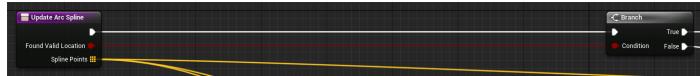
Funkcija `UpdateArcSpline` dobiva argument `SplinePoints` koji je tipa `VectorArray` i on je jedna od izlaznih vrijednosti funkcije `Trace Teleport Destination`. U ovom se polju nalaze sve točke koje čine luk indikator udaljenosti mjesta teleportacije. Funkcija `UpdateArcEndpoint` prima jedan vektorski argument koji je lokacija sudara simulacije projektila i statičnog svjetskog elementa (`WorldStatic`). Obje funkcije kao argument dobivaju i `Bool` varijablu iz funkcije `TraceTeleportDestination` tj. vrijednost njene izlazne varijable `Success`.



U Unreal Engineu 4 ne moraju biti pridruženi svi argumenti funkcijama ako funkcija može funkcionirati i bez njih. Iz tog razloga nije uvijek potrebno pridružiti sve moguće ulazne argumente određenoj funkciji kao što je to ovdje slučaj, no ovoj su funkciji za normalan rad potrebni sve argumenti. U svijetu paradigme objektno orijentiranog programiranja to se svojstvo zove i *function overload*

Slika 37: Pozivanje ažuriranja luka i točke teleportacije

Proučimo detaljnije funkciju `UpdateArcSpline` koja na *tick* ažurira točke `Spline` elementa. Ako je uspješno pronađena lokacija na koju će se igrač teleportirati, jedan dio kôda se preskače. Uvjet je ulazna varijabla pod nazivom `FoundValidLocation`. Slika 38 prikazuje ulazni čvor funkcije kao i prvi *branch* koji ovisno o uvjetu provodi ili preskače dio kôda.

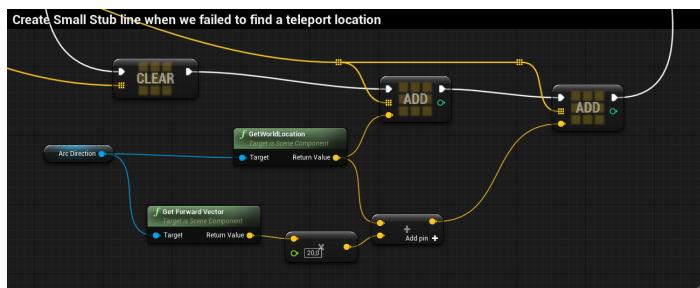


Slika 38: Ulaz u funkciju `UpdateArcSpline` i provjeravanje uvjeta

Dio kôda koji se provodi u slučaju da nije pronađeno valjano mjesto za teleportaciju nalazi se pod komentarom *Create Small Stub line when we failed to find a teleport location.* Prvo se čisti dobiveni `VectorArray` u funkciji `Clear`. U prazni `VectorArray` dodaje se jedan vektor funkcijom `Add` koji predstavlja lokaciju na kojoj se nalazi `ArcDirection` komponenta ili lokaciju odakle želimo da se prikazuje `Spline`. Komponenta `ArcDirection` komponenta je tipa `Arrow`, a ona je prazna komponenta koja ima smjer i koristi se upravo u svrhe zadavanja vektora za korištenje koji nije direktno vezan za neku drugu vidljivu komponentu.

Lokaciju `ArcDirection` komponente dobivamo pomoću funkcije `GetWorldLocation`. Kako bismo prikazali vidljivu liniju, potrebna nam je još barem jedna točka. Tu točku dobit ćemo iz lokacije početne točke i prednjeg vektora točke u kojoj se nalazi `ArcDirection`. Iz funkcije `GetWorldLocation` dobiva se vektor koji se zbraja s prednjim vektorom `ArcDirection` komponente pomnoženim s 20. Prednji vektor dobiva se funkcijom `GetForwardVector`, a njegovo množenje s 20 zapravo pomiče vektor, odnosno točku za 20 što su 2 decimetra. Dobivena se točka funkcijom `Add` dodaje u `VectorArray` koji sada ima 2 elementa. Prikaz na slici 39

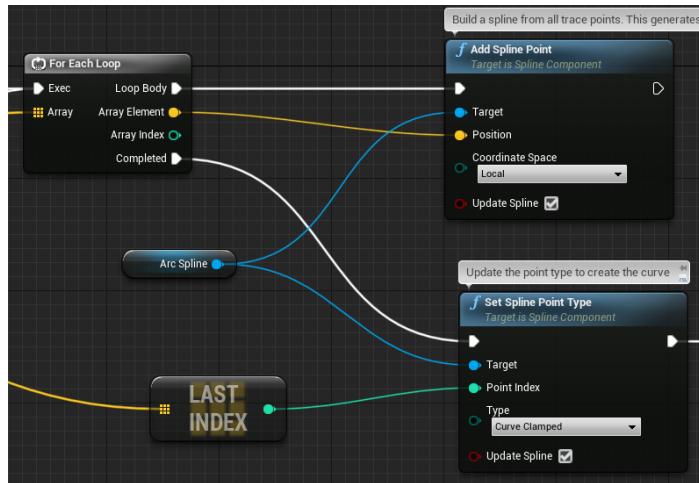
Lokaciju i usmjerenje komponente tipa `Arrow` možemo lako provjeriti u testnom scenaru ili u *viewportu* na način da joj u opcijama tj. u *Details Panelu* pod kategorijom *Rendering* kvačicom označimo opciju `Visible`



Slika 39: Stvaranje prikaza kad teleportacija nije moguća

Točke `Spline` komponente dodaju se u petlji `ForEach` koja za svaki element u polju vektora prolazi kroz funkciju `AddSplinePoint` koja dodaje točku `Spline` komponente. Komponenta na koju se dodaje `Spline Point` je `ArcSpline`, a pozicija `Spline Pointa` je pozicija elementa iz polja vektora. Polje vektora može sadržavati veći broj elemenata ukoliko je pronađena valjana lokacija za teleportaciju ili samo dva elementa ukoliko valjana lokacija nije pronađena. Bez obzira na sadržaj polja kroz funkciju `AddSplinePoint` se prolazi onoliko puta koliko elemenata polje sadrži. Kad je petlja završila s postavljanjem `Spline Pointova` izvršava se funkcija `SetSplinePointType` koja prema indeksu zadnjeg elementa polja postavlja tip `Spline Pointa` na

CurveClamped. Vizualizacija lûka radi na isti način s ili bez nje. Promjena tipa pod Type kao i ulaznog indeksa ne mijenja ponašanje vizualizacije luka. Komentar iznad funkcije govori da se ova funkcija koristi kako bi se promjenio tip točke i stvori krivulja. Ovdje se tip točke mijenja samo na zadnjoj točki pa nije jasno kako to utječe na tip točke ostalih točaka. Krivulja, odnosno lûk stvara se i bez korištenja funkcije `SetSplinePointType` što je i logično budući da točke luka već čine krivulju.



Slika 40: Izgradnja splajna iz točaka

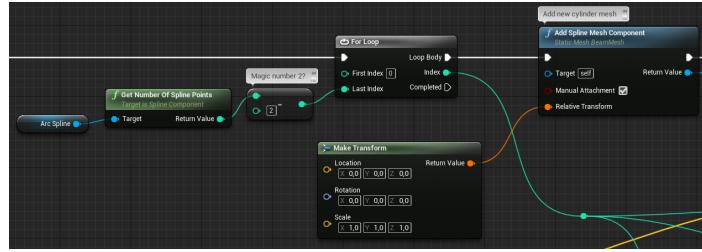
Postavljenje **Spline Pointova** nije dovoljan da bi se video lûk koji se inače vidi kod teleportacije. Potrebno je svakom **Spline Pointu** dodijeliti *mesh* što se radi u petlji. Kroz petlju se prolazi onoliku puta koliki je broj **Spline Pointova** umanjen za dva jer ne želimo uzeti početnu i završnu točku. Broj **Spline Pointova** dobivamo funkcijom `GetNumberOfSplinePoints` kojoj kao argument dajemo **Spline** element za kojeg ispitujemo broj elemenata. Taj se broj nakon oduzimanja 2 daje kao zadnji indeks u petlju ili broj puta koliko će se kroz petlju proći. Funkcijom `AddSplineMeshComponent` dodaju se *mesh* komponente svakom **Spline Pointu**. Argument `RelativeTransform` nije obavezan za funkciju `AddSplineMeshComponent` pa joj zapravo nije potrebno proslijediti varijablu tipa `Transform`. Razlog tomu je kasnije zadavanje početka i završetka svake *spline mesh* komponente.

Postavlja se pitanje – može li se kôd kompajlirati ako se izbriše funkcija `MakeTransform`?

Svaki **SplineMeshComponent** funkcijom `Add` dodaje se u polje **Spline Meshes** kako bismo imali referencu na svaki **SplineMeshComponent** te ga mogli izbrisati ili mu postaviti drugu poziciju. Nakon toga slijedi funkcija `SetStartAndEnd` koja se također odvija u svakom prolasku kroz prethodno spomenutu petlju. Ona zahtijeva **SplineMesh** komponentu kojoj je cilj postaviti početak i kraj te početnu i završnu lokaciju kao njihove vektorske tangente. Vektorske tangente su funkciji potrebne kako bi se mogao odrediti nagib *Spline Mesha*.

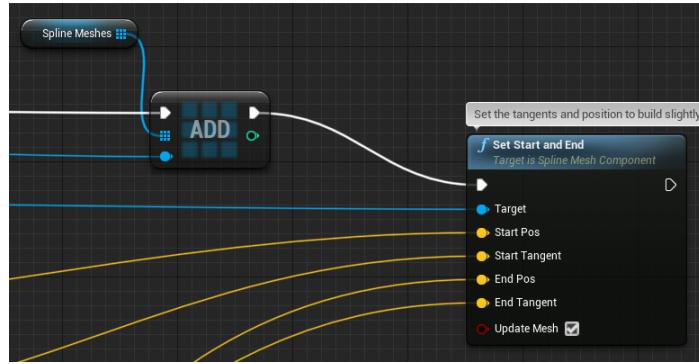
Svaki element na kojeg se planira utjecati ili uništiti pod specifičnim uvjetom, a stvoreni je prilikom izvodjena programa, mora imati referencu jer inače tom elementu nemamo kako pristupiti. Referenca se stavlja uvođenjem varijable istog tipa kao što je objekt ili varijabla koju želimo referencirati te postavljanjem njene vrijednosti u izlazu određene funkcije gdje ona nastaje.

Tangenta je pravac koji krivulju dodiruje samo u jednoj točki



Slika 41: Dodavanje novog *Cylinder mesha* na svaku **spline** komponentu

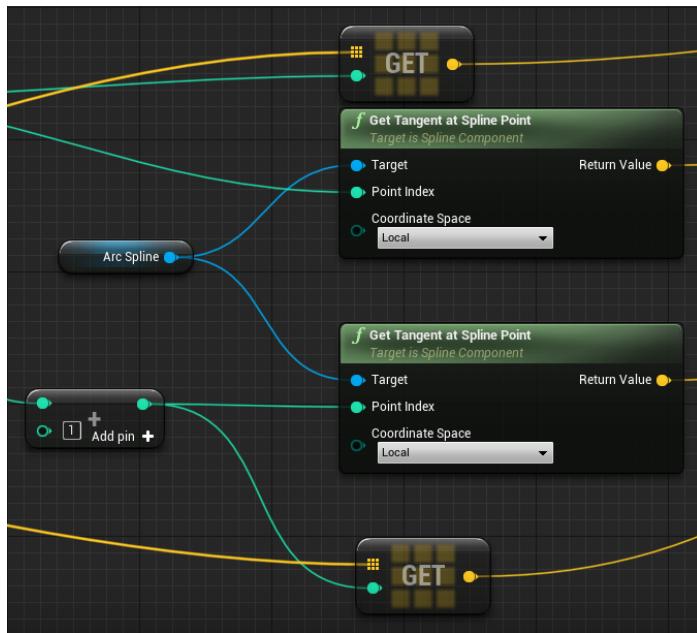
Detalji na slici 42.



Slika 42: Funkcija koja stvara cilindre na svakom splajnu

Ostalo je još prikazati dio kôda u kojem se dobivaju ulazne vrijednosti za funkciju **SetStartAndEnd**. Lokaciju trenutnog **Spline Pointa** dobivamo u funkciji **Get** gdje se iz vektorskog polja koje sadrži sve vektore iz funkcije **TraceTeleportDestination** uzima onaj vektor pod indeksom trenutnog prolaska **for** petlje umanjenog za jedan. Dobiveni vektor je početni vektor koji je argument **StartPos** u funkciji **SetStartAndEnd**. Na isti se način uzima sljedeći element u polju jednostavnim inkrementiranjem indeksa koji se potom šalje kao argument u funkciju **Get** koja dobiva podatke iz istog vektorskog polja. Ovaj vektor je argument **EndPos** za funkciju **SetStartAndEnd**. Tangente se dobivaju pomoću funkcije **GetTangentAtSplinePoint**. Njoj je potreban **Spline** nad kojim se želi pronaći tangenta te indeks njene specifične točke nad kojom se želi pronaći tangenta. Način dobivanja indeksa isti je kao i kod funkcije **Get** pa te iste indekse uzimamo kao potrebne argumente. Dobiveni rezultati redom su **StartTangent** i **EndTangent** koji su potrebni funkciji **SetStartAndEnd**. Detalji na slici 43.

Zadnja funkcija koja nas zanima iz *actora* **BP_MotionController** je funkcija **UpdateArcEndpoint**. U njoj se prvo postavlja vidljivost komponente **ArcEndPoint** na istinu, ako je pronađeno valjano mjesto za teleportaciju i ako je teleporter trenutno aktivan (pritisnuta je glijica i igrač bira mjesto teleportacije). Vidljivost se postavlja u funkciji **SetVisibility**, a tu funkciju prati funkcija **SetWorldLocation** koja postavlja lokaciju komponente

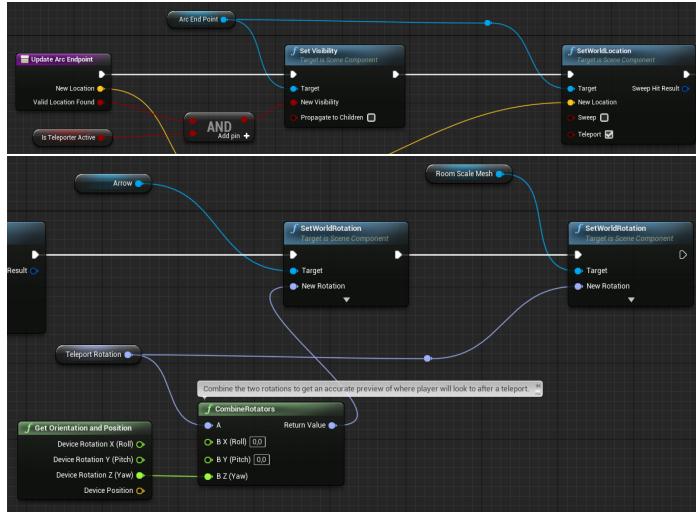


Slika 43: Postavljanje ulaznih vrijednosti za `SetStartAndEnd`

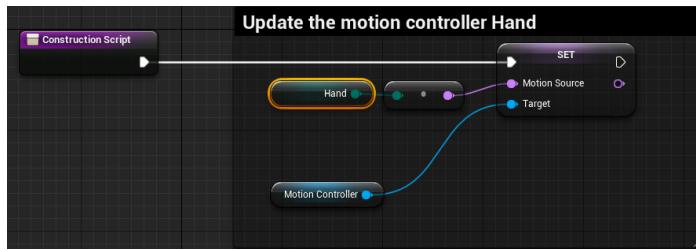
`ArcEndPoint` na lokaciju gdje se simulirani projektil sudario sa statičnim elementom. Zatim se postavlja rotacija strelice funkcijom `SetWorldRotation`. Rotator se dobiva iz funkcije `CombineRotators` koja kombinira `Rotator TelerpotRotation` i `Rotator` koji se dobiva funkcijom `GetOrientationAndPosition`. Funkcija `GetOrientationAndPosition` vraća lokaciju i rotaciju HMD-a no ovdje se uzima samo rotacija i to specifično samo rotacija po *Z* ili rotacija po *X* i *Y* osi. Ovo postavlja rotaciju strelice na način da je naprijed na gljivici uvijek naprijed za igrača te, ako nije ništa pritisnuto, pokazuje se rotator samo od HMD-a jer drugom `Rotatoru` nije pridružena vrijednost. Na kraju se u funkciji `SetWorldRotation` postavlja rotacija komponente `RoomScaleMesh` koja je za nas nevažna jer koristimo Oculusov Guardian, a ne SteamVR verziju zaštite. Rotacija ove komponente također se mora ažurirati kako bi ostala relativno ista sa stvarnim prostorom u kojem se nalazi igrač.

Važno je još samo spomenuti da se referenca na animaciju šake postavlja u konstruktoru `actora BP_MotionController`. To se može vidjeti na slici 45.

Oculusov sustav zaštite `Guardian` izvršava se u samom HMD-u, dok se SteamVR zaštita odvijaja na razini igre



Slika 44: Funkcija UpdateArcEndPoint



Slika 45: Referenciranje na animaciju šake

2.3 ZAKLJUČAK

Kroz ovu lekciju pokazali smo kako funkciraju osnovne funkcionalnosti kretanja u prostoru pomoću VR uređaja te kontrolera. Ove prepostavljene funkcionalnosti napravljene su tako da većina VR uređaja radi na *plug'n'play* način te tako da se ne moraju stalno iznova programirati osnovne stvari.

Kao dopunu ove lekcije korisno bi bilo i istražiti koncepte:

- *light-house* praćenje i *inside-out* praćenje
- *room-scale*
- praćenje rotacije zapešća za smjer teleportacije umjesto gljivice – po nekad neprirodno, ali korisno za istražiti

Dodatak za čitatelja je da pokuša natjerati Unreal Engine da ne projicira mjesto teleportacije na pod kad se teleportacija usmjeri negdje iznad poda. Pokušajte se teleportirati u zrak.

Prostor za bilješke:

3 KRETANJE

U ovom poglavlju glavni nam je cilj uspostaviti micanje i sve povezane koncepte. Za početak ćemo krenuti s micanjem igrača. Igrač se u igri mora moći kretati sukladno s naredbama kontrolera te zbog toga je potrebno implementirati odgovarajuće funkcionalnosti. Važno je i obratiti pozornost na koliziju s drugim objektima.

U svrhu kretanja igrača potrebno je implementirati funkcionalnosti u skladu sa zakonima fizike. S obzirom na to da ne možemo direktno utjecati na kolizije igrača tj. `PlayArea` kreirali smo novu kapsulu na koju ćemo primijeniti svu fiziku te će nam u igri služiti kao `CollisionBox`. Drugim riječima, pomoću navedene kapsule provjeravat ćemo je li došlo do kolizije između igrača i ostalih elemenata igre poput poda, zida itd. Ovakvim pristupom kretanju igrač se ne miče izravno, nego se pomiče kapsula, a igrača postavljamo u sredinu te kapsule.

Naravno, potrebno je i konfigurirati postavke kapsule pa smo tako postavili njenu visinu (`Capsule Half Height`), radius kapsule (`Capsule Radius`), uključili opciju simuliranja fizike (`Simulate Physics`) te smo dodali igraču nekoliko tisuća kilograma. Ova vrlo zanimljiva dodjela vrijednosti je zapravo vrlo korisna. Naime, prilikom testiranja igre došli smo do saznanja da ukoliko se sudarimo s drugim objektom, doći će do greške koja uzrokuje neočekivano ponašanje, tj. naš će igrač odletjeti visoko u nebo. Povećanjem mase igrača na nekoliko tisuća kilograma on će postati vrlo težak i na taj će ga način biti nemoguće podići s tla.

Postavke	Vrijednost
Transform	
Location	Z: 90
Shape	
CapsuleHalfHeight	90
CapsuleRadius	30
Physics	
Simulate Physics	True
Mass in kg	1 000 000
Linear Damping	10

Tablica 1: Prikaz promjena postavki u Details panelu komponente `Capsule Collision`

Opcija `Linear Damping` (otpor u svim smjerovima) odgovorna je za padaće i odskakivanje. Što je ta opcija veća, postojat će veći otpor pa ćemo

iz tog razloga sprije padati i imati veće trenje. Kako bismo prilagodili fiziku našim potrebama moramo kreirati jedan **Physics Material**. To odradujemo desnim klikom na praznom mjestu u *content browseru* pod opcijom *Physics > Physics Material*. Postavke tog materijala mogu se vidjeti u tablici 2.

Postavke	Vrijednost
Physical Material	
Restitution	0

Tablica 2: Prikaz promijenjenih opcija u *Physical Materialu*

Opcija *Restitution* predstavlja vrijednost odbijanja – koliko će jako ugrađena fizika djelovati na prijenos energije, veći broj znači više odbijanja, a vrijednost 1 predstavlja ”normalno” odbijanje.

U **Collision** dijelu *details panela collision* kapsule sada možemo u opciji **Phys Material Override** kapsuli možemo pridružiti fizički materijal.

Nakon toga, opciju **Collision Presets** smo postavili na **Custom** zato što želimo sami konfigurirati određene opcije. Prvo ćemo promijeniti određene opcije. Opcije **WorldStatic** i **WorldDynamic** predstavljaju *collision channels* koji su dodijeljeni objektima koji mogu biti u *viewportu*. **WorldStatic** je **CollisionChannel** koji se postavlja svim *actorima* koji se neće micati u svijetu na bilo koji način, poput zida ili tla, dok **WorldDynamic** opcijom prikazujemo strukture koje se mogu kretati, mijenjati i sl. Ako je na određenu strukturu primjenjiva i fizika, tada će i opcija **PhysicsBody** biti uključena.

Nakon toga želimo ignorirati određene dijelove (stupac **Ignore**) pa tako ignoriramo opciju **Pawn** koja zapravo predstavlja našeg igrača što bi značilo da nema nikakve direktne interakcije s igračem, nego ćemo igrača postaviti unutar kapsule. Nakon toga, ignoriramo opcije **Vehicle** i **Destructable** jer nisu prisutne u našem projektu. Isto tako, opcije **Visibility** i **Camera** su ignorirane. Izmjene možete vidjeti u tablici 3.

Opcije **Visibility** i **Camera** nisu toliko važne za naš projekt jer nemaju utjecaj na igračovo iskustvo igranja. Te su opcije puno važnije i značajnije za igre koje se igraju iz trećeg lica.

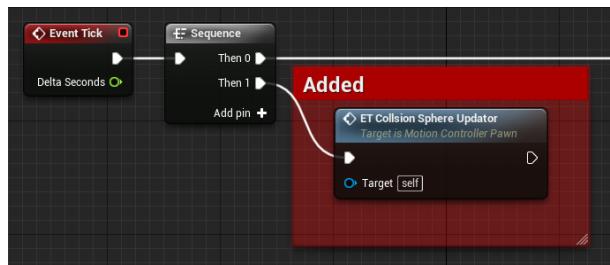
Postavke	Vrijednost
Collision	
Phys Material Override	PM_NoBounce (kreirani materijal)
Collision Presets	Custom
Collision Enabled	Collision Enabled (Query and Physics)
Object Type	Pawn
TRACE RESPONSES	
Visibility	Ignore
Camera	Ignore
OBJECT RESPONSES	
WorldStatic	Block
WorldDynamic	Block
Pawn	Ignore
PhysicsBody	Block
Vehicle	Ignore
Destructible	Ignore

Tablica 3: Prikaz dodatnih promijenjenih opcija u komponenti *Collision capsule*

3.1 ROOMSCALE KRETANJE

Trenutno se nalazimo u *actoru MotionControllerPawn* u njegovom *event graphu*. U ovom dijelu objasnit ćemo događaj (eng. *event*) **ET Collision Sphere Updater**. Prije nego što predemo na objašnjavanje događaja potrebno je znati da se navedeni događaj događa na svaki tick igre (o navedenom pojmu *tick* smo pričali u prijašnjim poglavljima).

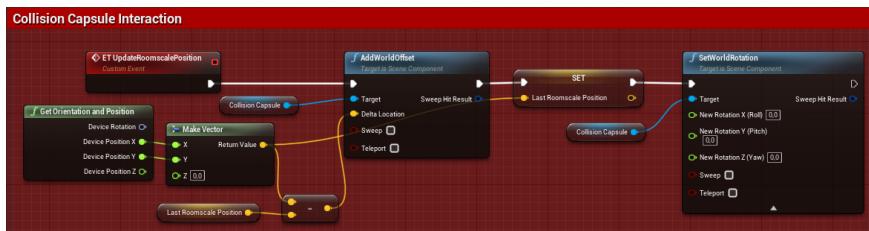
Prije mogućnosti pozivanja spomenutog *eventa* potrebno ga je stvoriti na način da se pritisne desni klik na prazni prostor, upiše se *add custom event* u tražilicu te se jednostavno kreira događaj te se nazove po želji ili po primjeru. To vrijedi i za svaki *custom event* koji će se stvoriti u projektu.



Slika 46: Prikaz poziva ET Collision Sphere Updatera

ET Collision Sphere Updater je tzv. *Custom Event* u kojem se vrši ažuriranje pozicije igrača i kapsule, pri čemu se pozivaju tri dodatna događaja,

`ETUpdateRoomscalePosition`, `ETUpdateActorPosition` i `MovementCheck`. U događaju `ETUpdateRoomscalePosition` se vrši ažuriranje kapsule na način da pozovemo funkciju `GetOrientationAndPosition` u kojoj ignoriramo orijentaciju, a dohvaćamo poziciju *headseta*, tj. VR naočala, i to samo *X* i *Y* koordinate gdje pomoću funkcije `Make Vector` kreiramo novi vektor. Nakon toga uzimamo vrijednosti iz varijable `LastRoomscalePosition` u kojoj je zapisana pozicija u kojoj je igrač bio tren prije nego što se premjestio na trenutnu lokaciju. Oduzimanjem vektora i vrijednosti zapisane u varijabli `LastRoomscalePosition` dobije se novi rezultat koji se prosljeđuje funkciji `AddWorldOffset`. U Unreal Engineu navedena funkcija služi za dodavanje vektora kretanja na određenu poziciju i na taj način se ostvaruje kretanje. Nakon `AddWorldOffset` postavlja se vrijednost varijable `LastRoomScalePosition` preko `Make Vectora`. Funkcija `SetWorldRotation` služi da se igrač, nakon što se zabije u objekt, ne prevrne, tj. postavlja rotaciju na nulu (što se događa na svaki tick) te tako zapravo onemogućuje prevrtanje igrača.



Slika 47: Događaj ET UpdateRoomScalePosition

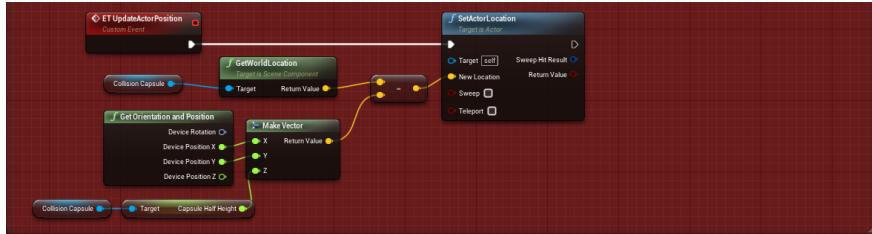
Druga stvar koju radimo u ovom dijelu je postavljanje igrača na mjesto kapsule koje vršimo u događaju `ETUpdateActorPosition`. Jednostavno rečeno, postavljanje igrača svodi se na to da se igrač postavlja na novu poziciju na kojoj se nalazi kapsula. Dakle, funkcijom `GetWorldLocation` dohvaćamo lokaciju kapsule koja je zapisana u objektu `CollisionCapsule`. Funkcijom `GetOrientationAndPosition` dohvaćamo *X* i *Y* poziciju *headseta* koje se prosljeđuju funkciji `MakeVector`. Navedenoj funkciji ručno se postavlja *Z* vrijednost koja je opet dohvaćena iz varijable `CollisionCapsule`. Nakon toga se oduzima lokacija kapsule kreiranim vektorom te se pokreće funkcija `SetActorLocation`. Navedena funkcija postavlja *actora* na definiranu lokaciju. Vidi sliku 48.

3.2 MOVEMENT CHECK

Cijelu priču oko kretanja igrača započet ćemo provjerom je li igraču uopće dozvoljeno kretanje.

S obzirom na to da znamo da ćemo okidanje pokreta vršiti na pritisak tipke kada će se istovremeno zabraniti kretanje igrača, potrebno je napraviti

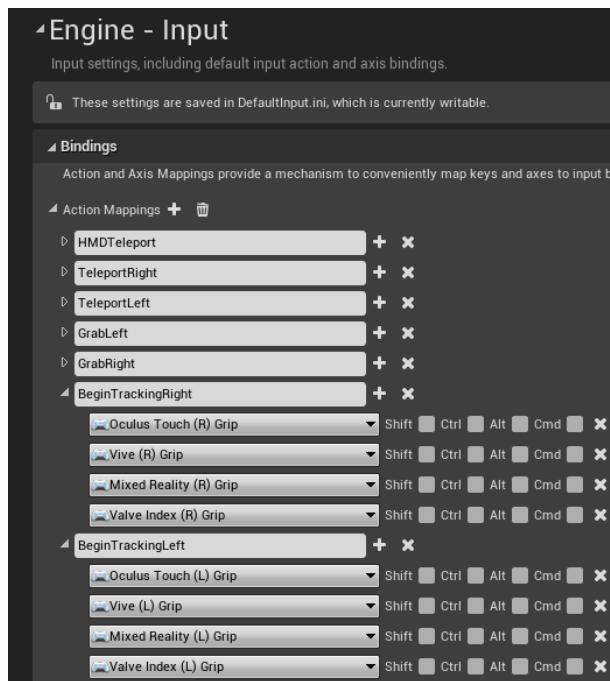
Dodatan zadatak za čitatelja:
pokušajte se zabiti u zid
bez uključene funkcije
`SetWorldRotation`.



Slika 48: Događaj ET UpdateActorPosition

preduvjet na kojem ćemo kasnije graditi okidanje na pokret, a sada ga stavljamo isključivo radi provjere.

Prvo je potrebno postaviti *input actione*. Pod *Edit > Project Settings*, u *Engine* odjeljku treba odabratи *Input* opciju i zatim pod *Action Mappings* kliknuti na plus, imenovati *input action*, proširiti ga i odabratи *inputs* za kontroler. Grip tipka se nalazi na VR kontroleru s bočne strane desnog, odnosno lijevog kontrolera. Dodatni *action mappingi* mogu se vidjeti na slici 49.



Slika 49: Dodane postavke unutar Project Settings

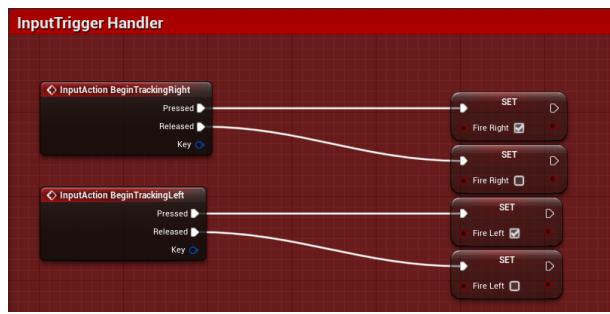
Nakon dodavanja *inputa* možemo ih pozvati kao događaje u kôdu. Na taj se način pozivaju dva događaja:

- `BeginTrackingLeft` i
- `BeginTrackingRight`

Događaje dodajemo tako da ih pozovemo u *blueprintu* i na pritisak lijeve

tipke *grip* postavljamo vrijednost varijable `FireLeft` na `True`. Na puštanje postavljamo vrijednost varijable `FireLeft` na `False`, a analogno vrijedi i za varijablu `FireRight` na događaju `BeginTrackingRight`.

Varijable dodajemo u lijevom donjem kutu pod sekcijom *variables* pritiskom na `+`. Postavke varijable postavljamo u gornjem desnom kutu u sekciji *Details*. Detaljnije o varijablama možete istražiti sami, a ovdje ćemo samo spomenuti da su nam važne komponente *Variable name* pomoću kojeg ćemo postavljati i dohvaćati vrijednosti te komponente i *Variable Type* koji govori kojeg je ta varijable tipa. Za ručno postavljanje pretpostavljene vrijednosti potrebno je nakon dodavanja varijable provesti *compile* koji je prva opcija u *toolbaru*. Prikaz na slici 50



Slika 50: Postavljanje varijabli na `FireLeft` i `FireRight`

Prva funkcija koja se izvodi u događaju `MovementCheck` je `CheckPlayerMovement`.

Na početku varijable `FireRight` i `FireLeft` označuju je li pritisnuta *grip* tipka na kontrolerima. Nakon toga, u provjeru uključujemo varijablu `CanPlayerMoveOrRotate`. Ova se varijabla postavlja kako bi se spriječilo trenutno micanje ili rotacija nakon teleportacije, a povezat će se na kôd *preseta*. Varijablu `CanPlayerMoveOrRotate` postavljamo na `False` na događaj `InputAction TeleportRight` i `InputAction TeleportLeft` na `Pressed` kao prva funkcija u nizu. To se može vidjeti na slici 51.



Slika 51: Postavljanje vrijednosti varijable `CanPlayerMoveOrRotate`

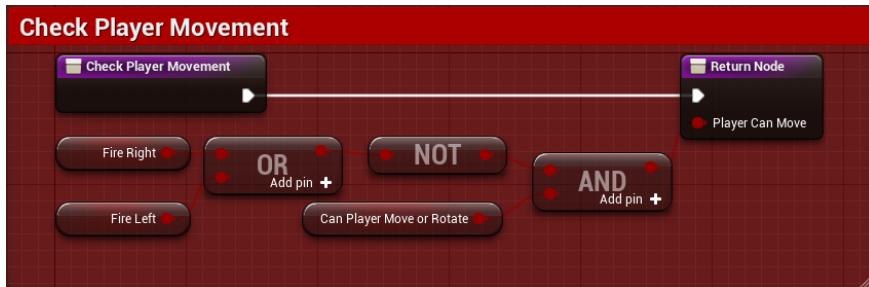
Varijabla `CanPlayerMoveOrRotate` postavlja se na `True` kao zadnji element čvora koji se okida na događaj `InputAction TeleportLeft` i `InputAction TeleportRight`. Prije njega funkcijom `Delay` odgađamo postavljanje te varijable za 0.3 sekunde. To služi kako se ne bismo mogli nakon teleportacije isti tren kretati ili rotirati. Ovo je pokazano na slici 52.

Postavljanjem te varijable na `True` vrijednost, nakon aktivacije teleporta, dolazi do zabrane micanja, odnosno rotacije i teleportacije. Da rezimiramo,



Slika 52: Postavljanje vrijednosti varijable `CanPlayerMoveOrRotate` nakon odgode (*delaya*)

ukoliko nije pritisnuta *grip* tipka i ukoliko igrač nije usred procesa telepor-tacije, igraču je dozvoljeno kretanje. Slika 53 vizualno opisuje funkciju.



Slika 53: Provjera kretanja igrača

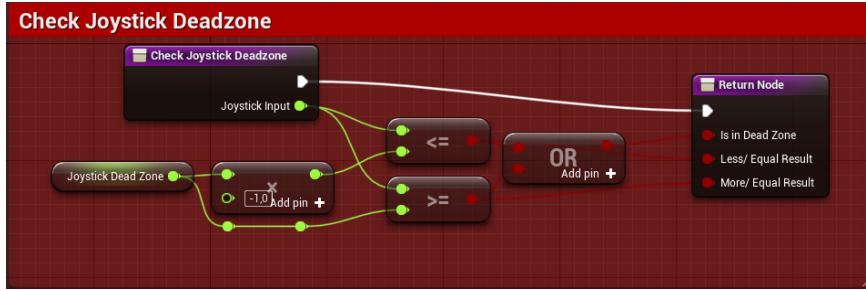
Nakon inicijalnih provjera, ukoliko je igraču dozvoljeno kretanje, prelazimo na sljedeći dio koji se tiče provjere tzv. „mrtvih zona“. U funkciji `CheckJoystickDeadZone` provjeravamo nalazi li se gljivica u „mrtvoj zoni“ čiji smo kraj postavili na 40% od cijelog hoda tipke u bilo kojem smjeru (vrijednost zapisana u varijabli `JoystickDeadZone`). Kako bismo dodatno pojasnili, varijabla `JoystickDeadZone` spremi postotak hoda u kojem od sredine gljivice kontrolera do njezinog ruba želimo da se ništa ne događa kada je pomaknemo. Vrijednosti se kreću od 0 do 1 (u postotcima 0% i 100%), dakle prema gore na *y* osi i desno po *x* osi, te od 0 do -1 (u postotcima 0% i -100%) prema dolje na *y* osi i lijevo po *x* osi. Na slici 55 vidi se prazan hod kontrolera.

Funkcija `CheckJoystickDeadzone` dobiva ulazne varijable preko funkcija koje će se opisati u nastavku.

Ona provjerava je li vrijednost ulaza manja ili jednaka, ili veća ili jednaka od postavljene vrijednosti u varijabli `JoystickDeadzone`. Spomenuta varijabla kod ispitivanja manje ili jednako množi se s -1 jer vrijednosti položaja gljivice poprimaju negativne vrijednosti. Ukoliko je jedan od uvjeta zadovoljen, vrijednost izlazne varijable `IsInDeadZone` postaje `True`, a inače je `False`. Izlazne varijable `Less/Equal Result` i `More/Equal Result` za kretanje nisu važne, no one se koriste kod provjere rotacije, a poprimaju vrijednosti navedenih usporedaba prije logičke operacije `Or`.

Ulaze u funkciju `CheckJoystickDeadzone` dobivamo funkcijama `Get Motion`

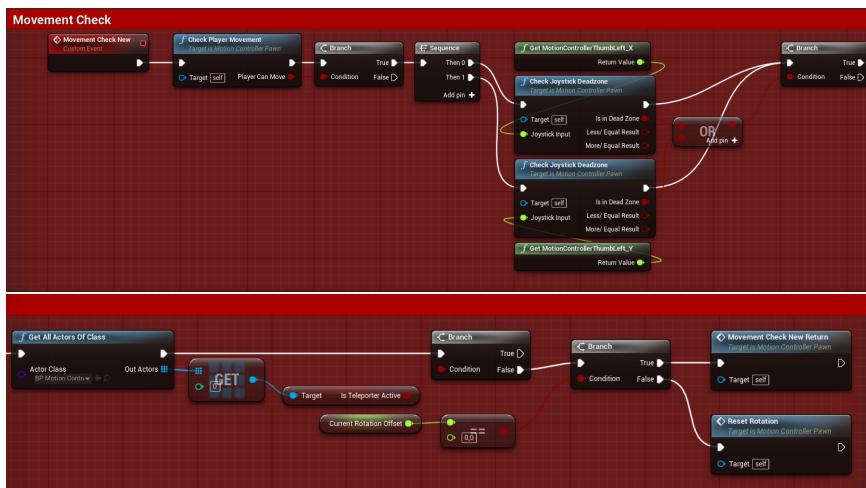
Tipka *grip* se na VR kontroleru nalazi s bočne strane desnog i lijevog kontrolera



Slika 54: Prazan hod kontrolera

Controller Thumb Left_X te Get Motion Controller Thumb Left_Y. Unos vrijednosti koje dobivamo od kontrolera vršimo istovremeno za lijevi i desni joystick. Upravljanje teleportacijom je implementirano u klasi BP Motion Controller te pomoću funkcije GetAllActorsOfClass dohvaćamo sve *actore* navedene klase dok je za ovaj slučaj ona potrebna za dohvatanje vrijednosti varijable IsTeleportActive. U našem primjeru postoji samo jedan takav *actor* pod indeksom nula (točnije rečeno, postoji samo jedna instanca kojoj je indeks sigurno nula). Ovo je način kako dobivamo pristup varijabli, komponenti, funkciji ili događaju neke druge klase bez *castanja* jer se ovdje ne mora *castati* isti tip podataka. Nakon što dohvativamo IsTeleporterActive *actora* BP_MotionController, ispitujemo njenu vrijednost (varijabla koja se postavlja ukoliko je teleporter aktiviran). Ukoliko navedena varijabla nije postavljena na vrijednost True, dakle teleporter nije aktiviran, tada imamo zadovoljen uvjet za kretanje pomoću gljivice koji ćemo tek implementirati.

Kada se ovaj čvor izvrši do kraja, potrebno je pozvati jedan *custom event* koji će pozvati događaj samog kretanja pomoću gljivice. Unaprijed se stvara jedan događaj koji će ponovno postaviti rotaciju te ga je potrebno pozvati, a detaljnije će se se objasniti u rotaciji.



Slika 55: Provjera mogućnosti kretanja pomoću kontrolera

Zašto se implementira sustav mrtve zone? Igraču se može dogoditi da slučajno dotakne kontroler što bi izazvalo trenutno rotaciju ili kretanje. Upravo zbog toga, implementirali smo zaštitni mehanizam u obliku „mrtve zone“ koji je aktivan toliko dugo dok se ne premaši navedenih 40% pomaka gljivice.

3.3 In Game Position

Pomoću funkcije `GetWorldTransform` dohvaća se transformacija kamere (komponenta `Camera`). Nakon toga uzima se lokacija te rotacija po Z osi. Uzete vrijednosti ćemo iskoristiti za kreiranje nove varijable `Transform` u funkciji `MakeTransform`. Ta se nova `transform` vrijednost koristiti u `TransformDirection` funkciji.

Navedena funkcija uzima varijablu `transform` i usmjerava ju prema vektoru koji joj proslijedimo kao drugi argument. Kako bismo dobili vektor pomoću kojeg ćemo dobiti `Transform`, moramo dohvatiti X i Y vrijednost lijeve gljivice kontrolera. Iz dohvaćenih vrijednosti stvaramo vektor putem funkcije `MakeVector` (Z vrijednost vektora će u ovom slučaju biti postavljena na nulu). Pri tome vrijedi napomenuti da će se X i Y vrijednosti zamijeniti kod kreiranja vektora zato što su koordinate gljivice kao u koordinatnom sustavu – X je lijevo-desno, a Y je gore-dolje. Za *actora* vrijedi drugačije – X je naprijed-nazad, a Y je lijevo-desno. Dobiveni vektor množimo zadatom vrijednošću koja predstavlja brzinu. Nakon izvršenja funkcije `AddWorldOffset` pozivamo događaj `ET Update Position` kojeg smo pozivali i nakon ažuriranja *roomscale* pozicije. To znači da će se kod kretanja pomoću gljivice igrač ponovno postavljati na mjesto gdje je kapsula isto kao što to radi kod *roomscale* kretanja. Opisano se može vidjeti na slici 56.



Slika 56: Kretanje unutar igre

Zaključimo: cijelo pokretanje podijeljen je na tri odvojene funkcionalnosti. Prva postavlja kapsulu na pravu poziciju u *RoomScale* (znači kada se igrač koji igra igru pomakne, pomaknut će se i kapsula u VR-u). Druga funkcionalnost je da se igrača postavi na poziciju na kojoj je kapsula, dok je zadnja postavljanje kapsule dok se vrši kretanje gljivice pri čemu se odmah nakon toga poziva postavljanje igrača (druga funkcionalnost).

3.4 ROTACIJA

Nakon što smo objasnili koncept pomicanja, tj. kretanja, pozabavit ćemo se rotacijom. Slično kao i kod kretanja, na početku se vrši provjera kojom želimo utvrditi može li se igrač kretati i rotirati. Provjeravamo postoje li

Varijable `Transform` sadrže vrijednosti lokacije, rotacije i uvećanja.

prazan hod na način koji je objašnjen u prethodnom poglavlju. Nakon toga provjeravamo varijable **Less/Equal Result** i **More/Equal Result**. Ako je vrijednost varijable **Less/Equal Result** istinita, varijabla **PlayerRotation** se postavlja na -45 , a ako je vrijednost varijabla **More/Equal Result** istinita, postavlja se na 45 . Ukoliko se kontroler nalazi u praznom hodu, dolazi do resetiranja čvora **Do Once**. Dakle na jedan *input* igrača dolazi do rotacije lijevo ili desno te se nova rotacija može dogoditi ponovno tek kad se gljivica vrati u prostor praznog hoda i onda ponovno izvan njega. Razlog uvođenja ovog mehanizma je taj što ukoliko igrač stalno drži gljivicu kontrolera, dalo zilo bi do neprestane vrtnje za 45 stupnjeva te bi to bilo neugodno iskustvo. Prikaz je na slici 87.



Slika 57: Rotacija igrača

Kako bismo umanjili mogućnost izazivanja tzv. *motion sicknessa* dodaje se funkcija **StartCameraFade**. U ovoj se funkciji prvo dohvata kamera, nakon toga se dodaje zacrnjenje ekrana (ali ne potpuno, opcija **To Alpha** postavljena na 0.5 , dakle pola) u trajanju od 0.1 sekunde kako bi se napravio gladi prijelaz prilikom rotacije kamere. Detalji postavki funkcije **StartCameraFade** prikazani su u tablici 4.

Postavke	Vrijednost
Target	GetPlayerCameraManager
From Alpha	0
To Alpha	0,5
Duration	FadeOutDuration
Color	TeleportFadeColor

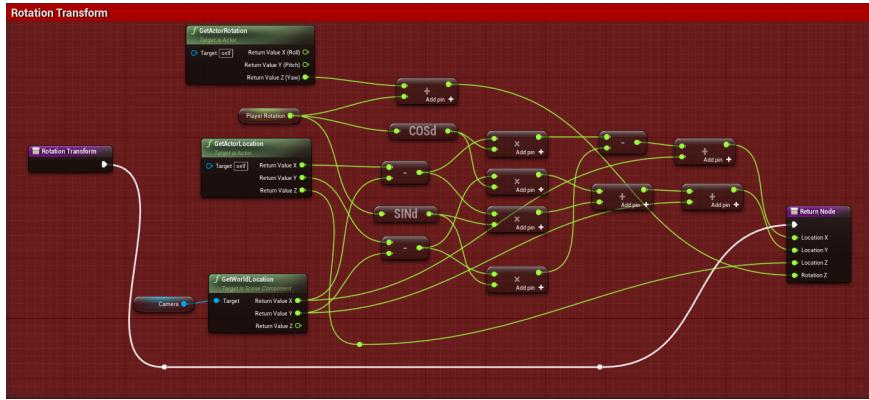
Tablica 4: Start Camera Fade

Nakon odgode koja traje jednakо dugo kao i funkcija **Start Camera Fade**, dakle 0.1 sekundu, pokreće se funkcija **Rotation Transform**. Odgodu smo postavili kako bi se funkcija **Start Camera Fade** mogla izvršiti do kraja prije nego što se počne pokretati funkcija **RotationTransform**.

Matematička logika iza ove funkcije je kompleksna pa zbog toga se nećemo previše upuštati u detalje, a njena implementacija je preuzeta. Za ovu funkciju je bitno zapamtiti da uzima igračevu lokaciju, rotaciju te lokaciju kamere. Rezultat ove funkcije je kreiranje nove lokacije na kojoj bi se igrač

Motion sickness (morska bolest, kinetoza) je događaj u kojem se osoba može osjećati loše i nelagodno prilikom brze izmjene slike ili brzih pokreta. Taj se osjećaj može dogoditi ukoliko osoba koristi VR, a nije naviknuta na vizualne podražaje koji se ne slažu s onime što ostatak tijela percipira

trebao nalaziti te nove rotacije, odnosno smjera u kojem igrač gleda. Prikaz na slici 58.



Slika 58: Funkcija rotacije

Izlazi koje daje funkcija `RotationTransform` potrebni su kao argumenti za funkciju `SetActorTransform` koja od dobivenih vrijednosti dobiva `TransformLocation` za X , Y i Z te `TransformRotation` za Z .

Nakon izvršenja funkcije `SetActorTransform` još se jednom pokreće funkcija `Start Camera Fade` koja vraća kameru iz zacrnjenog stanja u ono normalno. U tablici 5 nalaze se varijable koje se trebaju postaviti jer se razlikuju od postavki prikazanih u tablici 4.

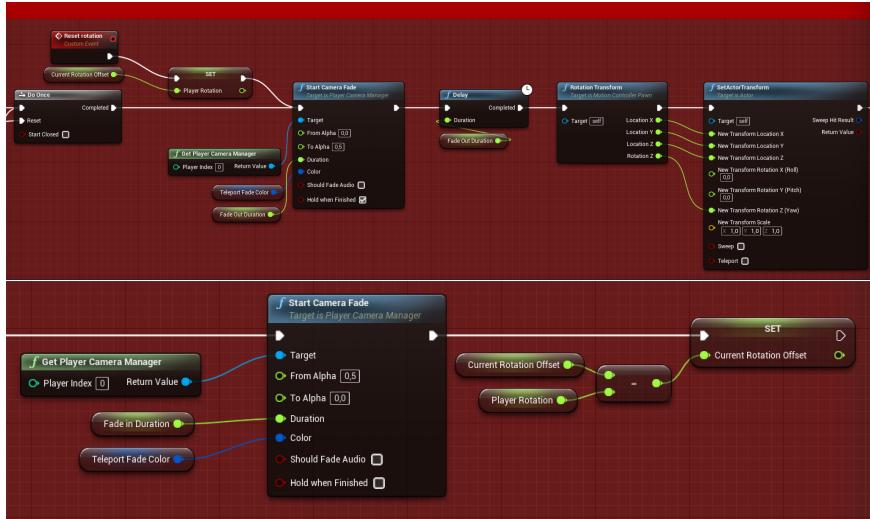
Postavke	Vrijednost
From Alpha	0,5
To Alpha	0

Tablica 5: `Start Camera Fade` nakon rotacije

Prilikom rotacije kamere zacrnimo kameru, izvršimo rotaciju te vratimo kameru u početno stanje. Ovu kompleksnu funkciju za rotacije koristimo jer ne možemo direktno utjecati na kameru jer je vezana na HMD, ali možemo rotirati `PlayerArea`, no ako koristimo običnu funkciju `SetActorRotation`, rotacijom ćemo se i pomaknuti u VR svijetu jer se `RoomScale` pozicija ne uračunava u rotaciju.

Ukratko opisano, prilikom rotacije kamere zacrnimo kameru, izvršimo rotaciju te vratimo kameru u prvo stanje. Vizualni prikaz može se vidjeti na slikama 59.

Postoji još jedan dio kôda koji nije opisan, ali je bio prikazan na slikama do sada. On se tiče događaja `ResetRotation` kojeg pozivamo u događaju `MovementCheck`. On služi za ponovno postavljanje rotacije na vrijednost prije svih rotacija. Logika se odvija ovako: kada igrač pomakne gljivicu za kretanje, odnosno želi se kretati, a da pritom nije na početnoj rota-

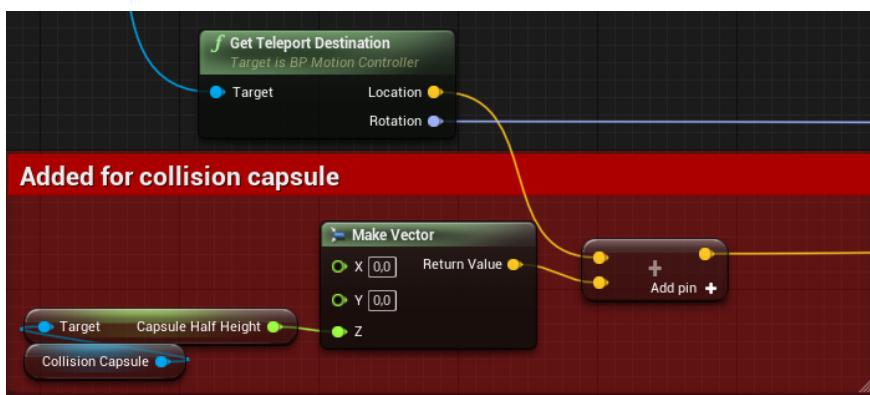


Slika 59: Prikaz završetka rotacije

ciji, prije kretanja dogodit će se ponovno postavljanje rotacije. Varijabla `PlayerRotation` postavlja se na vrijednost varijable `CurrentRotationOffset` te se s tom varijabljom odvija objašnjena funkcionalnost rotiranja.

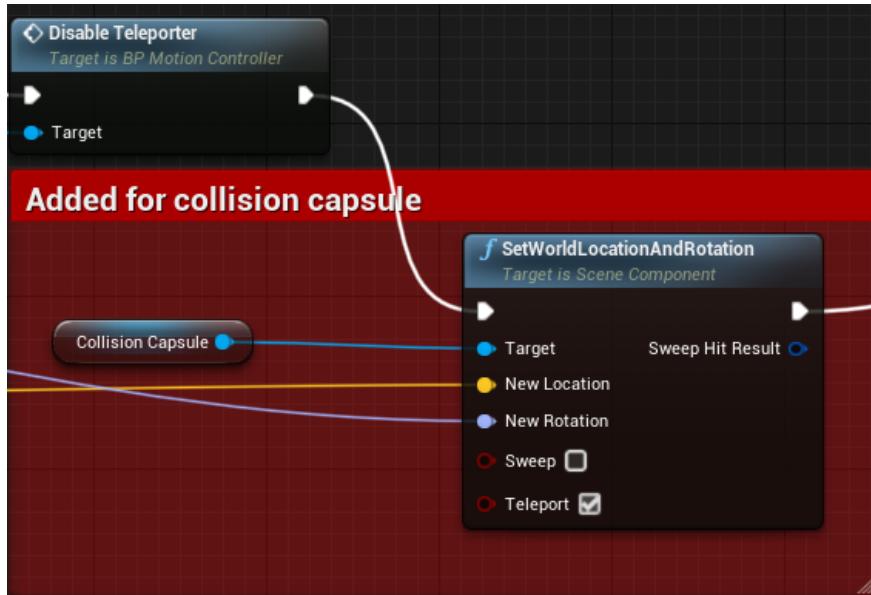
Vrijednost varijable `CurrentRotationOffset` postavlja se na kraju svake rotacije na vrijednost umanjenu za vrijednost varijable `PlayerRotation`. Tako uvijek znamo za koliko je igrač okrenut od početne rotacije.

S obzirom na to da u kretanju postavljamo igrača na poziciju kapsule, više se ne možemo teleportirati jer ćemo se isti tren vratiti na poziciju na kojoj je kapsula. Zato je potrebno modificirati funkcionalnost teleportacije na nekoliko mjesto. Vektor koji šaljemo funkciji teleport modificirati ćemo zbrajanjem s još jednim vektorm kojeg dobivamo od kapsule.



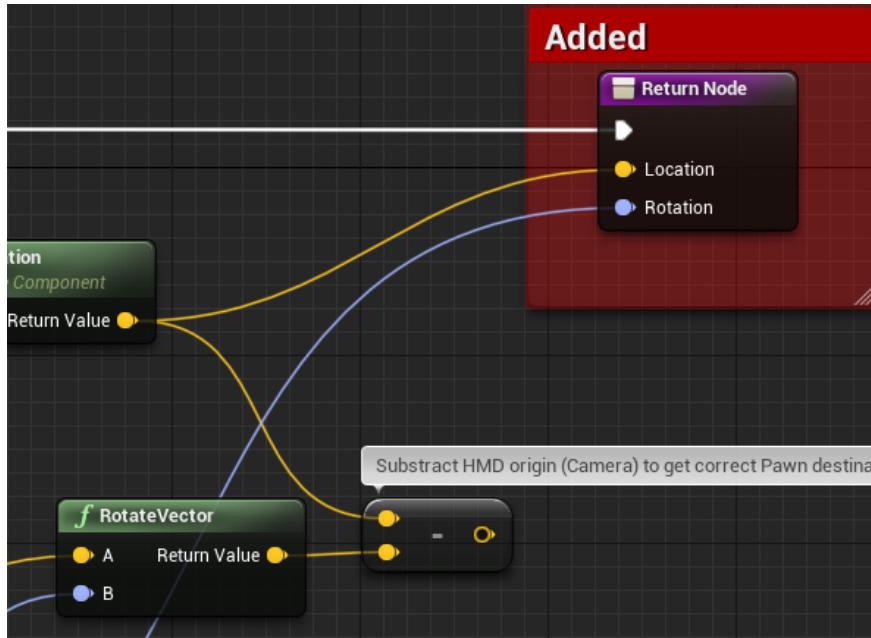
Slika 60: Promjena vektora za teleportaciju

Isto tako ne možemo koristiti funkciju `Teleport` već koristimo funkciju `SetWorldLocationAndRotation` kojoj proslijedujemo dobiveni vektor te kao metu postavljamo kapsulu.



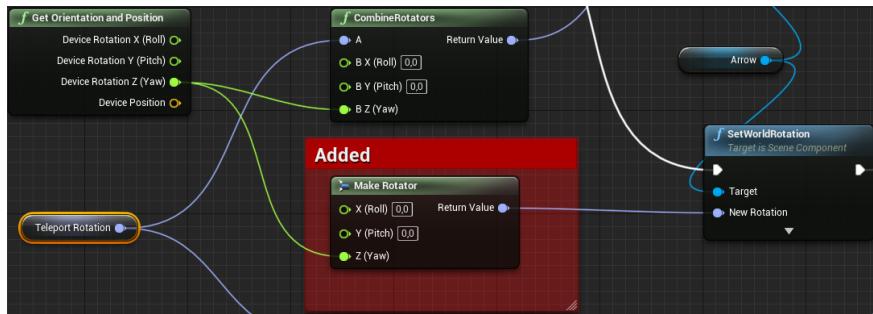
Slika 61: Promjena vektora za teleportaciju

Unutar funkcije `GetTeleportDestination` također je potrebno napraviti jednu promjenu, a to je direktno proslijediti vektor iz funkcije `GetWorldLocation` objekta `Teleport Cylinder`.



Slika 62: Promjena vektora za teleportaciju

Zadnja promjena koju je potrebno napraviti nalazi se u klasi `BP_MotionController` gdje se za rotator postavlja samo rotacija po *Z* za funkciju `SetWorldRotation` jer ne želimo da igrač može gljivicom odabrat smjer rotacije, a samim time se to i vizualno mora zako prikazati.



Slika 63: Promjena vektora za teleportaciju

3.5 ZAKLJUČAK

U ovom smo poglavlju apsolvirali osnove kretanja i snalaženja u prostoru. Kao dodatan izazov čitatelju preporučamo napraviti kretanje bez kapsule kolizije kako biste dobili odgovor na to zašto je ona potrebna.

Prostor za bilješke:

4 BILJEŽENJE POKRETA

Jedna od najvažnijih funkcionalnosti igre je korištenje sposobnosti za pobjeđivanje neprijatelja. Igrač aktivira sposobnosti raznim pokretima ruku koje VR *headset* (ili kamera) detektira. Različitim pokretima aktiviraju se različite sposobnosti. Kako bi se implementirala takva funkcionalnost potrebno je osmisliti način praćenja pokreta ruku, pohraniti zabilježene pokrete te ih i obrađivati kako bi se određenim pokretima pridružila sposobnost.

Za praćenje pokreta iznimno je važan *actor BP_TriggerArea*. Navedeni *blueprint* sadrži osam objekata klase **Sphere Collision**, dva događaja i još poneke varijable. Sfere za koliziju služe za bilježenje pokreta ruku. Nakon što igrač dotakne sferu poziva se funkcija događaja kojom se započinje proces pohrane izvedenog pokreta i uspoređivanja obavljenih pokreta s listom mogućih sposobnosti.

4.1 SFERE KOLIZIJE

Actor BP Trigger Area stvara se tako što se stisne desni klik na *Content Browser*, pod sekcijom *Create basic asset* odabere se *Blueprint class*. U dobivenom prostoru pod *Common Classes* odabiremo *Actor*. Na taj se način dodaju i svi budući *actori* u ovom radu.

Instanciranje *actora BP Trigger Area* događa se pozivom događaja *BP Non Base* koja se poziva na događaj *Event BeginPlay* koji označava početak igre za stvorenu instancu *actora*. To znači da će se sfere kolizije instancirati čim se pokrene igra.

Sfere kolizije stvaramo tako da ulazimo u stvorenog *actora*, odabiremo pogled *viewport* i prvo na komponentu *DefaultSceneRoot* dodajemo komponentu pritiskom na *Add component* i onda pretražujemo *arrow*. Ta će nam komponenta bazni element za sve sfere kolizije. Nakon što smo dodali tu komponentu, pod nju dodajemo osam sfera kolizije. One su nazvane:

- `TopLeftSphere`
- `TopRightSphere`
- `FrontLeftSphere`
- `FrontRightSphere`
- `SideLeftSphere`
- `SideRightSphere`

- BottomLeftSphere
- BottomRightSphere

Pozicije na koje su postavljene su maksimalno ispružene ruke prema gore za gornje sfere, maksimalno ispružene ruke naprijed za prednje sfere, maksimalno ispružene ruke u stranu za sfere sa strane te opuštene ruke uz tijelo za donje sfere. Postavljanje bi se tehnički trebalo napraviti kao jedna od opcija, no nama je za testiranje dovoljno imati jedan primjerak.

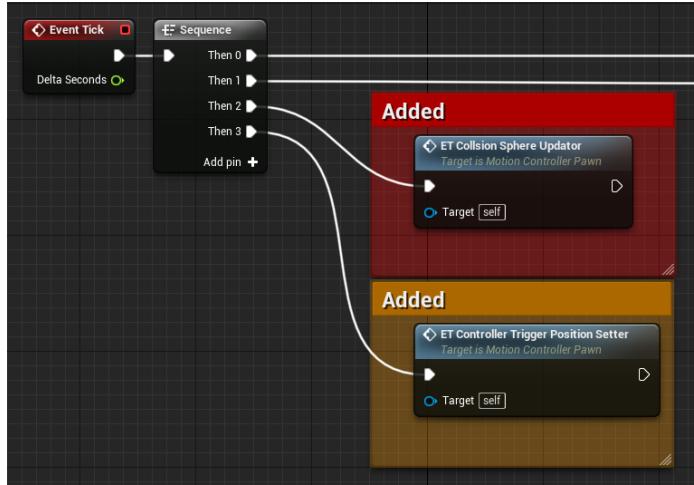
U **MotionControllerPawn** potrebno je dodati jednu komponentu na koju će se stvarati **BP Trigger Area**. Tu komponentu dodajemo pod kameru te je ona tipa **Arrow**. Ovo će služiti kao sidro za stvaranje *actora* sa sferama. Ovu smo komponentu nazvali **Trigger Area Box**. **Trigger Area Box** je vektor koji je podkomponenta kamere pa se cijelo vrijeme miče zajedno s igračem te prati usmjerenje igrača. Na taj način položaj i smjer strelice omogućuje točno postavljanje sferi u odnosu na igrača. U varijablu **Trigger Area** postavlja se referenca na *actora* **BP Trigger Area** kako bi se na taj način on mogao pozivati kada god je to potrebno odnosno kad god je potrebno ponovno pozicionirati sfere ispred igrača. Prikaz je na slici 64.



Slika 64: Stvaranje prostora za *trigger*

Nakon što smo dodali sfere za koje želimo bilježiti pokrete, potrebno je dodati i sfere koje će detektirati koliziju s već napravljenim sferama. Iz tog razloga potrebno je u **MotionControllerPawn** dodati još dvije komponente koje smo nazvali **Left Hand Motion Trigger** i **Right Hand Motion Trigger** koje postavljamo u **Default Scene Root** komponentu. Dodane sfere su trenutno statične te ih je potrebno aktivno ažurirati s obzirom na poziciju kontrolera. Dodajemo događaj **ET Controller Trigger Position Setter** u kojem ćemo na svaki *tick* ažurirati poziciju obje sfere. Događaj pozivamo na **Event Tick** što se može vidjeti na slici 65.

U spomenutom događaju koristimo dvije funkcije **Set World Location** za postavljanje pozicije svakog od kontrolera. Objekti nad kojim želimo vršiti funkciju su sfere **Left Hand Motion Trigger** i **Right Hand Motion Trigger** te za svakog od njih dobivamo lokaciju preko reference na kontroler. Uzima se referenca na kontroler iz koje se dohvata komponenta **Motion Controller** te preko funkcije **Get World Location** koju prosljeđujemo kao argument funkciji **Set World Location**. To se vidi na slici 66



Slika 65: Pozivanje događaja ET Controller Trigger Position Setter



Slika 66: Postavljanje pozicije sfera prema poziciji kontrolera

Kako bismo bili sigurni da će jedino sfere na rukama okidati koliziju na sferama *trigger areae* potrebno je dodati novi kolizijski kanal. To činimo tako da odemo u *Project Settings*, pod *Collision* pod postavkom *Object Channels* pritišćemo *New Object Channel*. Tada nam iskače mali prozor koji će se popuniti kao na tablici 6.

Postavka	Vrijednost
Name	Trigger Area
Default Response	Ignore

Tablica 6: Izmjena postavki kod stvaranja novog kolizijskog kanala

Sada u opcijama *Left Hand Motion Trigger* i *Right Hand Motion Trigger* u *actoru MotionControllerPawn* potrebno je pod stavkom *Collision* promijeniti stavke prema tablici 7

Praćenje pokreta ruku pokreće se nakon što igrač pritisne tipku na jednom od kontrolera i prestaje kada korisnik pusti tipku. To znači da nije potrebno znati gdje se sfere nalaze cijelo vrijeme već samo kada igrač pritisne tipku. U tu svrhu važan je događaj *InputAction BeginTrackingLeft* i *InputAction BeginTrackingRight* koji su dodani u poglavljju 3. Prilikom okidanja jednog od ta dva događaja poziva se referenca na BP *Trigger Area* (varijabla *Trigger Area*) pomoću koje se taj *actor* postavlja na loka-

Postavka	Vrijednost
Collision Presets	Custom
TRACE RESPONSES	
Visibility	Ignore
Camera	Overlap
OBJECT RESPONSES	
Trigger Area	Overlap

Tablica 7: Izmjena kolizijskih postavki za *Left Hand Motion Trigger* i *Right Hand Motion Trigger*

ciju komponente **Trigger Area Box** prema rotaciji kamere. Prilikom postavljanja rotacije važno je postaviti samo *Z* os (*Yaw*) zato što je u tom slučaju jedino važno u koju je stranu igrač trenutno okrenut. Drugim riječima, rotacije po drugim osima su nevažne. Time su i sfere (koje su dio **BP Trigger Area**) postavljene na mjesto na kojem trebaju biti za praćenje pokreta.

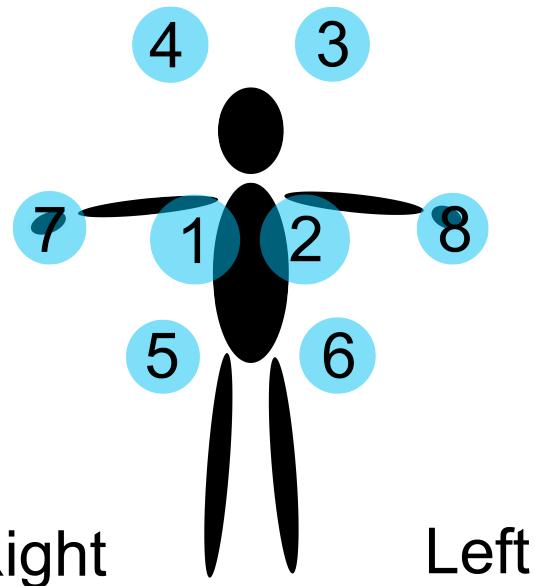
Kako bismo znali koju smo od stvorenih sfera taknuli, potrebno je svakoj sferi pridodijeliti *tag*, odnosno oznaku. Svakoj ćemo sferi dodati dvije oznake od kojih je jedan *trigger* što označava da radimo s **Trigger Area**, a drugi je za indeksiranje. Otvaramo *actora* **BP Trigger Area** i za svaku komponentu zasebno dodajemo oznake. Oznake koje je potrebno unijeti su prikazani u tablici 8 i na slici 67. Za svaku sferu je prva oznaka *trigger* pa je nećemo svaki puta nabrajati u tablici.

Oznaka se dodaje na način da kad je izabrana željena komponenta u *Components* kartici, u njenom *Details panelu* pretražujemo ključnu riječ *tag* te se u sekcijsi *tags* u dijelu *Component tags* pritišće + dvaput. Oznaka s indeksom 0 je *trigger*, a oznaka s indeksom 1 navedena je u tablici.

Sfera	Indeks
TopLeftSphere	3
TopRightSphere	4
FrontLeftSphere	2
FrontRightSphere	1
SideLeftSphere	8
SideRightSphere	7
BottomLeftSphere	6
BottomRightSphere	5

Tablica 8: Popis indeksa prema sferi

Nakon što su sfere postavljene tamo gdje trebaju biti, može se krenuti s praćenjem. Događaj **On Component Begin Overlap** (postoji i za **Right**



Slika 67: Vizualni prikaz sfera po indeksu

Hand Motion Trigger i za Left Hand Motion Trigger) okida se svaki put kada igrač dotakne neku sferu. On se dodaje u Event Graph Motion Controller Pawn na način da se označi željena komponenta u *Components* te u *Details panelu* pod sekcijom *Events* pritisne + na opciji *On Component Begin Overlap*. Smatra se da je igrač dotaknuo sferu BP Trigger Area kada se Right Hand Motion Trigger ili Left Hand Motion Trigger dotaknu sferom.

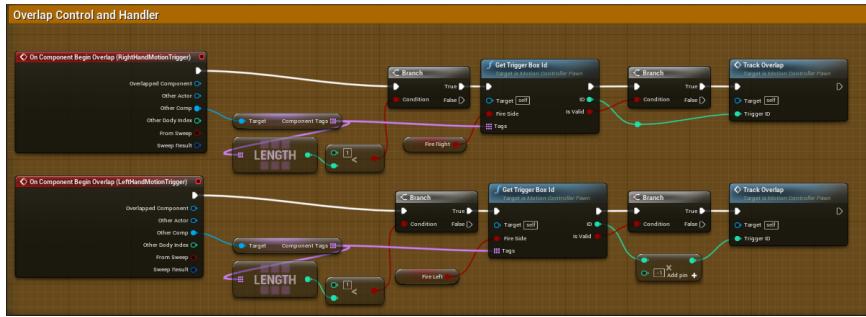
4.2 OKIDANJE DOGADAJA DODIRA SFERE

Nakon što se okinuo događaj *On Component Begin Overlap* kao argument se uzima komponenta *Other Comp* kojom se jedna od ručnih sfera preklopila, u ovom slučaju ta će komponenta biti jedna od sfera *Trigger Area*. Prva stvar koju obavlja pokrenuta funkcija jest provjeravanje ima li komponenta čija je referenca u varijabli *Other Comp* više od jedne oznake.

Potom se potvrđuje da komponenta *Other Comp* ima više od jedne oznake pa događaj može krenuti dalje jer to znači da se može pozvati funkcija *Get Trigger Box ID*. Nakon toga poziva se funkcija *Get Trigger Box ID* koja traži jedan argument. Za argument se uzimaju oznake komponenata *Other Comp*. Na slici 68 može se vidjeti taj *blueprint*.

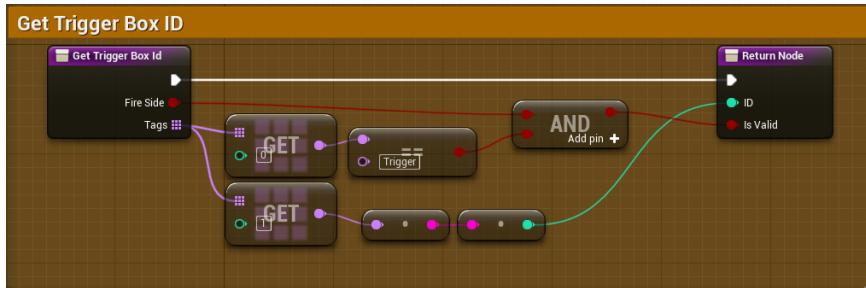
U funkciji *Get Trigger Box ID* se provjerava je li prva oznaka jednaka tekstu „Trigger“ i ako je, vraća se *True*, u suprotnom se vraća *False*. Također, kako bi se vratila *True* vrijednost, jedna od varijabli *FireRight* ili *FireLeft* mora biti postavljena na *True*. Jedna od tih varijabli je istinita ako je pritisnuta tipka na kontroleru kojom se aktivira praćenje sfera. Drugo što vraća

Tag je vrsta oznake koja se nekom *actoru* ili komponenti *actora* može dodijeliti. Svaka komponenta može imati nula ili više oznaka.



Slika 68: Događaj koji se okida na koliziju sfera

funkcija **Get Trigger Box ID** je integer u kojem se nalazi ID sfere koja je taknuta. Ako je funkcija **Get Trigger Box ID** vratila istinu, događaj **On Component Begin Overlap** nastavlja svoje izvođenje pozivom funkcije **Track Overlap**, a inače ne radi ništa. Detaljan vizualni prikaz nalazi se na slici 69.

Slika 69: Funkcija **Get Trigger Box ID**

U slučaju da je događaj **On Component Begin Overlap** pozvan s **Left Hand Motion Trigger** onda se integer varijabla koju je vratila funkcija **Get Trigger Box ID** množi s -1 kako bi se znalo kojom je rukom dotaknuta sfera. Taj je podatak bitan zato što se svaka sposobnost poziva kombinacijom dodira sfera i lijevom i desnom rukom zato da igrač koristi obje ruke, a množenjem s -1 se zna da je igrač dotaknuo sferu lijevom rukom.

4.3 UNOŠENJE PODATAKA O DODIRU SFERE U POLJE

Funkcija **Track Overlap** poziva funkciju koja pohranjuje ID dotaknute sfere u pripadnu strukturu i provjeravaju podudaranje dosad dotaknutih sfera sposobnostima. **Track Overlap** prima samo jedan argument: **Trigger ID** tipa integer pomoću kojeg se prosljeđuje ID dotaknute sfere. Prvo što se provjerava je je li barem jedna od dvije Boolean varijable **Fire Right** ili **Fire Left** istinita. To se radi zato što istinitost jedne od tih varijabli određuje je li moguće izvesti sposobnosti. Ako su obje varijable lažne gumb nije ni na jednom kontroleru stisnut i igrač ne može izvoditi sposobnosti.

Ukoliko je uvjet u grananju istinit, poziva se funkcija `Add action`. Detalji su prikazani na slici 70.



Slika 70: Funkcija `Get Trigger Box ID`

Funkcija `Add action` prima dva argumenta. Prvi je argument `Trigger ID` tipa integer u koji je pohranjen ID dotaknute sfere. Drugi argument je `Trigger Time` tipa Float te se u njega prosljeđuje trenutno vrijeme koje daje funkcija `Get Game Time in Seconds`.

U funkciji `Add action` prvi se put pojavljuje javno polje `Trigger Chain` tipa `S_Chain`. `S_Chain` je vlastita struktura podataka koja se sastoji od `Integer` varijable `triggerBoxID` i `Float` varijable `TriggerTime`. `Trigger Chain` služi za pohranjivanje ID-ja sfere koja je dotaknuta i vremena kada je ona dotaknuta.

Strukture se dodaju desnim klikom na prazno mjestu u *Content browseru* pod sekcijom *Create advanced asset* odaberemo *blueprints* te pritisnemo na *Structure*. Strukturu `S_Chain` kreiramo kao u tablici 9.

Ime	Tip
Trigger Box ID	integer
Trigger Time	float

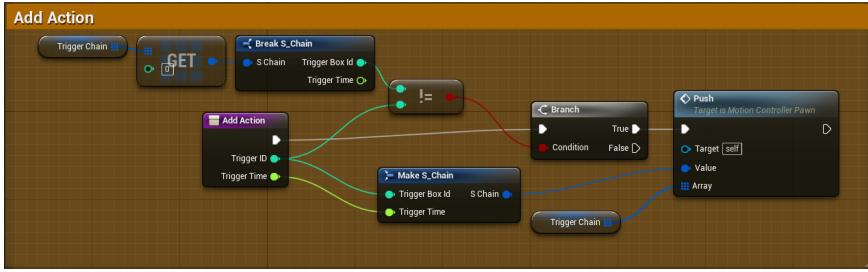
Tablica 9: Prikaz početnih varijabli u strukturi `S_Chain`

Potrebno je dodati novu varijablu pod nazivom `Trigger Chain` koji će biti tipa `S_Chain`.

Prilikom ulaska u funkciju `Add action` prvo se provjerava je li vrijednost argumenta `Trigger ID` jednaka vrijednosti `Trigger Box ID` prvog elementa polja `Trigger Chain` koja je dohvaćena pomoću funkcije `Break Chain`. Provjera je nužna radi toga što se ne smije dogoditi da je ID iste sfere zapisan u `Trigger Chainu` jedan za drugim. Zbog te provjere nije moguće istu sferu dotaknuti dva puta za redom. Ako se ID-jevi podudaraju, funkcija ne radi ništa. U slučaju da su ID-jevi različiti, stvara se element tipa `S_Chain` u kojeg se pohranjuje `Trigger ID` i `Trigger Time`. Nakon toga se poziva funkcija `Push`.

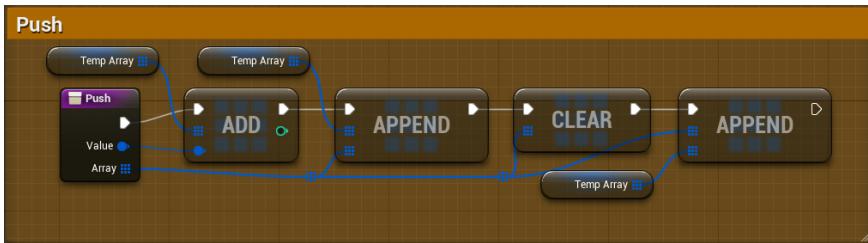
Funkcija `Push` kao argument prima polje `S_Chain` u koje se zapisuje `Trigger Box ID` i `Trigger Time` te varijablu `Trigger Chain`.

`Push` je funkcija koja služi za ubacivanje varijable tipa `S_Chain` na prvo



Slika 71: Funkcija Add Action

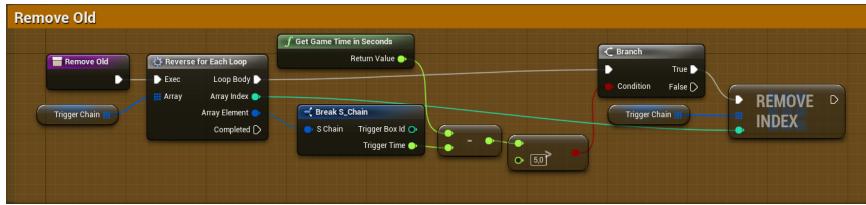
mjesto polja **Chain**. Varijabla se ubacuje na prvo mjesto zato što će se kasnije, prilikom uspoređivanja dotaknutih sfera dodirima potrebnim za pozivanje sposobnosti prvo gledati najnoviji dodiri a nakon toga stariji dodiri. Funkcija prvo stvara privremeno polje tipa **S_Chain** (naziv polja je **Temp Array**) kojem dodaje varijablu **Chain** koja je proslijedena kao argument funkcije. Nakon toga se na privremeno polje doda polje **Trigger Chain**. Na taj način **Temp Array** poprima oblik kakav **Trigger Chain** u konačnici treba poprimiti. Nakon toga se brišu svi elementi iz polja **Trigger Chain** pa se na njega dodaju elementi iz polja **Temp Array**. Prikaz na slici 72.



Slika 72: Funkcija Push

Izvršenjem funkcije **Push** završava se i funkcija **Add Action** te se vraća u funkciju **Track Overlap**. U tom trenutku u funkciji **Track Overlap** se poziva funkcija **Remove old**. Funkcija **Remove old** služi za micanje određenih elemenata iz polja **Trigger Chain**. Konkretno, miču se oni elementi čiji **TriggerTime** ima prenisku vrijednost. Funkcija **Remove Old** koristi **for each** petlju koja prolazi kroz svaki element **Trigger Chaina**. **For each** petlja kreće od zadnjeg elementa polja prema prvom. Od trenutnog vremena igre dobavljenog pomoću funkcije **Get Game Time in Seconds** oduzima se **Trigger Time** koji se izvlači iz svakog od elemenata **Trigger Chaina**. Provjerava se je li dobivena vrijednost oduzimanja veća od 5. Ako vrijednost nije manja od 5, sfera je dotaknuta umutar 5 sekundi što znači da smije ostati u **Trigger Chainu** te se ne radi ništa. U suprotnom je sfera dotaknuta u više od 5 sekundi pa se ona briše tako što se u funkciju **REMOVE INDEX** prosljeđuje **Trigger Chain** i indeks elementa polja koji se trenutno gleda. Sada se može vidjeti da je razlog korištenja obrnute **for each** petlje taj što su na kraju polja elementi koji su stariji. Zbog toga će se prvo maknuti svi

elementi koji su stariji od 5 sekundi, a kada se svi oni maknu, **for each** petlja će samo proći kroz ostale elemente.



Slika 73: Funkcija Remove Old

4.4 PRIPASIVANJE SPOSOBNOSTI

Uklanjanjem svih elemenata starijih od 5 sekundi vraća se na funkciju **Track Overlap** koja poziva funkciju **Match Ability** ako igrač nema sposobnost u rukama. To se provjerava tako da se gleda je li varijabla **Ability ID** jednaka -1 i ako jest, ulazi se u funkciju **Match Ability**. **Ability ID** je varijabla tipa integer u koju se spremaju ID sposobnosti koju igrač može ispaliti, a ako igrač ne može ispaliti niti jednu sposobnost njegova vrijednost je -1.

Match Ability je funkcija koja pregledava zabilježene doticaje sfera u polju **Trigger Chain** i vraća Ability ID sposobnosti s kojom se pokreti podudaraju. U radu s tom funkcijom postoje dvije vrlo važne strukture koje je potrebno objasniti. Prva struktura je boolean polje **Match**, a druga je polje **List of Abilities**. **Match** je polje Boolova koje će imati onoliko elemenata koliko postoji sposobnosti. Ukoliko je određeni element polja **Match** istinit, onda igrač može ispaliti sposobnost s ID-jem jednakim indeksu istinitog elementa. Polje **List of Abilities** malo je komplikiranije. **List of abilities** je dvodimenzionalno polje tipa **Abilities**. **Abilities** je struktura podataka koja se sastoji od varijable **Hand** tipa **Bool** i polja **Ability** tipa **Integer**. Prikaz vrijednosti u strukturi **Abilities** prikazana je tablično u tablici 11

Ime	Tip
Hand	bool
Ability	integer array

Tablica 10: Prikaz varijabli u strukturi Abilities

Kao što je rečeno, svaki indeks u polju **Abilities** odgovara određenom projektalu. Također, svaki element polja **Abilities** je polje u kojem su zapisani integeri u kojima piše kojim redom je potrebno dodirnuti sfere kako mogao pojaviti projektil. U nastavku je navedeno kako je implementirano (indeks → element):

Postavka	Vrijednost	Postavka	Vrijednost
0		9	
Hand	False	Hand	False
Ability	[-2, 8, 2, 1, 7]	Ability	[-2, -6, -3]
1		10	
Hand	True	Hand	True
Ability	[1, -7, -1, -2, -8]	Ability	[1, -3, 5, -8, 7]
2		11	
Hand	True	Hand	True
Ability	[-7, 7, 1]	Ability	[1, 5, -3, -8, 7]
3		12	
Hand	False	Hand	True
Ability	[8, -8, -2]	Ability	[1, -3, 5, 7, -8]
4		13	
Hand	False	Hand	True
Ability	[-2, 1, -6, 5]	Ability	[1, 5, -3, 7, -8]
5		14	
Hand	False	Hand	False
Ability	[1, -2, -6, 5]	Ability	[-2, -6, 4, -8, 7]
6		15	
Hand	False	Hand	False
Ability	[-2, 1, 5, -6]	Ability	[-2, 4, -6, -8, 7]
7		16	
Hand	False	Hand	False
Ability	[1, -2, 5, -6]	Ability	[-2, -6, 4, 7, -8]
8		17	
Hand	True	Hand	False
Ability	[1, 5, 4]	Ability	[-2, 4, -6, 7, 0]

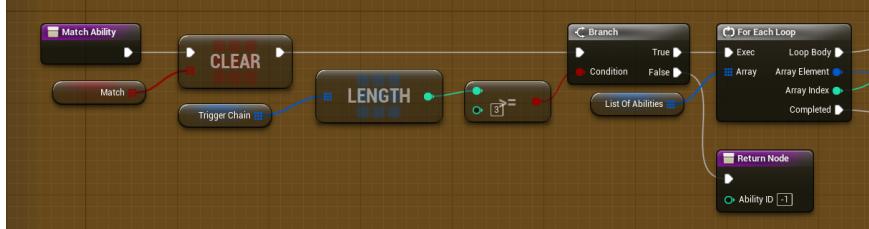
Tablica 11: Prikaz varijabli u strukturi Abilities

Varijabla `Hand` bit će istinita ukoliko je kontroler s kojim se ispaljuje sposobnost desna ruka, a laž ako je u pitanju bio lijevi kontroler. Polje `Ability` sadrži integer elemente pomoću kojih se određuje kojim se redom moraju dotaknuti sfere kako bi se određena sposobnost mogla izvesti.

Prva stvar koja se napravi nakon poziva funkcije `Match Ability` je brisanje svega iz polja `Match` i provjera duljine polja `Trigger Chain`. Ukoliko `Trigger Chain` ima manje od 3 elementa vraća se -1 i funkcija se završava. Razlog tomu je činjenica da je za izvođenje sposobnosti potrebno dotaknuti barem tri sfere što znači da u `Trigger Chainu` mora biti najmanje toliko sferi kako bi se sposobnost mogla izvesti. Ukoliko `Trigger Chain` ima bar 3 elementa funkcija se nastavlja.

Iduća stvar koju radi funkcija `Match Ability` jest korištenje `for each` pet-

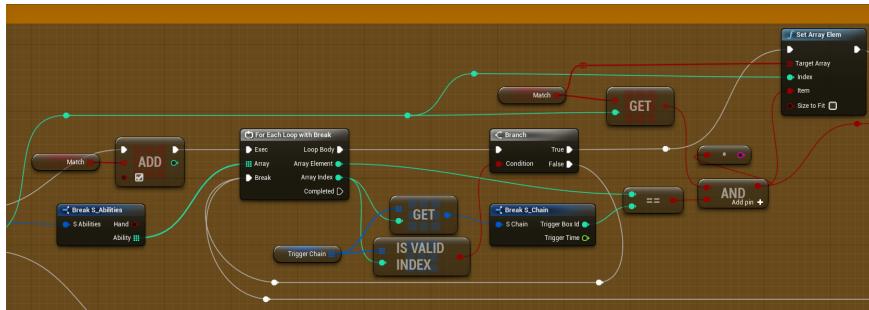
Ije kroz koju će se iterirati svi elementi polja **List of Abilities**.



Slika 74: Početak funkcije **Match Ability**

Nakon ulaska u petlju dodaje se jedan element u polje **Match** koji će imati isti indeks kao i trenutno iteriran element iz polja **List of Abilities** te će njegova vrijednost početno biti istinita. To se radi na taj način kako bi se moglo točno odrediti koji element polja **Ability** odgovara kojoj sposobnosti. Nakon toga stvara se nova **for each** petlja u kojoj se iterira polje integera **Ability**. Polje **Ability** se nalazi u trenutno iteriranom elementu polja **List of Abilities**. Trenutno iterirani element polja **Ability** uspoređuje se s varijablom **Trigger Box ID** dobavljenom iz elementa polja **Trigger Chain** koji ima isti indeks kao i iterirani element polja **Ability**. Ukoliko su uspoređivane vrijednosti jednakne ne radi se ništa, inače se element polja **Match** postavlja na laž i izlazi se iz unutarnje **for each** petlje.

Osim usporedbi navedenih u prethodnom odlomku, postoji još jedan način na koji element polja **match** može postati lažan. Naime, različite sposobnosti zahtjevaju različit broj pogodenih sfera. U slučaju da su taknute 4 sfere, a sposobnost zahtjeva 5 sferi provjerava se postoji li u **Trigger Chain** indeks jednak indeksu trenutno iteriranog elementa polja **Ability**. U slučaju da ne postoji, element polja **Match** se postavlja na **False** i izlazi se iz te petlje bez da se uopće izvodi usporedba navedena u prošlom odlomku.

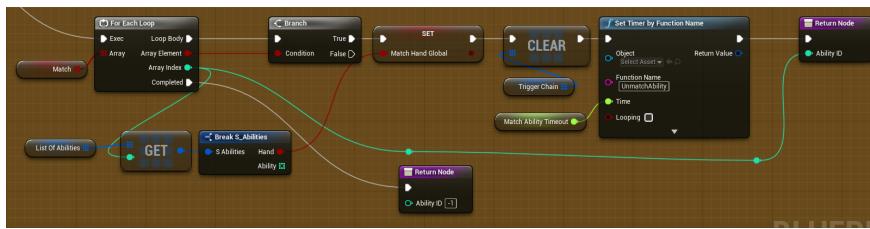


Slika 75: Uspoređivanje zapisanih vrijednosti s onima koje su okinute

Nakon što je **for each** petlja iterirala kroz svaki od elemenata polja **List of Abilities** polje **Match** je popunjeno s elementima na način opisan u prethodnom odlomku. Ukoliko su pokreti ruku bili točni, vrijednosti varijabli **Trigger Box ID** u polju **Trigger Chain** odgovarat će vrijednostima u jednom od polja **Ability**. Potrebno je pronaći sposobnost za koju vrijedi

pogodeno polje Ability i vratiti njezin ID.

Nakon što se završi iteriranje kroz `for each` petlju za polje `List of Abilities` ulazi se u novu `for each` petlju kroz koju se iterira `Match`. Za svaki element polja `Match` se gleda je li istinit ili lažan. Ako je element lažan ne radi se ništa. Ako je iterirani element istinit uzima se vrijednost varijable `Hand` iz elementa polja `List of Abilities` koji ima jedan indeks kao iterirani element. Javna varijabla `Match Hand Global` postavlja se na vrijednost dobavljene varijable `Hand`. Nakon toga se brišu svi elementi iz polja `Trigger Chain`, postavlja se timer od 5 sekundi na pozivanje funkcije `UnmatchAbility` i vraća se indeks trenutno iteriranog elementa, odnosno elementa koji je istinit. Ako se u polju `Match` ne pronađe niti jedan istinit element igrač nije imao točne pokrete ruke što znači da niti jedna sposobnost nije pronađena. U tom slučaju se vraća vrijednost -1 nakon što je završena `for each` petlja.



Slika 76: Pronalaženje sposobnosti

U prethodnom odlomku spomenuto je postavljanje varijable `Match Hand Global`. Ta varijabla pokazuje iz koje ruke se sposobnost treba ispaliti. Pomoću te ruke će se gađati sposobnost i u toj ruci će se sposobnost nalaziti.

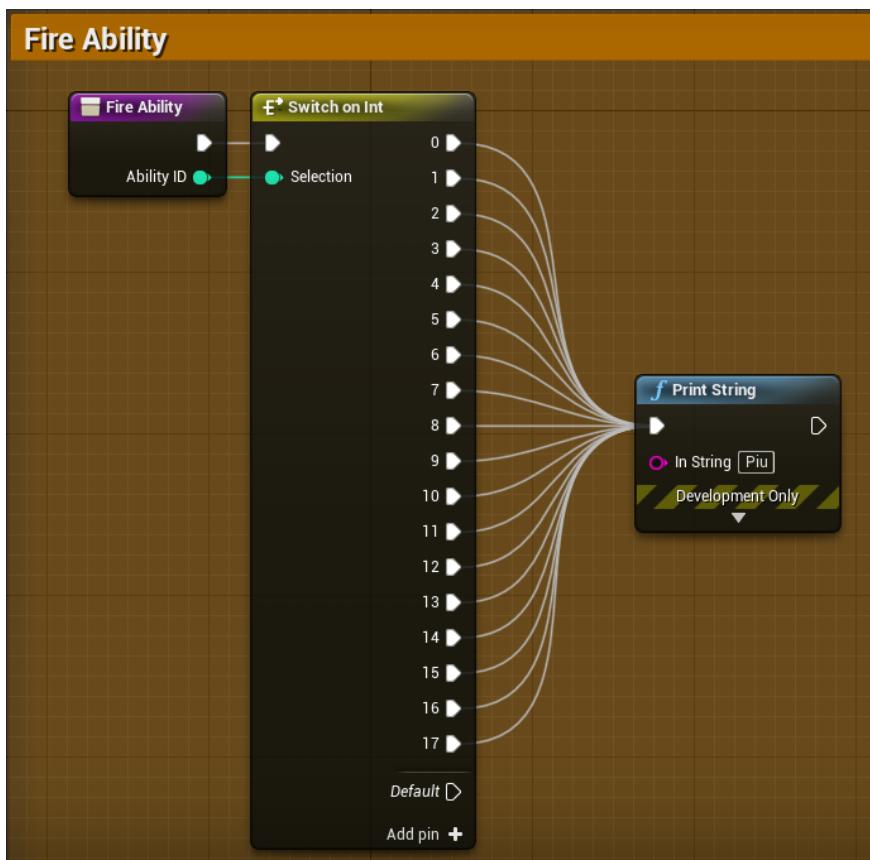
Pri izlasku iz funkcije `Match Ability` postavlja se varijabla `Ability ID` na vrijednost koju je vratila funkcija `Match Ability`. Ranije je spomenuto da postavljanje timera od 5 sekundi za poziv funkcije `UnmatchAbility`. Sve što ta funkcija (vezano za *gameplay*) radi jest ona postavlja `Ability ID` na -1 što znači da igrač više ne može ispaliti sposobnost koja je postavljena u `Match Ability`. To se radi zato da igrač ne bi mogao držati sposobnost u ruci neograničeno vremena već mu se daje period od 5 sekundi za bacanje sposobnosti prije nego se pozove `UnmatchAbility`.



Slika 77: Prikaz funkcije `UnmatchAbility`

Postavljanje `Ability ID`-ja na vrijednost vraćenu iz funkcije `Match Ability` zadnja je stavka za koju je funkcija `Track Overlap` odgovorna. Izlaskom

iz te funkcije igrač u ruci ima sposobnost ako je uspješno izveo pokret. Idući korak je iskorištavanje sposobnosti koje je igrač dobio. U svrhu toga bitno je ponovno spomenuti događaje `InputAction BeginTrackingLeft` i `InputAction BeginTrackingRight`. U jednom od prethodnih odlomaka opisano je što je točno ta funkcija radila kada se tipka pritisnula, a za bacanje sposobnosti bitno je znati što se događa kada se tipka otpusti. Kada se tipka otpusti boolean varijabla `Fire Right` ili `Fire Left` (ovisno koja je tipka dotad bila pritisnuta) se postavlja na `False`. Nakon toga se brišu svi elementi iz `Trigger Chaina` zato što se prestaju pratiti pokreti te se poziva funkcija `Fire Ability` koja za argument prima jednu varijablu tipa integer.

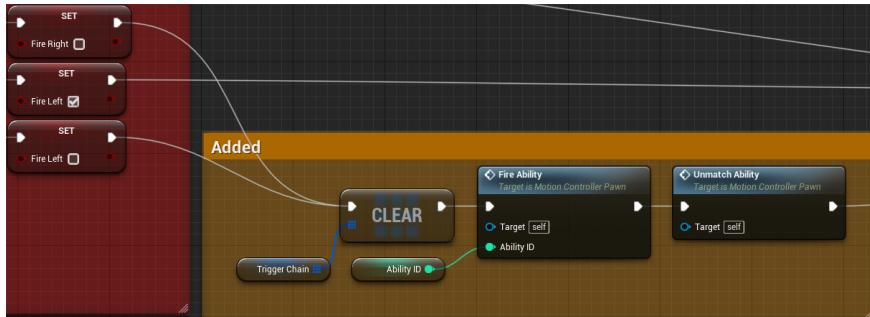


Slika 78: Prikaz funkcije `Fire Ability`

U funkciju `Fire Ability` uvjek se proslijeđuje varijabla `Ability ID` u kojem je spremljen ID sposobnosti koju igrač može ispaliti ili -1 ako igrač ne može ispaliti sposobnost. U slučaju da je proslijeđen broj -1, neće se dogoditi ništa. U slučaju da je proslijeđen neki drugi broj pozvat će se funkcija koja ispaljuje pripadnu sposobnost. Za početak sve poziva istu funkciju `Print String` koja ispisuje "Piu".

Čišćenje `Trigger Chaina` kao i pozivanje funkcija `Fire Ability` i `Unmatch ability` odvija se u `EventGraphu` poslije postavljanja varijabli `FireLeft` i

`FireRight` na `False` što se događa prilikom puštanja tipke *grip*. Opisano se može vidjeti na sljedećoj slici 79.



Slika 79: Prikaz pozivanja funkcija `Fire Ability` i `Unmatch Ability`

Ostalo je još jedino postaviti sfere na prostor oko igrača na pritisak tipke *grip*. Pošto želimo da se postavljanje dogodi samo jednom nakon pritiska tipke *grip* i da se ponovo postavljanje može dogoditi jedino nakon puštanja te tipke dodaje se `Do Once` u koji se ulazi prilikom pritiska tipke, a resetira se nakon puštanja tipke.

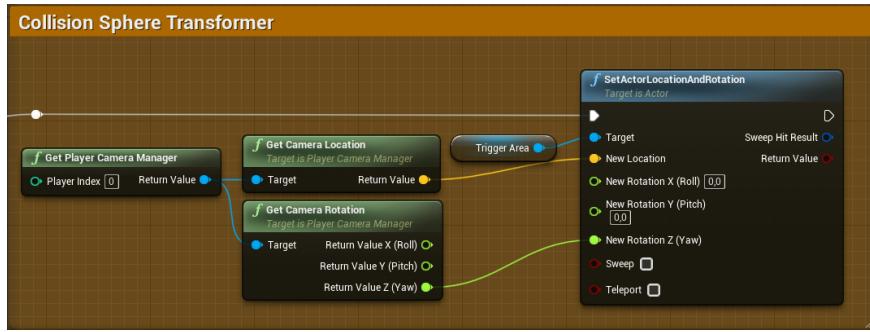


Slika 80: Prikaz `Do Once`

Nakon `Do once` slijedi postavljanje `Trigger Area` na točnu lokaciju. To se izvodi funkcijom `SetActorLocationAndRotation` koja se vrši nad objektom `Trigger Area`. Lokacija se dobiva preko funkcije `GetCameraLocation` kojoj moramo kao argument dati objekt kamere igrača. Spomenuti objekt dobivamo funkcijom `GetPlayerCameraManager`. Rotacija koja je isto potrebna funkciji `SetActorLocationAndRotation` dobivamo pomoću funkcije `GetCameraRotation` kojoj je potrebno kao argument dati isti objekt koji je trebao i funkciji `GetCameraLocation`. Rezultat funkcije `GetCameraRotation` jer cijeli rotator no nama je potrebna rotacija samo po Z jer je za usmjeravanje sfera jedino važno da su okrenute u smjeru u kojem gledamo. Nije bitno koliko visoko ili nisko usmjeravamo pogled te jesmo li nagnuti u koju stranu. Opisano se može vidjeti na slici 81

4.5 ZAKLJUČAK

U ovom poglavlju je pokazana najvažnija funkcionalnost projekta – kolizije sa sferama pomoću kojih je moguće prave magije bacati. Ovo je važna



Slika 81: Postavljanje actora Trigger Area

funkcionalnost jer se to može predstaviti kao *selling point* za aplikaciju, odnosno to je ona jedna funkcionalnost koja čini aplikaciju zanimljivom, inovativnom te može privući pozornost igrača.

Kao dodatni izazov motivira se programera igre implementira još neke sfere za koliziju te još neke magije koje igrac moze bacati.

Također bi bilo dobro razmisliti koliki je broj sfera idealan i zašto nije dobro imati veći ili manji broj sfera od toga. Kao pomoć u razmišljanju je pitati se kako velik broj sfera utječe na preciznost, a kako malen broj sfera utječe na broj magija koje igrač može bacati.

Napredan izazov za ambiciozne čitatelje bio bi i dinamično pozicioniranje sfera pri pokretanju igre, odnosno da sfere budu locirane tako da budu na dohvat ruke igrača.

Prostor za bilješke:

5 STVARANJE PROJEKTILA

Prošla lekcija završena je pucanjem sposobnosti gdje se samo pomoću funkcije print ispisivalo Piu na ekran. U ovoj lekciji ćemo dodati različite projektile koji će biti moći protiv različitih neprijatelja te kreirati jednostavni materijal pomoću kojeg ćemo moći razlikovati projektile.

5.1 KREIRANJE KLASE PROJEKTILA I NJEGOVE INSTANCE

Prvo je potrebno kreirati novog *actora* pod nazivom `BP_Projectile`. On će biti osnovni projektil za sve ostale projektile. U *viewportu* kreiranog aktora možemo promatrati komponentu koju ćemo dodati. Komponenta koja se dodaje je `Sphere` te se ona postavlja kao Root komponenta. Kako bi se projektil mogao kretati potrebno mu je dodati komponentu `ProjectileMovement`. Za tu komponentu ne moramo paziti koja će joj komponenta biti roditelj jer ova komponenta uvijek stoji za sebe i ne može imati scenske komponente kao roditelje. U *Details panelu* komponenete `ProjectileMovement` potrebno je postaviti sljedeće postavke.

Postavka	Vrijednost
Intial Speed	1500
Max Speed	1500
Rotation Follows Velocity	True
Projectile Gravity Scale	0,2

Tablica 12: Prikaz promjena u Details panelu `ProjectileMovement`

Sada možemo kreirati četvero djece *actora* `BP_Projectile`. To radimo desnim klikom na aktora i opciju `Create Child Blueprint Class`. Aktore nazivamo

- `BP_AirProjectile`
- `BP_FireProjectile`
- `BP_IceProjectile`
- `BP_ThunderProjectile`

Sada imamo četiri projektila koje možemo stvoriti kada je to potrebno. Stvaranje se odvija u `MotionControllerPawn` pa ćemo napraviti *custom* događaj za ispaljivanje svakog od projektila. Potrebno je kreirati događaje sljedećih naziva.

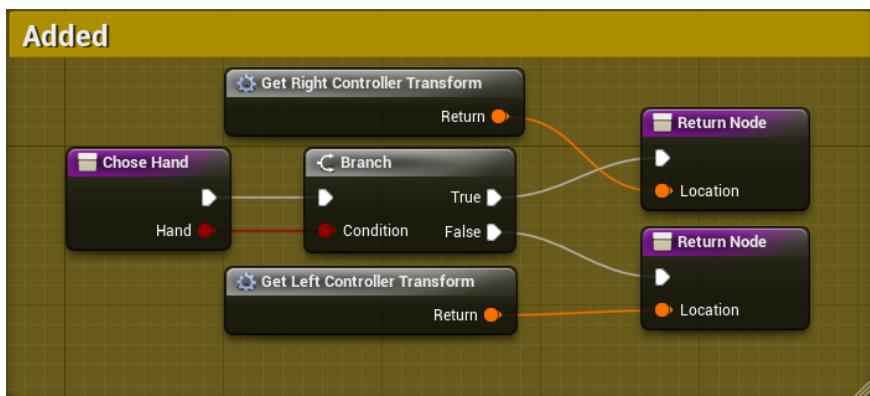
- `MotionFire Air`

- MotionFire Fire
- MotionFire Ice
- MotionFire Thunder

Opisati će se samo jedan od nabrojenih događaja jer ostali slijede istu logiku te im je jedino klasa koju stvaraju različita.

Prva stvar koju radi događaj **Motion Fire Air** je pozivanje funkcije **Choose Hand** koja prima jedan argument tipa *bool* prema kojem će se odlučiti u kojoj ruci se treba stvoriti projektil za sposobnost.

U funkciju **Choose Hand** se proslijeđuje varijabla **MatchHandGlobal** koja je u funkciji **Match Ability** postavljena na *true* ako je u pitanju desna ruka, a inače je postavljena na *false*. U slučaju da proslijeđena varijabla ima istinitu vrijednost funkcija će vratiti lokaciju i rotaciju desnog kontrolera, a inače će vratiti lokaciju i rotaciju lijevog.



Slika 82: Funkcija Choose Hand

Vrijednosti za vraćanje se izračunavaju pomoću makroa **Get Right Controller Transform** i **Get Left Controller Transform**. Makro su zapravo *blueprints* koje imaju ulaznu i izlaznu točku i koriste se za skraćivanje koda. U slučaju ovih makroa, kada se pozove jedan od dva navedena makroa prvo se dobavlja referenca na odgovarajući kontroler. Pomoću dobavljenih reference uzima se lokacija i rotacija kontrolera te se pravi nova transformacija koristeći tu lokaciju i rotaciju. Makroi vraćaju dobivenu transformaciju koja sadrži lokaciju i rotaciju kontrolera te će funkcija **Choose Hand** vratiti vrijednost koju dobije od makroa. Na slici je prikazan samo makro **GetRightControllerTransform** jer je makro **GetLeftControllerTransform** sasvim isti osim što je inicijalni objekt za kojeg tražimo transformaciju **Left Controller** dok je ovdje na slici 83 **Right Controller**.

Funkcija **Choose Hand** će vratiti lokaciju i rotaciju funkciji **Motion Fire Air** koja će koristiti te vrijednosti kako bi pomoću funkcije **SpawnActorFromClass** stvorila zračni projektil (**Air Projectile**) na toj lokaciji i rotaciji. Prilikom



Slika 83: Macro GetRightControllerTransform

kreiranja projektila njegov se vlasnik postavlja na *self*. Slika 84 prikazuje sve događaje koje smo kreirali zajedno sa ostalim funkcionalnosti.



Slika 84: Događaji za ispaljivanje sposobnosti

U nastavku je tablično prikazano koji događaj stvara koju klasu projektila

u funkciji `SpawnActorFromClass` kako ne bi bilo zabune.

Dogadaj	Klasa koju stvaramo
MotionFire Air	BP_AirProjectile
MotionFire Fire	BP_FireProjectile
MotionFire Ice	BP_IceProjectile
MotionFire Thunder	BP_ThunderProjectile

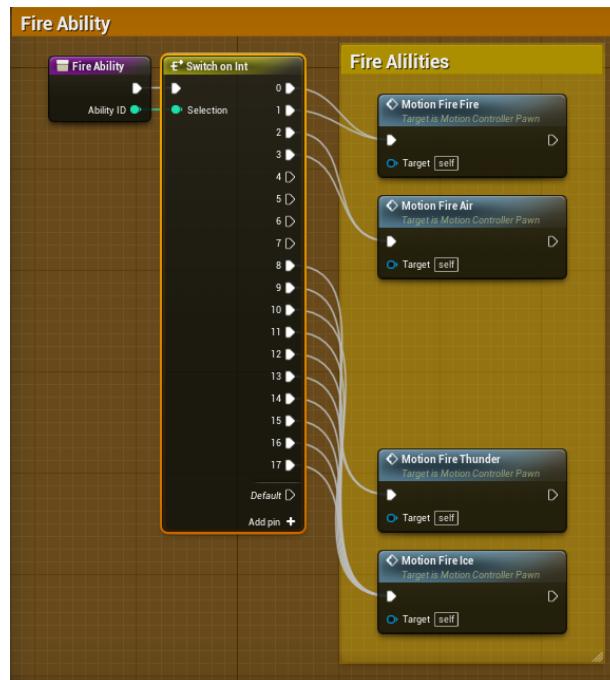
Tablica 13: Događaji koji stvaraju projektile

Dodane događaje moramo negdje i pozvati, a to radimo u funkciji `Fire Ability` koju smo kreirali u prethodnoj lekciji. Brišemo `Print String` funkciju i dodajemo događaje prema sljedećoj tablici 14;

Dogadaj	Okidan na izlaze
MotionFire Air	[2, 3]
MotionFire Fire	[1, 2]
MotionFire Ice	[10, 11, 12, 13, 14, 15, 16, 17]
MotionFire Thunder	[8,9]

Tablica 14: Pozivanje događaja koji stvaraju projektile

Neiskorišteni izlazi `Switch`a iskoristiti će se u sljedećoj lekciji, a kako spomenuti događaji izgledaju u svojem pozivanju može se vidjeti na slici 85.



Slika 85: Funkcija `Fire Ability`

5.2 RAD PROJEKTILA

U jednom od prethodnih odlomaka je spomenuta izrada „custom“ *collision channela* `TriggerArea`. Na isti je način izrađen i *collision channel Projectile* koji se dodaje projektilima i `defaultno` je postavljen na `block` što znači da se sa svime sudara.

Postavka	Vrijednost
Name	Projectile
Default Response	Block

Tablica 15: Postavke Projectile kolizijskog kanal

`Defaultnu` postavku projektila je potrebno izmijeniti kod nekih elemenata. Primjerice kod komponenti `Left Hand Motion Trigger` i `Right Hand Motion Trigger` potrebno je postaviti *projectile* na `ignore`. Također, potrebno je kolizije od `GrabSphere` u `MotionControllerPawn` i kolizije od `HandMesh` u `BP_MotionController` postaviti tako da ignoriraju projektile. To je isto tablično prikazano u sljedećoj tablici 22.

Postavka	Vrijednost
<hr/>	
MOTIONCONTROLLERPAWN	
LEFTHANDMOTIONTRIGGER	
Projectile	Ignore
<hr/>	
RIGHTHANDMOTIONTRIGGER	
Projectile	Ignore
<hr/>	
BP_MOTIONCONTROLLER	
<hr/>	
GRABSHERE	
Projectile	Ignore
<hr/>	
HANDMESH	
Projectile	Ignore
<hr/>	
BP_TRIGGERAREA	
<hr/>	
SVAKA SFERA	
Projectile	Ignore
<hr/>	

Tablica 16: Izmjene postavki kolizijskih kanala actora

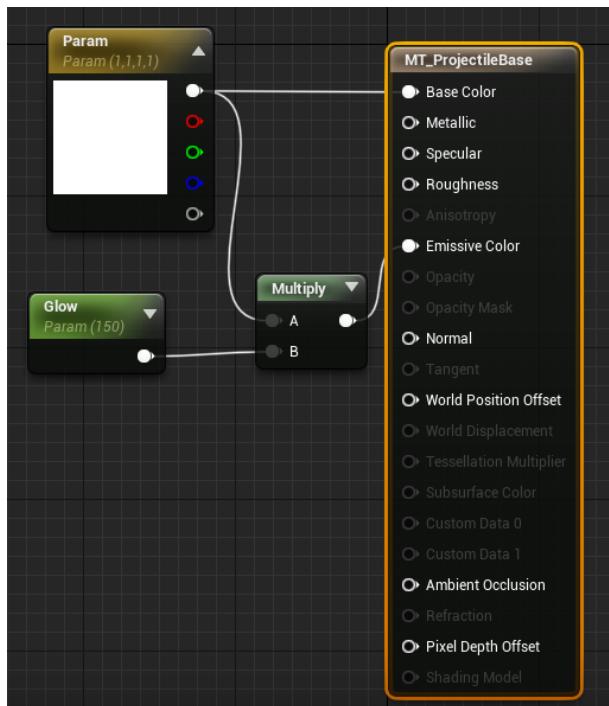
Projektil je potrebno uništiti nakon što nešto pogodi pa se na Event Hit dodaje funkcija `DestroyActor` sa određenom odgodom. Ako nakon nekog vremena projektil ništa ne pogodi potrebno je također uništiti ga. Iz tog se razloga opcija `Initial Life Span` postavlja na 10.

U igri postoji više vrsti projektila i više vrsti neprijatelja. Kako bi igra

Postavka	Vrijednost
Initial Life Span	10

Tablica 17: Promjene života projektila

bila zanimljivija i kako bi se „primoralo“ igrača da koristi sve sposobnosti svaki neprijatelj ima projektil protiv kojeg je slab, jak i ostale projektile no više o tome u lekcijama u nastavku. Slijedi izrada materijali za projektile kako bi ih mogli razlikovati. Oni se izrađuju tako da se stvori materijal za `MT_ProjectileBase` kojem se doda vektorski parametar pomoću kojeg će se odrediti *Base Color* te se dodaje skalarni parametar *Glow*. Umnožak *Glowa* i vektorskog parametra tvorit će *Emissive Color* materijala. Slika *bleuprinta* materijala nalazi se ovdje 86.



Slika 86: Jednostavni materijal

Od tako stvorenog materijala mogu se izraditi instance materijala. To se radi desnim klikom te odabirom opcije *Create Material Instance* za svaki od projektila tj. četiri puta. Svakoj od materijalnih instanci može se mijenjati vektorski parametar i *glow* čime će svaka materijalna instanca za svaki projektil biti jedinstvena. Imena instanci materijala su sljedeća.

- `MTI_AirProjectile`
- `MTI_FireProjectile`
- `MTI_IceProjectile`

- MTI_ThunderProjectile

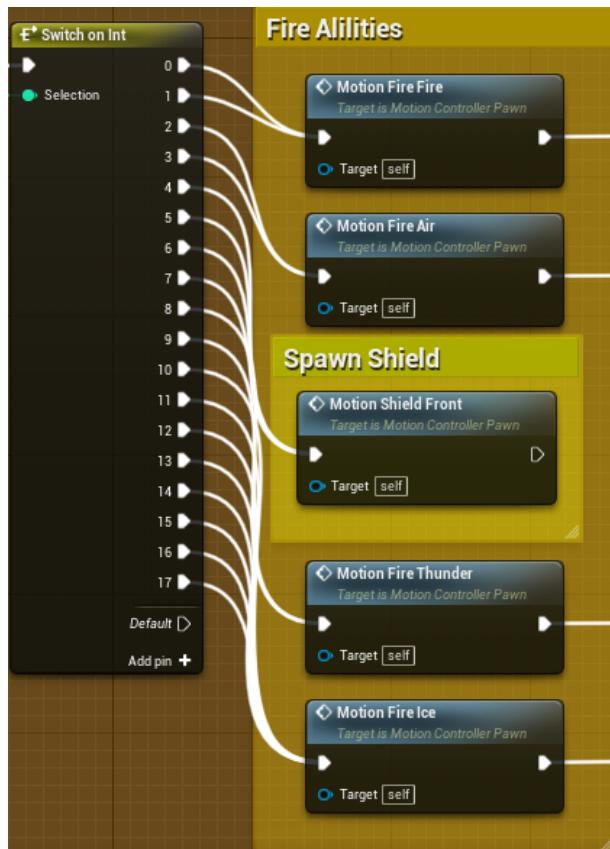
Tablično su prikazane postavke za svaku instancu materijala u tablici 18.

Postavka	Vrijednost
MTI_AIRPROJECTILE	
VECTOR PARAMETER VALUES	
R	0,3
G	0,95
B	1
SCALAR PARAMETER VALUES	
Glow	2
MTI_FIREPROJECTILE	
VECTOR PARAMETER VALUES	
R	0,4
G	0
B	0
SCALAR PARAMETER VALUES	
Glow	500
MTI_ICEPROJECTILE	
VECTOR PARAMETER VALUES	
R	0
G	0,65
B	0,70
SCALAR PARAMETER VALUES	
Glow	0
MTI_THUNDERPROJECTILE	
VECTOR PARAMETER VALUES	
R	1
G	1
B	0
SCALAR PARAMETER VALUES	
Glow	500

Tablica 18: Postavke *Projectile* kolizijskog kanal

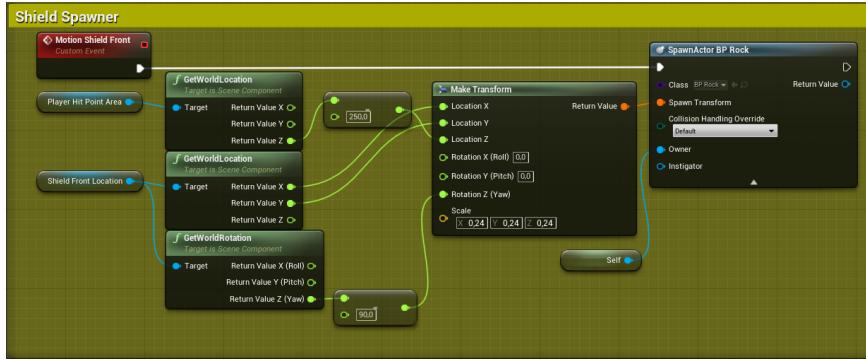
6 STVARANJE KAMENA

Stvaranje kamena je jedna od sposobnosti igrača. Moguće ga je stvoriti ako se sfere pogode u točnom redu kao i sve ostale sposobnosti. Nakon što se pogodi prava kombinacija okida se (eventualno, prilikom ulaska u **Fire Ability**) događaj **Motion Shield Front** koji služi za postavljanje kamena.



Slika 87: Pozivanje događaja za kamen

Na početku tog događaja se postavlja lokacija kamena prilikom stvaranja. Tako se Z os postavlja na 2,5 metra ispod komponente **PlayerHitPointArea** koja se nalazi u igraču i dio je **MotionControllerPawn**a. Osi X , Y te rotacija Z se postavljaju pomoću komponente **Shield Front Location** koja je dio **MotionControllerPawn**a. **Shield Front Location** je točka koja naravno ima svoje koordinate u odnosu na igrača te se uzimaju njezine X i Y koordinate i na tim se koordinatama stvara lokacija kamena. Rotacija kamena se dobiva iz rotacija **Shield Front Location** komponente tako da se uzima Z koordinata lokacije kamena te se umanjuje za 90 i na taj način se dobije rotacija za kamen. Te četiri vrijednosti argumenti su za funkciju **Make Transform** koja stvara varijablu u kojoj se nalaze lokacija i rotacija stvorenog kamena. Nakon toga se pomoću funkcije **SpawnActor** stvara actor **BP_Rock**.



Slika 88: Stvaranje štita

Glavni *blueprint* za kamen je BP_Rock. Taj *blueprint* sastoji se od jednog *mesh-a* naziva SM_Rock kojem je dodijeljen materijal Rock_6. SM_Rock je *static mesh* koji je pronađen na internetu i uveden je u projekt *drag & drop* metodom. S *meshom* su došli i materijali i teksture. Njegove *collision* postavke su prepostavljene zato što već blokira projektile i ignorira TriggerArea. Važna varijabla je ShieldHealth u kojoj se na početku nalazi vrijednost 50. Ta će se varijabla smanjivati kako ga neprijatelji pogađaju s projektilima. Pošto će jedan neprijateljev projektil iznositi 25, kamen će moći podnijeti 2 takva udarca.

Već je spomenuti *mesh* SM_Rock koji je zapravo *static mesh* za kamen. *Static mesh* moramo pretvoriti u uništivi *mesh* kako bi se on mogao raspasti. **Destructible mesh** je *dynamic mesh* koji može primiti *impact damage* kojeg primjenjuje objekt koji u njega udara. Kamen se razbija u manje komade nakon nekog vremena ili nakon što je primio dovoljno štete. **Destructible mesh-ovi** stvaraju se iz *static mesh-eva*.

Slika prikazuje primanje štete kamenu nakon 5 sekundi postojanja. Funkcija SetActorEnableCollision služi za isključivanje i uključivanje kolizijskih postavki te se ovdje koristi kako bi nakon raspada kamena nakon dodatnog vremena pao kroz pod. Ovdje jedino nedostaje objekt **Destructible Component** pa će se on u nastavku kreirati.



Slika 89: Uništavanje kamena

Za pretvaranje *static mesh-a* u **Destructible mesh** potrebno je omogućiti *plugin* Apex Destruction. Nakon što ga se omogući potrebno je ponovno

pokrenuti Unreal Engine. Nakon što se ponovno uđe u Unreal Engine može se stvoriti **Destructible Mesh**. To se radi tako da se desno klikne na željeni *static mesh* te se odabere opcija *Create Destructible Mesh* i odabere *Yes*. Novi *mesh* se nazove **DM_Rock**.

Postavke **Destructible mesha** su prikazane tablično u sljedećoj tablici .

Postavka	Vrijednost
VORONOI	
Cell Site Count	25
GENERAL	
Random Seed	76

Tablica 19: Promjene života projektila

Jedino je još preostalo promijeniti ili dodati uništivu komponentu pod komponentama te kao **Destructible Mesh** odabrati **DM_Rock**.

6.1 ZAKLJUČAK

U ovoj kratkog lekciji naučili smo stvarati kamen te uništiti ga kad pretrpi određenu štetu.

Kao dodatni izazov kod izrade projekta pokušajte dodati još nekoliko sličnih objekata kamenu. Neka neki od tih objekata budu krhkiji, a drugi neka su malo izdržljiviji.

7 NEPRIJATELJI

S arhitekturalnog aspekta, neprijatelji su izrađeni slično kao i projektili. Postoji više neprijatelja, isto kao i projektila. Svaki od neprijatelja je implementiran na način da nasljeđuje nadklasu `BP_Enemy` i sadrži neke posebnosti koje su karakteristične samo za njega. Na taj način je izrazito olakšana implementacija neprijatelja jer se većina zahtjevnih funkcija, poput primanja štete, mora implementirati samo jednom.

Nakon izrade `BP_Enemy` koji je tipa `Character` izrađuje se njegovo dijete (podklasa) `BP_Wizard` kojem se dodaje privremena *mesh* komponenta. *Mesh* se satoji od stošca i kugle te predstavlja izgled lika na jednostavan način kako bi se s time moglo testirati. Također, dodaje se i `FiringPoint` koja predstavlja mjesto na kojem će se stvarati neprijateljski projektili. Ta komponenta je tipa `Arrow` i usmjerena je u istom smjeru kako i neprijatelj.

Nakon toga potrebno je napraviti podklasu klase `BP_Projectile` i pripadnog materijala. Podklasa će se zvati `BP_Enemy Projectile`. U tom *blueprintu* se nadjačava funkciju roditelja `Apply Damage` koja se poziva prilikom događaja `Event Hit`. Početna i maksimalna brzina `BP_Enemy Projectile` se postavlja na 700, a gravitacija na 0 kako projektil ne bi opadao s vremenom kao što to rade projektili koje igrač ispaljuje. Sve promjene u odnosu na roditeljski blueprint mogu se vidjeti u sljedećoj tablici ??.

Postavka	Vrijednost
Initial Speed	700
Max Speed	700
Projectile Gravity Scale	0

Tablica 20: Promjene neprijateljskog projektila

`BP_Enemy Projectile` ima svoju podklasu `BP_WizardProjectile`. `BP_WizardProjectile` je projektil kojeg će neprijatelji ispaljivati pa se kod njega implementira funkcija za ispaljivanje projektila. U toj funkciji se uzima lokacija `Firing Pointa` kao početna pozicija projektila (i lokacija stvaranja) i položaj igrača. Referenca se dobija pomoću variabile `PlayerReference` u koju se prilikom eventa `Begin Play` postavlja referenca na igrača. Slika 90.

Potrebno je dodati novi `Collision channel` `EnemyProjectile` čija je pretpostavljena vrijednost `Block`. Detalji u tablici 21.

Međutim, `BP_TriggerArea` će ignorirati taj kanal, isto kao i `Collision Capsule` komponenta i `BP_Enemy Projectile` jer je potrebno da projektili ignoriraju sami sebe, odnosno da se ne mogu sudariti s drugim neprijateljskim projektilima. Detalji u tablici 22.

Postavka	Vrijednost
Name	EnemyProjectile
Default Response	Block

Tablica 21: Neprijateljski kanal projektila

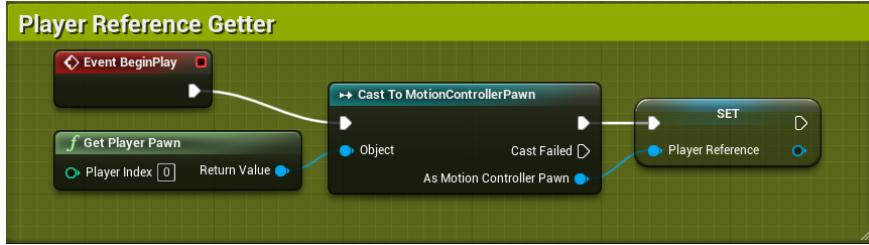
Postavka	Vrijednost
BP_TRIGGERAREA	
SVAKA SFERA	
EnemyProjectile	Ignore
MOTIONCONTROLLERPAWN	
COLLISIONCAPSULE	
Projectile	Ignore
BP_ENEMYPROJECTILE	
SPHERE	
Projectile	Ignore

Tablica 22: Izmjene postavki kolizijskih kanala *actora*

BP_Enemy pripada klasi **Character** što znači da je on **Pawn** koji se može kretati kao čovjek. **Pawn** je *actor* koji može biti kontroliran. Kao i kod projektila svaki od neprijatelja je podklasa klase BP_Enemy što znači da je svaki neprijatelj kreiran odabirom opcije **Create Child Blueprint Class**. S obzirom na to da je BP_Enemy klasa koja sadrži funkcije za rad neprijatelja, ali se neće stvarati u igri, nije joj potrebno dodjeljivati *meshes*. Podklasama BP_Enemyja potrebno je dodjeljivati *meshes* jer se hoće stvarati pa je potrebno vidjeti kako funkcioniraju. Pošto će i neprijatelji koristiti projektile, možemo se koristiti kreiranim projektilom za korisnika (klasa BP_Projectile) na način da kreiramo *child blueprint*. Jedina je razlika tko će kome nanositi štetu. Za neprijateljske projektile potrebno je stvoriti poseban **collision channel** zato što će na taj način neprijateljski projektili i projektili igrača se ignorirati međusobno.

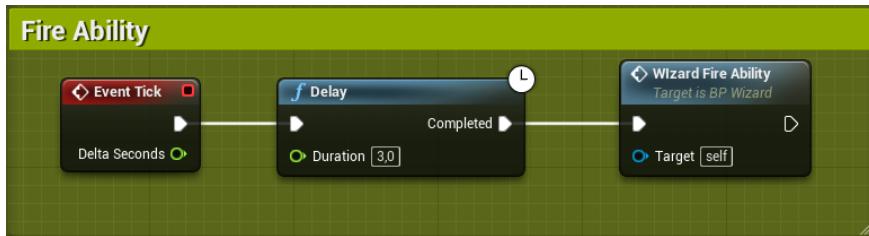
Posao neprijatelja je da pokušaju pogoditi igrača i tako poraziti igrača. Kako bismo to napravili moraju znati lokaciju igrača da bi mogli na njega pucati projektile. Iz tog razloga svaki neprijatelj će u trenutku stvaranja dobiti lokaciju igrača tako što će dohvatiti referencu na prvi Player Pawn koji označava ono što se kontrolira u igri od strane korisnika. U ovom slučaju postoji samo jedan pa se stoga referencira na prvi. Nakon toga će se Player Pawn *castati* u objekt tipa MotionControllerPawn što se može raditi zato što postoji samo jedan Player Pawn i on je uvijek tipa MotionControllerPawn te će se pomoći toga dobiti referenca na igrača

pomoću čega se lako može dobiti lokacija. Slika 90 to prikazuje detaljnije.



Slika 90: Dobivanje reference na igrača

Za početak ćemo ispaljivati sposobnost svake tri sekunde. Na slici 91 se vidi ispaljivanje sposobnosti **Fire Ability**



Slika 91: Ispaljivanje *Fire abilityja*

Funkcija koja se poziva nalazi se na slici 92. Ona stvara objekt **BP_WizardProjectile** pa je potrebna transformacija iz funkcije **MakeTransform**. Lokaciju za tu funkciju dobiva iz funkcije **GetWorldLocation** čiji je objekt **FiringPoint**, a rotaciju dobiva preko **FindLookAtLocation**. Spomenuta funkcija dobiva dva vektora od kojih je početak **FiringPoint** od protivnika, a meta je igrač. Pozicija igrača dobiva se preko reference igrača. Vrijednost visine za taj vektor nasumično se smanjuje za raspon od 0 do 60. Detalje možemo vidjeti na slici 92.



Slika 92: Dobivanje reference na igrača

Svakoj podklasi klase **WizardEnemy**ja kao i njemu potrebno je dodati funkciju **FireAbility** koja poziva događaj **WizardFireAbility** te koja se vidi na slici 93.

U **BP_EnemyProjectile** dodajemo događaj **OnEventHit** koji poziva funkciju **ApplyDamageEnemyProjectile** i njoj prosljeđuje **DamagedActor** i vektor *impact location*.

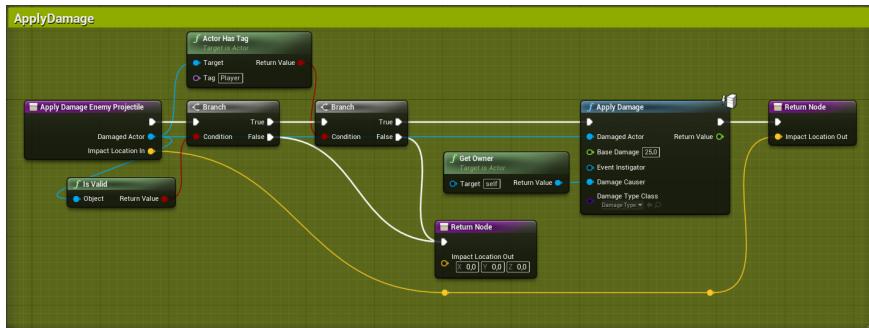


Slika 93: Dobivanje reference na igrača



Slika 94: Dobivanje reference na igrača

Funkcija `ApplyDamageEnemyProjectile` prvo provjerava je li pogodjeni objekt valjan, zatim je li objekt ima oznaku `Player` te ako ima, poziva funkciju `ApplyDamage` kojoj se proslijeduje `DamagedVector`, inače ništa. Količina štete koja se radi je 25.



Slika 95: Dobivanje reference na igrača

Na svakoj podklasi projektila potrebno je *overrideati* funkciju `ApplyDamageEnemyProjectile` tako da se nakon poziva roditeljske funkcije pozove funkcija `DestroyActor`.

7.1 UMJETNA INTELIGENCIJA

Kako bi neprijatelji uopće znali da što trebaju raditi, potrebno im je dodijeliti umjetnu inteligenciju koja će im reći kako se trebaju ponašati u određenim situacijama. Umjetna inteligencija se neprijateljima dodjeljuje čim se stvore.



Slika 96: Dobivanje reference na igrača

7.1.1 OSNOVE STABLA PONAŠANJA

BT_Enemy je stablo ponašanja (en. *behavior tree*) pomoću kojeg se određuje kako će se neprijatelji ponašati. Stabla ponašanja se stvaraju pritiskom na desni klik miša i odabirom opcije *Behavior Tree* koja se nalazi pod opcijom *Artificial Intelligence*. Stabla ponašanja u Unreal Engineu su uređena pomoću komponenti *selector* i *sequence*. *Selector* je čvor koji služi za odabiranje zadatka koji će se izvršavati. *Selector* uvijek izabere prvi zadatak koji može izvršiti od onih koji su mu ponuđeni (gledući zadatke s lijeva na desno). *Sequence* izvršava sve zadatke redom (slijeva na desno) sve dok ne dođe do nekog zadatka kojeg ne može izvršiti. Kada *sequence* ne može izvršiti zadatak penje se na nadčvor.

U korijenskom čvoru stabla BT_Enemy nalazi se element BB_Enemy. BB_Enemy je *blackboard*, a *blackboard* je zapravo mjesto u kojem se mogu pohranjivati i iz kojeg se mogu čitati podaci potrebni za stvaranje odluka. *Blackboard* se stvara pritiskom na desni klik miša i odabirom opcije *Blackboard* koja se nalazi pod opcijom *Artificial Intelligence*. To znači da će se u BB_Enemy spremati podatci koji će se dobaviti od neprijatelja, a koji će služiti BT_Enemy stablu kako bi se donosile odluke. Varijable ili, bolje rečeno, ključevi, u koje će se pospremati vrijednosti izrađuju se na samom početku izrade stabla. Bitno je ključevima dodijeliti klasu jednako onome što će se u ključeve pohranjivati.

Prvi ključ od šest kojeg ćemo spomenuti je *Enemy Actor* koji je tipa *Actor* u kojeg se pohranjuje referenca na igrača (zato što je igrač u ovom slučaju neprijatelj neprijatelj). Drugi je ključ *HasLineOfSight* koji je istinit ako neprijatelj vidi igrača, a lažan ako ga ne vidi. Treći je ključ *PatrolLocation*. To je vektor u kojeg se pohranjuje lokacija s koje će neprijatelj početi s ophodnjom. Četvrti ključ, *DodgeLocation* je vektor u kojeg se pohranjuje lokacija bliska onoj na kojoj se trenutno nalazi neprijatelj. Na tu će se lokaciju neprijatelj pomaknuti kako bi igraču otežao gađanje. Zadnji je ključ *SelfActor* koji je tipa *Actor*.

BB_Enemy prvi je čvor BT_Enemy stabla odlučivanja. Iz BB_Enemy se ide u AI Root koji je zapravo selector čvor iz kojeg se bira koju će akciju izvršavati neprijatelj. Prvi čvor u kojeg će AI Root ući je dekorator. Dekoratori su

uvjeti koji se pridružuju čvorovima. Pomoću dekoratora se provjerava može li se izvršiti grana stabla pod njima (odnosno mogu li se izvršiti aktivnosti u njihovim podčvorovima). U ovom konkretnom dekoratoru se provjerava je li vrijednost varijable `HasLineOfSight` (koja je pohranjena u `BB_Enemy`) postavljena na `True`.

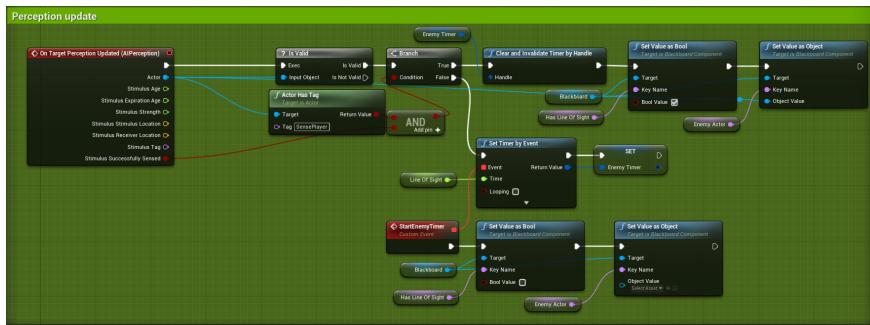
7.1.2 PERCEPCIJA NEPRIJATELJA

Kako bi se moglo odlučiti vidi li umjetna inteligencija igrača ili ne, potrebno je koristiti `AIPerception` komponentu. Ta se komponenta naziva `AIPerception` te ju je moguće izraditi pritiskom na gumb `Add Component` i odabirom `AIPerception` komponente. Klasa u kojoj se nalazi `AIPerception` je `AIController`, a ona se izrađuje desnim pritiskom na `Blueprint Class` i pretraživanjem polja `All Classes`. Naziv klase `AIController` je `AIC_EnemyController`. Napravljenom kontroleru potrebno je dodati koje će stablo koristiti kako bi znao što trebaju raditi `Pawnovi` kojima upravlja. `AIPerception` komponente služe kako bi `Pawnovi` (u ovom slučaju neprijatelji) mogli osluškivati, kako bi znali što osluškuju, koji su parametri za to što osluškuju i kako reagirati kada se dogodi to što su osluškivali. Svakoj `AIPerception` komponenti može se postaviti na koji način će ona osluškivati događaj na koji treba odreagirati. U slučaju s `AIPerception` komponentom za postavljanje varijable `HasLineOfSight` koristit će se `AI Sight config` koji označava da će neprijatelj pokušati vidjeti gdje se igrač nalazi umjesto da to pokušava shvatiti sluhom. To je postavljeno tako zato što je vid najintuitivniji i najlakši način za detekciju nekog `pawna`. Nakon postavljanja `AI Sight configa` potrebno je postaviti koliko će daleko neprijatelj moći vidjeti, kada će neprijateljevo vidno polje početi slabiti i koliko će široko biti neprijateljevo vidno polje. Na taj način se može namjestiti koliko blizu neprijatelj mora biti da bi video igrača. Također potrebno je postaviti `Dominant Sense` na `AISense_Sight`. Potrebno je i označiti opciju `Detect Neutrals` bez koje se ne bi moglo ništa registrirati.

Kad neprijatelj vidi igrača okida se događaj `On Target Perception Updated` (`AIPerception`). Prva stvar koja se provjerava u `On Target Perception Updated` (`AIPerception`) je ima li `actor` koji viđen tag `SensePlayer`. Jedino sfera koja se nalazi na mjestu igrača ima tu oznaku te ju sve ignorira što se tiče kolizije i ne može se vidjeti u igri (opcija `visible` na `False`). Sfera s oznakom je `static mesh` od `pawna AISense`. `AISense` je `pawn` zato što će ga na taj način `AIPerception` moći registrirati. Takvu je sferu bilo potrebno izraditi iz razloga što neprijatelj nije mogao vidjeti niti jednu drugu komponentu vezanu za igrača. Kako bi sfera stalno bio na istom mjestu kao i igrač, njegova lokacija se na `tick` ažurira na lokaciju i rotaciju kamere igrača koristeći varijablu `Sense Reference` kao referencu na tog `actor-a`. `Sense Reference` se postavlja prilikom stvaranja sfere na događaju

Event **BeginPlay**. Sense Reference uzima lokaciju komponente **Sense Location** koja je dio **MotionControllerPawn**.

On **Target Perception Updated** (**AIPerception**) ima dva argumenta. Prvi argument je tipa **Object** i naziva **Actor** u kojeg se pohranjuje referenca na *actora* kojeg je neprijatelj vidoio. Drugi argument je **Bool** varijabla **Stimulus Successfully Sensed** koja je istinita ako je **AIPerception** uspešno detektirao kad nešto uđe, a **False** ako je detektirao izlazak. Prva stvar koja se provjerava je je li **Stimulus Successfully Sensed** istinit i ima li **actor** na kojeg se referencira varijabla **Actor** tag naziva **SensePlayer**. Ako je jedna od ili obje tvrdnje lažne onda se postavlja timer za događaj **StartEnemyTimer** pomoću funkcije **Set Timer by Event**. Ta funkcija vraća timer naziva **Enemy Timer** koji će omogućiti pozivanje funkcije **StartEnemyTimer** u rasponima u kojima je to određeno varijablom **Line of Sight** koja je tipa **Float**. Ta se varijabla prosljeđuje u **Set Timer by Event** i označava vrijeme nakon kojeg će se pozvati funkcija za pozivanje događaja.



Slika 97: Percepcija protivnika

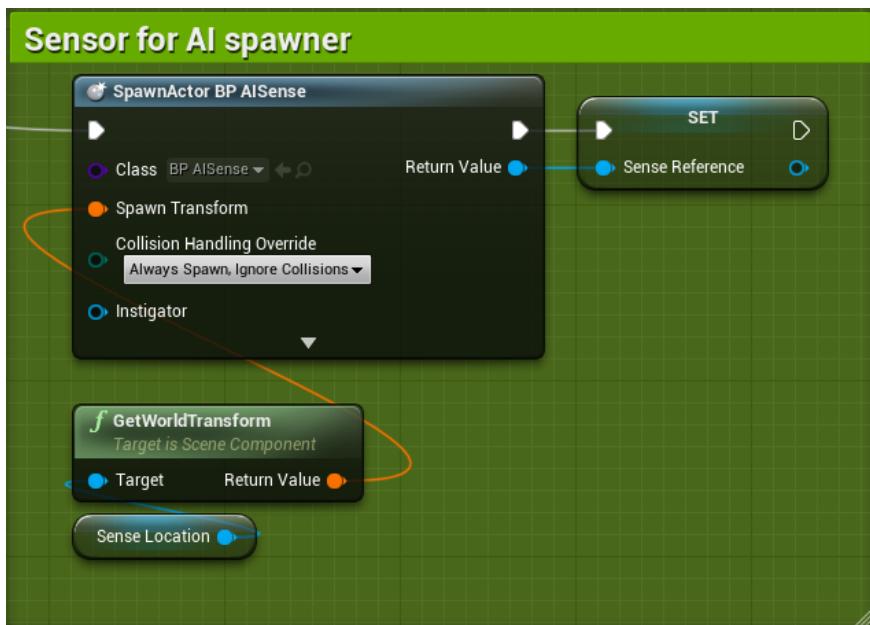
7.1.3 POSTAVLJANJE VARIJABLE ZA GLEDANJE IGRAČA

U slučaju da je **Stimulus Successfully Sensed** istinit i *actor* tag jednak **SensePlayer**, onda se prvo briše brojač **Enemy Timer** što znači da se funkcija **StartEnemyTimer** više neće pozivati. Nakon toga se poziva funkcija **Set Value as Bool** koja postavlja vrijednost varijable **Has Line Of Sight** koja se nalazi u *actoru Blackboard* (**BB_Enemy**) na **True**. Nakon toga se pomoću funkcije **Set Value as Object** za vrijednost varijable **Enemy Actor** koja se nalazi u Blackboardu (**BB_Enemy**) postavlja varijabla **Actor** koja je argument funkcije **On Target Perception Updated** (**AIPerception**).

U prošlom je odlomku spomenuta funkcija **StartEnemyTimer**. Ta funkcija postavlja vrijednost varijable **Has Line Of Sight** na **False** i vrijednost varijable **Enemy Actor** na **null** na isti način kako se te vrijednosti postavljaju u funkciji **On Target Perception Updated** (**AIPerception**). Na taj će način varijabla **Has Line Of Sight** biti stalno na **False** sve do trenutka

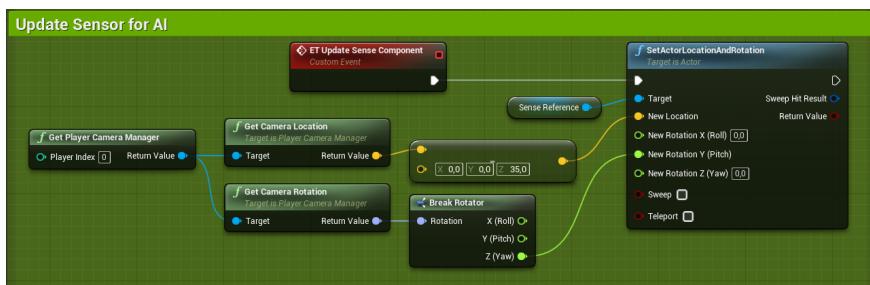
kada neprijatelj vidi igrača. U tom trenutku više se ne poziva funkcija `StartEnemyTimer` koja postavlja `Has Line Of Sight` na `False` i promjeni se vrijednost varijable `Has Line Of Sight` na `True`. Nakon što neprijatelj više ne vidi igrača ponovno će se postaviti brojač na funkciju te će nakon četiri sekunde (koliko treba brojaču da aktivira funkciju) neprijatelj prestati loviti igrača. Ukoliko igrač unutar te četiri sekunde ponovno uđe u vidno polje poništiti će se brojač i neprijatelj uopće neće prestati loviti igrača.

Neprijatelj ne može vidjeti VR igrača pa ćemo napraviti jednog *actora* kojeg ćemo postavljati na poziciju igrača, učiniti ga nevidljivim, a neprijatelj će ga i dalje vidjeti. Na početak igre stvara se taj *actor*, bilježimo njegovu referencu.



Slika 98: Dobivanje reference na igrača

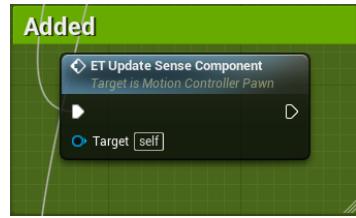
Važno je da sferi koju dodajemo u tog *actora* damo oznaku *Player*. Poziciju tog *actora* ažuriramo na *tick* što se vidi na slici.



Slika 99: Dobivanje reference na igrača

Taj se događaj poziva na `Event Tick` i može se vidjeti na slici 100.

U neprijateljskom kontroleru dodajemo događaj `Event OnPosses` koji po-



Slika 100: Dobivanje reference na igrača

kreće funkciju RunBehaviourTree kojem je izabran BT_Enemy.



Slika 101: Dobivanje reference na igrača

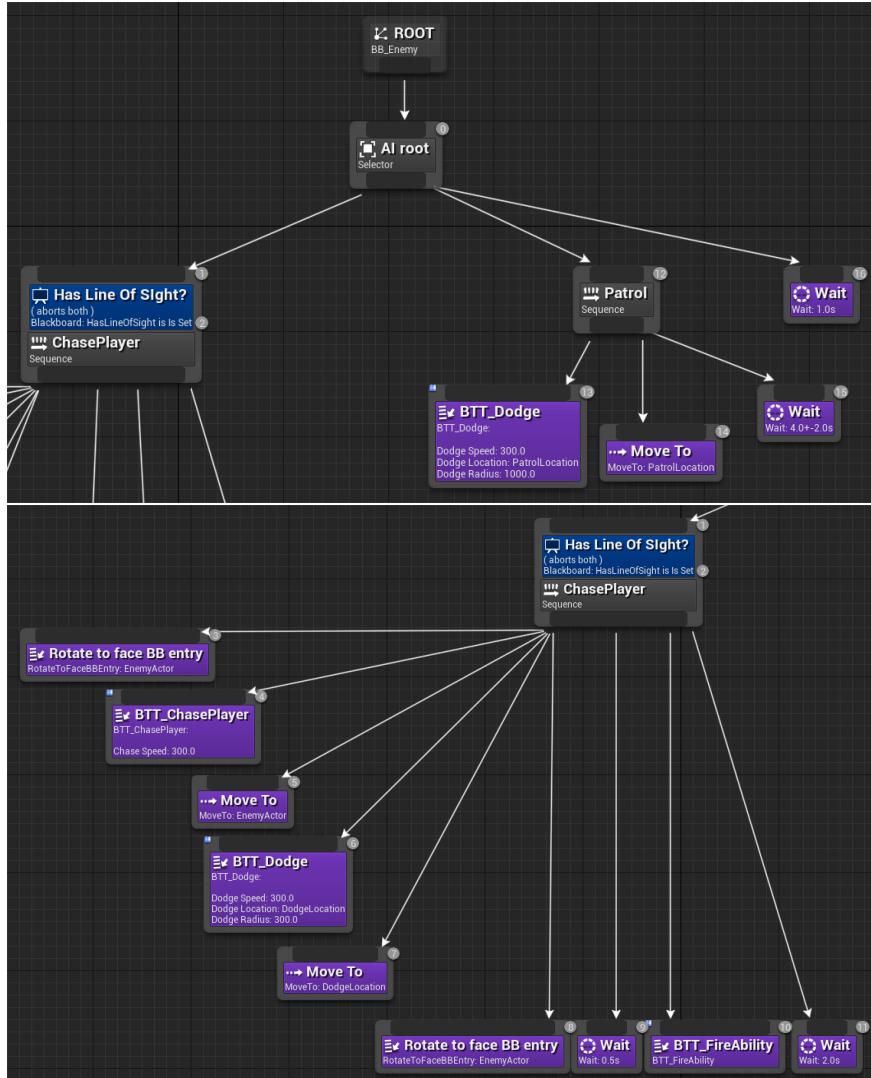
7.1.4 KADA NEPRIJATELJ VIDI IGRAČA

Sada kada je opisano kako se varijabla `Has Line Of Sight` postavlja na `True` i `False` možemo se vratiti na `BT_Enemy`. Dakle, ako je u vrijednost varijable `Has Line Of Sight` istinita, dekorator će pustiti da se izvršava `Sequence ChasePlayer`. Dekorator radi na način da se prilikom promijene vrijednosti varijable `Has Line Of Sight` ponovno provjerava je li ona istinita ili ne te se tako osigurava da neprijatelj zbilja radi ono što treba raditi kada vidi ili ne vidi igrača. `Sequence ChasePlayer` sastoji se od nekoliko zadataka. Zadatci se izrađuju tako da se u izborniku od `BT_Enemy` pritisne na opciju *New Task*. Nakon izrade zadatka može ga se u mapi gdje je pohranjen preimenovati po želji. Svaki zadatak kreće `Event Receive Execute AI`-om što označava da je funkcija događaj vezan za umjetnu inteligenciju i završava s `Finish Execute` u kojeg se prosljeđuje `True` ili `False` prema tome je li događaj uspješno izvršen. Bitno je da varijable koje se koriste u zadacima budu javno vidljive kako bi se mogle pozivati u stablu.

Prvi zadatak te sekvene je `Rotate to face BB entry` koja naređuje neprijatelju da se okreće u smjer gdje se nalazi igrač. Idući zadatak za neprijatelja je postavljen u `BTT_ChasePlayer`. U njemu se brzina hodanja neprijatelja postavlja na 500 tako da se pozove funkcija `Update Walk Speed` (ukoliko je pawn nad kojim je pozvana funkcija objekt od `BP_Enemy`). U `Update Walk Speed` se postavlja varijabla `Max Walk Speed` na 500. Varijabla `Max Walk Speed` je dio `BP_Enemy`.

U klasi `BP Enemy` dodajemo funkciju `UpdateWalkSpeed` koja postavlja varijablu `MaxWalkSpeed`, a meta joj je `CharacterMovement`.

S obzirom na to da ćemo sada napraviti da neprijatelj puca iz `AI Controllera` potrebno je maknuti pucanje na `tick` koje smo postavili u klasi `BP Wizard`.

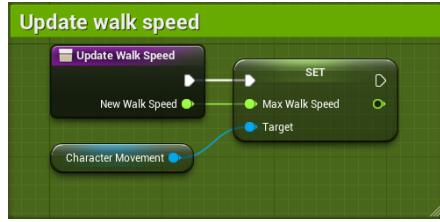


Slika 102: Stablo odlučivanja



Slika 103: Proganjanje igrača

Nakon što je promijenjena brzina kretanja neprijatelj dobiva zadatak da dođe do igrača. Za to se koristi funkcija `Move To`. Funkciji `Move To` daje se varijabla `Enemy Actor` kako bi neprijatelj znao gdje treba ići i prihvatljivi radijus (koliko daleko od neprijatelja smije biti) od 1000.

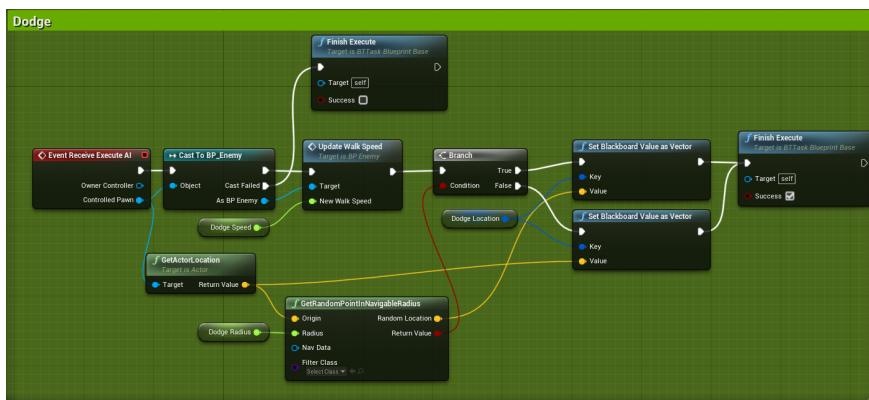


Slika 104: Progjanjanje igrača



Slika 105: Progjanjanje igrača

Nakon što dođe do igrača, neprijatelj dobija novi zadatak BTT_Dodge. U BTT_Dodge se opet mijenja brzina kretanja neprijatelja na isti način kao i u BTT_ChasePlayer. Nakon toga se nasumično bira lokacija bliska onoj na kojoj je neprijatelj trenutno. To se radi pomoću funkcije `GetRandomPointInNavigableRadius` koja prima dva argumenta. Prvi je `Origin` u kojoj je zapisana početna lokacija za koju se uzima lokacija neprijatelja, a druga je `Radius` u kojoj piše dozvoljeni radijus oko `Origin` koji se postavlja na 1,5. Ukoliko ta funkcija uspešno vrati neku lokaciju varijabla `Dodge Location` (varijabla od `BB_Enemy`) se na nju postavlja, a inače se varijabla `Dodge Location` postavlja na trenutnu lokaciju neprijatelja.



Slika 106: Izbjegavanje projektila

Nakon što je postavio `Dodge Location` pokreće se funkcija `Move To` te neprijatelj hoda do lokacije pohranjene u `Dodge Location`. Onda se ponovo

okreće prema igraču pomoću funkcije `Rotate to face BB entry`. Nakon toga čeka 0,5 sekundi (funkcija `Wait`).

Nakon što je završio s čekanjem, neprijatelj ima zadatak zadan u `BTT_FireAbility`. U `BTT_FireAbility` se poziva funkcija `Wizard Fire Ability` ako je kontrolirani *pawn* klase `BP_Wizard`.



Slika 107: Izbjegavanje projektila

U funkciji `Wizard Fire Ability` poziva se događaj istog imena. U događaju `Wizard Fire Ability` poziva se funkcija `SpawnActor BP Wizard Attack Projectile` koja stvara projektil, za vlasnika projektila uzima *actora* `BP_Wizard`, za lokaciju se uzima lokacija komponente `Firing Point` koja se nalazi na mjestu gdje završava ruka kod animacije za pucanje, a rotacija se dobije pomoću funkcije `Find Look at Rotation` koja prima početnu i završnu lokaciju i od toga stvori rotaciju. Za početnu lokaciju se uzima lokacija komponente `Firing Point`, a za završnu lokaciju kamere igrača, odnosno lokacija igrača. Pri određivanju završne lokacije od *Z* osi se oduzima nasumičan broj od 0 do 60. Na taj način će neprijatelj uvijek gađati igrača u drugu točku, ali ga svejedno neće promašiti jer se ne mijenja *X* ni *Y* os, a za visinu (*Z* os) se uzima visina glave pa se uzimanjem maksimalno 60 centimetara od visine ne mijenja činjenica da projektil ide u igrača. Od dobivene lokacije i rotacije se radi transform koji se šalje funkciji `SpawnActor BP Wizard Attack Projectile`. Nakon toga se završava `BTT_FireAbility` i nakon toga neprijatelj čeka 2 sekunde prije nego što kreće ispočetka izvršavati sekvencu `ChasePlayer`. S obzirom da je neprijateljsko pucanje implementirano na ovaj način, onesposobljena je prijašnje pucanje na *tick*.

7.1.5 KADA NEPRIJATELJ NE VIDI IGRAČA

Ukoliko je varijabla `Has Line Of Sight` negativna, neprijatelj ne može ući u sekvencu `ChasePlayer` već mora ući u sekvencu `Patrol`. Prvi zadatak kojeg neprijatelj mora izvršiti u toj sekvenci jest `BTT_Dodge`, ali s drukčijim radiusom nego u prethodnom primjeru. Nakon toga se neprijatelj pomiče (pomoću `Move To`) na lokaciju `PatrolLocation` koja se postavila na isti način kao i `Dodge Location` u `BTT_Dodge`. Nakon toga neprijatelj treba

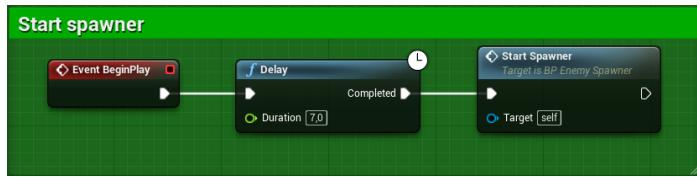
čekat jedno vrijeme u rasponu od 2 do 6 sekundi. To se postiže na način da se u funkciji `Wait` postavi varijabla `Wait Time` na 4 sekunde, a varijabla `Random Deviation` na 2 sekunde. Nakon toga se čeka još 1 sekunda i onda se u sekvencu kreće ispočetka.

7.1.6 KADA NEPRIJATELJ UMRE

Kada neprijatelj umre, okinut će se događaj `Dead` koji će pozvati funkciju `Un Possess` koja će maknuti umjetnu inteligenciju s neprijatelja kako se on ne bi mogao micati.

7.2 STVARANJE NEPRIJATELJA

Neprijatelji se u igri stvaraju pomoću *actora* `BP_EnemySpawner`. Taj *actor* će stvoriti neprijatelja na nekoj lokaciji ukoliko su ispunjeni određeni uvjeti. Sam *spawner* se stvara pri početku igre te se na njegov Event `BeginPlay` pokreće funkcija `Start Spawner`.

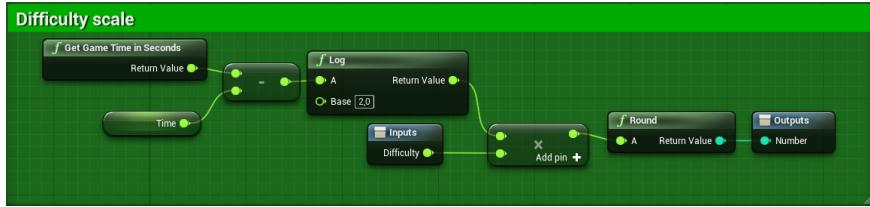


Slika 108: Pokretanje *spawnera*

Događaj `StartSpawner` služi za izradu brojača za pozivanje funkcije `Spawn Enemies`. Kao vrijeme pozivanja funkcije prosljeđuje se varijabla `Spawn Interval` (prepostavljeno 3). *Actor* `BP_Spawner` sastoji se od komponente `Box Collision` naziva `Spawn Area`. `Spawn Area` ima sve *collision channel* postavljene na *ignore* jer se ne smije sudarat ni sam čim.

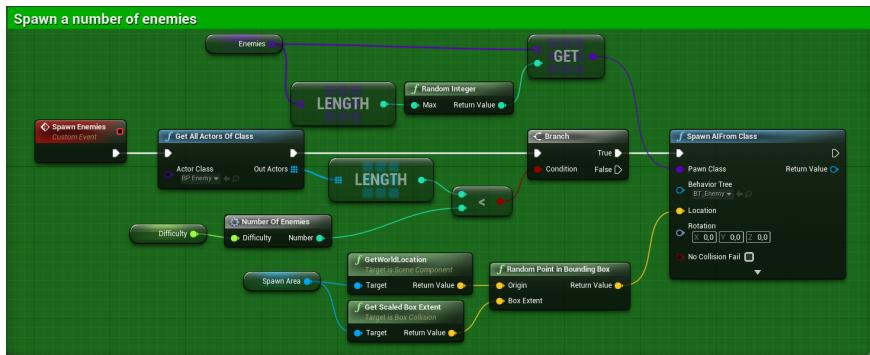
Funkcija `SpawnEnemies` služi za stvaranje neprijatelja. Prva stvar koju ta funkcija radi je provjeravanje koliko postoji *actora* klase `BP_Enemy`. Broj *actora* klase `BP_Enemy` se uspoređuje s maksimalnim brojem neprijatelja na mapi koji se dobije pomoću makroa `Number of Enemies`. `Number of Enemies` uzima u obzir trenutno vrijeme i vrijednost varijable `Difficulty` (defaultno 0,5) u kojoj je zapisana težina igre te se pomoću određene logaritamske formule računa koji je dozvoljen broj neprijatelja u zadanim trenutku.

S obzirom da se koristi logaritamska funkcija dozvoljen broj neprijatelja u početku će brzo postojat veći, a nakon određenog vremena dozvoljen broj će sve sporije rasti čime se produljuje moguće vrijeme igranja igre. Vraćajući se na funkciju `SpawnEnemies`, ukoliko je broj neprijatelja manji nego broj kojeg je vratio makro, može se stvoriti još neprijatelja pa se poziva funkcija



Slika 109: Makro Difficulty Scale

Spawn AIFrom Class. Za lokaciju na koju će se pojaviti neprijatelj uzima se nasumična lokacija komponente **Spawn Area** koja se može skalirati prema veličini *levela* kako bi se neprijatelji mogli pojaviti na što nasumičnijem mjestu. **Spawn Area** je jedina komponenta *actora* **BP_Spawner** koja je tipa **Box Collision**.

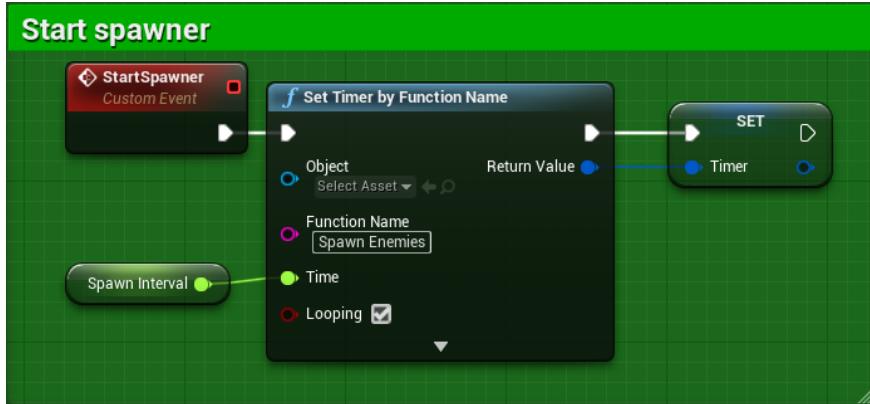


Slika 110: Događaj Spawn Enemies

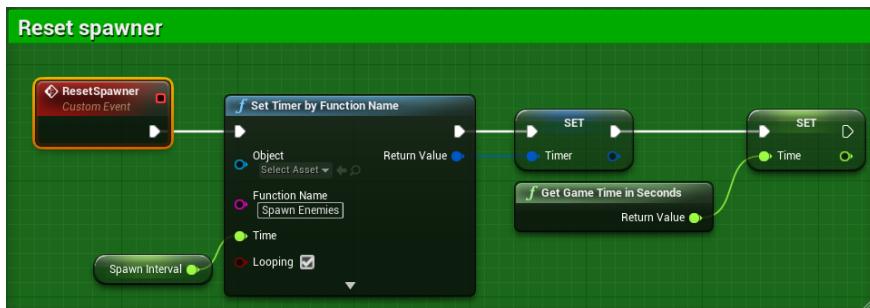
Postoji više vrsta različitih neprijatelja. Svi su neprijatelji sadržani u polju **Enemies** koje je tipa **Pawn**. Tip **Pawn** označuje sve *actore* koji mogu biti opsjednuti. U slučaju igrača mi smo ti koji ga opsjedamo (korisnici koji igraju igru), a u slučaju neprijatelja to je umjetna inteligencija koja kontrolira njegove akcije. Kada je vrijeme za stvaranje neprijatelja, uzima se jedan nasumično odabran broj manji od duljine polja **Enemies** te se stvara neprijatelj koji se u polju **Enemies** nalazi pod odabranim nasumičnim brojem. Prilikom stvaranja neprijatelju se dodjeljuje umjetna inteligencija pomoću funkcije **Spawn AIFrom Class** u koju se prosljeđuje odabrani neprijatelj. U funkciji **Spawn AIFrom Class** također je određeno da će neprijatelju biti dodijeljena umjetna inteligencija **BT_Enemy**. Trenutno je u polju **Enemies** samo jedan neprijatelj, međutim s dalnjim razvojem bit će ih dodano više.

Pokretanje *spawnera* nalazi se u klasi **BP Enemy Spawner**. Na slici se pokazuje okidanje događaja **Start Spawner**.

Kod ponovnog pokretanja igre bit će potrebno ponovno pokrenuti *spawner*. Ovdje imamo događaj koji ćemo ponovno pozvati koji se zove **Reset Spawner**.



Slika 111: Postavljanje brojača funkcije Start Spawner



Slika 112: Ponovno pokretanje spawnera

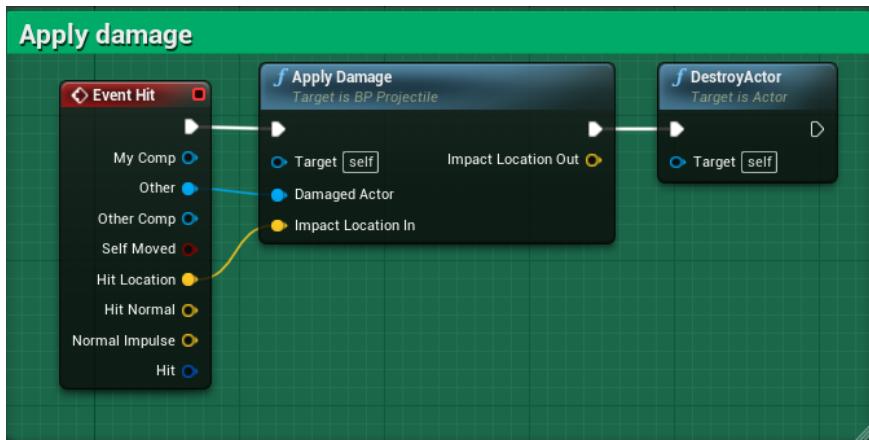
7.3 SUSTAV ZA ŠTETU

7.3.1 FUNKCIJE APPLY DAMAGE

Pomoću klase `BP_Projectile` se prenosi i podatak o količini štete koju će neprijatelj poprimiti. Naime, u klasi `BP_Projectile` je implementiran događaj `Event Hit` koji se poziva svaki put kada se projektil sudari s nečim. Događaj `Event Hit` pri okidanju poziva funkciju `Apply Damage` koja prima argument `Damaged Actor`, odnosno `actor` kojeg je projektil pogodio.

Prva stvar pri pozivu funkcije `Apply Damage` jest provjeravanje argumenta `Damaged Actor` za oznakom `Enemy` koji označava da je pogođeni `actor` uistinu neprijatelj. To je bitno zato što je jedino moguće, a jedino je i poželjno ostetiti neprijatelje. Također provjerava se je li `actor` koji je pogoden uopće sposoban primiti štetu. Naime, samo `actori` kod kojih je implementiran događaj `Any Damage` mogu biti oštećeni, odnosno objekt `Damaged Actor` mora biti referenca na nekog `actora` koji ima implementiran događaj `Any Damage`.

Nakon što uspješno prođu obje provjere dobavlja se referenca na oznake pogodenog neprijatelja pomoću objekta `Damaged Actor` i referenca na oznake ispaljenog projektila. Nakon toga se uspoređuje prvi tag projektila s



Slika 113: Izbjegavanje projektila

drugim tagom neprijatelja. U slučaju da su tagovi jednaki projektil je jak protiv neprijatelja i oštetić će ga duplo više od normalnog. Ako oznake nisu jednake, uspoređuje se drugi tag projektila s drugom oznakom neprijatelja, a ako su te oznake jednake, neprijatelj će biti oštećen za duplo manje od normalnog. Inače će neprijatelj biti oštećen za normalnu količinu štete.

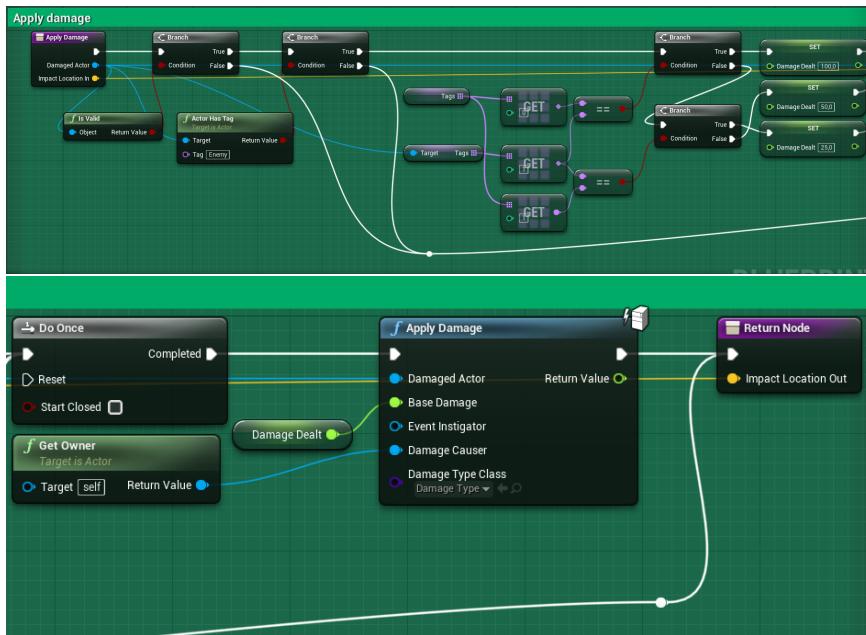
Količina štete postavlja se u varijablu **Damage Dealt** koja se postavlja na 100 ako je prva grupa oznaka jednaka, postavlja se na 25 ako je druga grupa oznaka jednaka, a na 50 se postavlja ako oznake uopće nisu jednake. Nakon postavljanja varijable **Damage Dealt** poziva se **Do Once** koji osigurava da će se sljedeći dio koda izvršiti samo jednom. Nakon toga se poziva funkcija **Apply Damage** od Unreal Enginea kojoj se proslijeđuje varijabla **Damage Dealt** u kojoj piše koliko će neprijatelj biti oštećen i proslijeđuje se objekt koji govori tko je vlasnik projektila. Tada je vlastita funkcija **Apply Damage** gotova.

Logika za primanje štete na protivniku nalazi se na slici 115 u klasi **BP Enemy**.

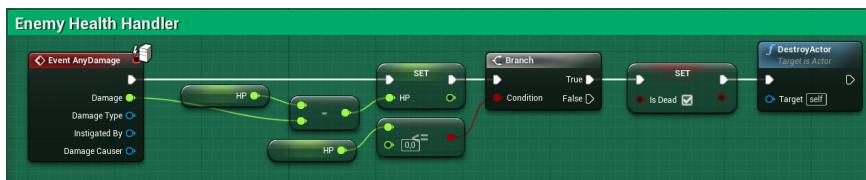
U klasi **MotionControllerPawn** dodajemo događaj **Event Any Damage** u kojem za sada samo ispisujemo "auch".

7.3.2 DOGADAJ ANYDAMAGE

Prethodni odlomak završen je objašnjnjem na koji način projektil šteti neprijatelju. Prilikom toga je spomenuto da se poziva funkcija od Unreal Enginea **Apply Damage** u kojoj se proslijeđuje količina štete, *actor* koji je oštetio i *actor* koji treba primiti štetu. U slučaju projektila *actor* koji prima štetu je neprijatelj. Kako bi se neprijatelj registrirao da je primio štetu potrebno je implementirati takav događaj. Iz tog razloga je u **BP_Enemy** implementiran događaj **Event AnyDamage** koji će se pozvati nakon što je u argument



Slika 114: Funkcija Apply Damage



Slika 115: Funkcija Apply Damage



Slika 116: Funkcija Apply Damage

za oštećenog *actora* u *Apply Damage* funkciji proslijeden neprijatelj.

Funkcija *Event AnyDamage* za argument ima varijablu tipa integer u koju se proslijedi količina štete koju će neprijatelj primiti. Ta se varijabla zove **Damage**. Za rad funkcije bitno je spomenuti i varijablu **HP** tipa integer u kojoj je pohranjen broj koji se odnosi na količinu štete koju neprijatelj može primiti prije nego što je uništen. Pretpostavljena vrijednost integera **HP** je 100 što znači da neprijatelj može primiti 100 štete kada se stvori. *Event AnyDamage* prvo od integera **HP** oduzme integer **Damage** i postavi

HP na novu, manju vrijednost. Na taj način je ažurirana količina štete koju neprijatelj može primiti nakon pogotka projektila. Nakon što se HP smanjio postoji mogućnost da je postao manji od 1 što znači da je igrač uništio neprijatelja. Iz tog je razloga nakon postavljanja varijable HP na novu vrijednost potrebno provjeriti koliko je li funkcija HP manja ili jednaka broju 0. U slučaju da nije, ništa se ne događa. U slučaju da je, neprijatelj je poražen. U slučaju da je neprijatelj poražen, potrebno je pozvati funkciju `DestroyActor` koja će ukloniti neprijatelja.

7.3.3 VRSTE NEPRIJATELJA

Na početku je rečeno da će neki neprijatelji primati povećanu štetu od određenih projektila, a od nekih manju štetu. Kako bi se to moglo implementirati potrebno je imati veći broj neprijatelja. Najbolji način za to napraviti je stvaranjem djece *blueprinta* `BP_Wizard`. Nakon stvaranja djece potrebno ih je premjestiti u prikladnu verziju te im dati prikladna imena (u našem slučaju `BP_Wizard Devil`, `BP_Wizard Possessed` i `BP_Wizard Skeleton`). Na svako od djece je potrebno postaviti oznaku na kojima će pisati protiv koje vrste projektila su jaki ili slabi. Nakon toga je i na svaki od projektila potrebno postaviti dvije oznake. U prvu oznaku bit će zapisano protiv kojeg je neprijatelja projektil slab, a u drugi protiv kojeg je jak.

Na igrača je također potrebno postaviti tag kako bi se znalo da je on taj koji je oštetio neprijatelja.

7.4 ZAKLJUČAK

Po završetku ove lekcije neprijatelji bi trebali biti uspješno stvoreni, trebali bi ganjati igrača te se ponašati kao stvarni protivnici - gađati, vidjeti i umirati.

Kao dodatan izazov programeri mogu stvoriti i više vrsta neprijatelja s raznim moćima - to bi trebalo biti moguće kombiniranjem svega znanja koje je do sada predstavljeno.

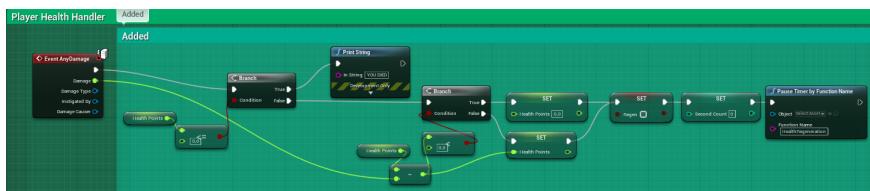
8 ZDRAVLJE IGRAČA

Neprijatelj može oštetiti igrača na sličan način kao što igrač može neprijatelja. To je važan element većine igara jer predstavlja izazov za igrača i povećava zadovoljstvo igrača kad se neka prepreka prijeđe. Ovo poglavlje bavi se zdravljem igrača, odnosno sustavom koji služi za ranjavanje igrača u igri.

Nakon što neprijateljski projektil pogodi nešto poziva se događaj **Apply Damage**. Događaj **Apply Damage** prima 2 argumenta. Prvi je tipa **Object** i zove se **Damaged Actor**, u njemu je pohranjena referenca na pogodenog *actora*. Drugi argument je vektor naziva **Impact Location In** u kojem je pohranjena lokacija sudara projektila s drugim *actorom* i bitan je za VFX.

Na početku događaja **Apply Damage** provjerava se je li **Damaged Actor** validan objekt, a ako nije, funkcija se prekida. Nakon toga se provjerava ima li taj objekt oznaku **Player**. Ako nema, funkcija se prekida. U slučaju da su oba uvjeta ispunjena, poziva se funkcija Unreal Enginea **Apply Damage** koja za argumente prima **Damaged Actor**, *actora* koji je uzrokovao štetu i količinu štete koja je uvijek jednaka 25.

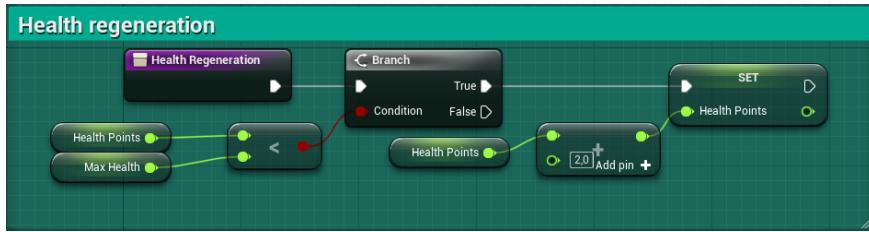
Poziv funkcije Unreal Engine **Apply Damage** s igračem kao **Damaged Actorom** uzrokuje okidanje događaja **Event AnyDamage** na igraču u kojeg se prosljeđuje količina štete od neprijateljskog projektila. Za taj je događaj važna varijabla **Health Points** u kojoj je napisano koliko se igraču mora nanositi štete prije nego bude poražen. Prva stvar koja se pri ulasku u događaj radi jest da se gleda je li varijabla **Health points** kojoj je oduzeta vrijednost varijable **Damage** manja od 0. Ako je, **Health Points** se postavlja na nulu, a inače se postavlja na dobivenu vrijednost nakon oduzimanja. Nakon toga se **Bool** varijabla **Regen** postavlja na **False** i integer varijabla **Second Count** se postavlja na 0. Tada se pauzira brojač za događaj **Health Regeneration**.



Slika 117: Funkcija **Apply Damage**

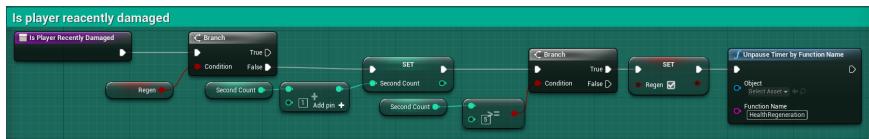
Događaj **Health Regeneration** služi za povećavanje **Health Points** (čija je početna vrijednost 100) varijable nakon što igraču neko vrijeme nije nanesena šteta. Povećanjem vrijednosti varijable **Health Points** omogućuje se da igraču može biti naneseno više štete pa je stoga bitno stavka kako igrač ne bi bila preteška. U prošlom odlomku je rečeno kako se pauzira događaj **Health Regeneration**. Tom istom događaju se dodjeljuje brojač

odmah prilikom početka igranja na Event **Begin Play**. Pritom se postavlja brojač koji ponavlja događaj **Health Regeneration** svakih 0,25 sekundi, a postavlja se i brojač za funkciju **IsPlayerRecentlyDamaged** na 1 sekunde. Funkcija **IsPlayerRecentlyDamaged** služi kako bi se znalo da je igrač primio štetu. Funkcija **IsPlayerRecentlyDamaged** prvo provjerava je li varijabla



Slika 118: Funkcija Apply Damage

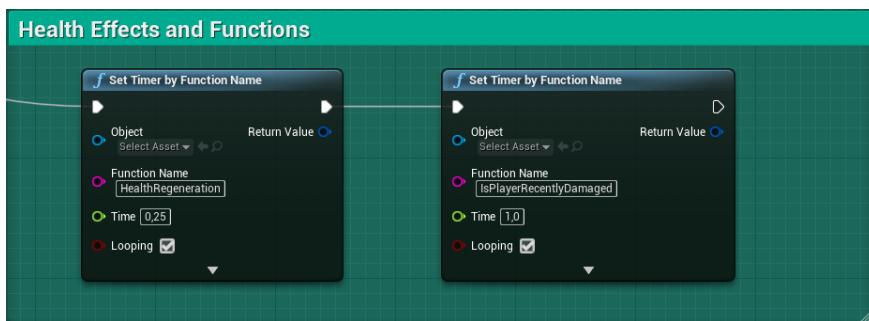
Regen istinita ili lažna. Ako je **False** (igrač je nedavno primio štetu) onda se varijabla **Second Count** (postavljena na nulu nakon primanja štete) inkrementira. Pošto se funkcija **IsPlayerRecentlyDamaged** pokreće svake sekunde, a **Second Count** inkrementira za 1 svaki put kad se pokrene, **Second Count** djeluje kao brojač. Nakon toga se provjerava je li **Second Count** veći ili jednak 5 te ako jest onda se varijabla **Regen** ponovo postavlja na **True**. Nakon toga se ponovno pokrene brojač za događaj **Health Regeneration** čime se omogućuje njegovo okidanje. Nakon što se uđe u događaj **Health**



Slika 119: Funkcija Apply Damage

Regeneration prvo se provjerava je li varijabla **Health Points** jednaka varijabli **Max Health**. Ako je jednaka više se ne može povećavati varijabla **Health Points**, pa se ne radi ništa, a inače se povećava za 2.

Brojači obiju funkcija pozivaju se na **BeginPlay**



Slika 120: Funkcija Apply Damage

9 IZBORNIK

U ovom će se poglavlju pokazati način implementacije izbornika u našoj igri. Izbornik je nezaobilazan dio svake igre. Prikazuje se prije pokretanja same igre te se u njemu može započeti nova igra, nastaviti stara igra, mogu postaviti postavke igre i sl.

U **Show/Destroy Menu** događa se upravo ono što i možemo zaključiti iz naziva, dakle prikazivanje i sakrivanje izbornika. Na početku se postavlja varijabla **MenuOpen**, koja je tipa **Boolean**. Kad je navedena varijabla postavljena na vrijednost **True** (dakle, izbornik je otvoren), tada se zabranjuju neke druge funkcionalnosti poput kretanja ili teleportiranja, a dolazi do stvaranja novog *actora* kojeg smo nazvali **BP Menu**. Navedeni actor u sebi sadrži komponentu koja se zove **widget** pa smo ga u ovom slučaju nazvali **Widget Menu**. Navedena komponenta uzima **Widget Class** koja se u našem primjeru zove **WG_Menu**.

Nakon ovo dijela postavlja se pitanje što je **widget**. **Widget** komponente unutar UE4 okoline dozvoljavaju stvaranje trodimenzionalnih elemnata korisničkog sučelja i njihovu manipulaciju. S obzirom na to da u našem primjeru radimo izbornik, koji je dio korisničkog sučelja, koristit ćemo upravo navedene komponente. Ovo radimo na ovaj način jer želimo li postaviti **widget** u okolinu tj. svijet, on mora biti pridružen na odgovarajućeg *actora*, barem dok govorimo o izradi VR igre. U nekim drugim igrama, **widget** se može pozvati preko funkcije **CreateWidget**. U ovom dijelu odredili smo još neke postavke poput **Draw Size** opcije ili **Collision Presets** dijela koji je postavljen na **Custom** zbog određenih preinaka (više o ovom dijelu pričali smo u prethodnim poglavljima). Također, u odjeljku **Rendering**, pod opcijom **Blend Mode** izabrali smo **Transparent** za postavljanje prozirnosti.

U prethodnom odjeljku smo spomenuli da kreirani *actor* u sebi sadrži **widget** klasu, koju smo nazvali **WG_Menu**. U ovom dijelu sadržano je grafičko sučelje koje se prikazuje korisniku jednom kad se izbornik učita. Objasnimo malo detaljnije strukturu sučelja. Na početku se kreira tzv. **Canvas Panel** koja je svojevrsna dizajnerska ploča koja omogućava postavljanje **widgeta** te dizajniranje samog izgleda sučelja. Unutar navedenog se nalaze **checkboxovi** koji su zamaskirani u tipke (**Training mode** i **Movement mode**), ispod **checkboxova** nalazi se tekst, dok se na vrhu sučelja nalaze dvije tipke, **Restart** i **Quit**. Komponente kanvasa postavljamo na variable (kako bismo se lakše snalazili u daljnjoj implementaciji). Također je potrebno napomenuti da je svakom od ovih elemenata pridružen tzv. **anchor** s određenim **anchor pointsima** (sidrene točke). Ove točke su nam važne jer se od njih oduzimaju odredene mjere (poput pozicije ili veličine elementa). Npr., za tipku **Restart** možemo reći da je ona od sidrene točke (koja je postavljena po sre-

dini kanvasa) pozicionirana za -400 po X osi te za 80 po Y osi. Važno je napomenuti da postoji više takvih sidrenih točaka te s istim vrijednostima na različitim sidrima element se ne bi nalazio na istoj poziciji.

Nakon što je opisano kako će grafički izgledati izbornik, u ovom će dijelu biti opisano kako ostvariti logiku iz pojedinih dijelova grafičkog sučelja. Prvo ćemo se pozabaviti opcijom ponovnog pokretanja igre. Nakon što igrač pritisne na tipku *Restart* pokreće se *custom event RestartFromMenu*. Ovaj događaj je nazvan *custom* što znači da smo mi implementirali njegovu logiku, makar postoji i drugačiji način. U tom načinu dolazi do *castanja* u *Game Mode* kojeg dohvaćamo pomoću funkcije *GetGameMode*. *Game Mode* predstavlja skup informacija i pravila o tome kako se igra mora odvijati pa je logično da su u ovim dijelu spremljene i informacije o ponovnom pokretanju igre. S obzirom na to da ovakav način ne radi kako smo zamislili da bi ponovno pokretanje igre trebalo raditi, odlučili smo kreirati vlastiti događaj za ovu funkcionalnost. Kad igrač pritisne na tipku *Quit* pokreće se funkcija *QuitGame* pomoću koje se izlazi iz igre.

9.1 FLIPFLOP I LOKACIJA IZBORNIKA

U ovom dijelu govorit ćemo o strukturi vrlo zanimljiva naziva. Riječ je o modifikaciji *switch* selekcije koja se zove *FlipFlop*. Način rada ove strukture je sljedeći: prvi put kad se prođe kroz nju, izvršava se izlaz A, tj. proces se nastavlja odvijati prema strani na koju pokazuje izlaz A. Kada se ova struktura pozove drugi put, tada se izvršava B izlaz, pozovemo li je treći put, izvršavat će se A izlaz, itd. U našem primjeru, izlaz iz A uvjeta vodi prema stvaranju novog *actora* (opisano u prethodnom poglavljiju). Novi *actor* odnosno izbornik će se stvoriti na određenoj lokaciji pa s toga dohvaćamo lokaciju (Z os) i rotaciju kamere, pritom računajući visinu kamere i određeni pomak kako bismo dobili visinu, tj. odgovarajuću lokaciju na kojoj ćemo prikazati izbornik.

Nakon što smo odredili visinu i lokaciju na kojoj će se prikazati izbornik, prelazimo na novu funkciju *SetTickableWhenPaused*. Jednostavno rečeno, ova funkcija čini da *actor* bude *tickable* (u nedostatku boljeg prijevoda: priljiva, klikljiva, označiva) kad je igra pauzirana. Kada je igra pauzirana, aktivirana je funkcija *SetGamePause*, tj. kada je vrijednost navedene funkcije postavljena na *True*, igra je pauzirana. Međutim, kad izvršimo implementaciju na ovakav način, doći će do pogreške prilikom iscrtavanja izbornika. Da izbjegnemo navedeni *bug*, funkcijom *SetGlobalTimeDilation* postavljamo vrijednost *Time Dilation* na nulu, što je jednako tome da se ništa ne događa unutar igre (a to je točno jer je igra pauzirana). Važno je napomenuti da je igra u ovom trenutku i dalje pokrenuta, samo je njen odvijanje usporeno i postavljeno na nulu. Nastavljajući dalje dolazimo do *SpawnActor*

BP Scoreboard dijela koji stvara tablicu rezultata (eng. *scoreboard*) na isti način na koji se kreira i izbornik, samo s lijeve strane izbornika.

Sve ovo se događa na grani koja izlazi iz A dijela FlipFlop strukture. Kada izlazimo iz B izlaza, funkcijom `SetGlobalTimeDilation` postavljamo vrijednost `Time Dilation` na 1, što znači da smo izašli iz pauze i da se igra nastavlja odvijati normalnom brzinom. Varijablu `MenuOpen` postavljamo na `False` vrijednost, što znači da smo zatvorili izbornik, dozvolili smo našem igraču da se opet kreće, miče, teleportira, itd. te nakon toga uništava *actore* koji su se stvorili prilikom pauziranja igre (izbornik i tablica rezultata). Navedeni *actori* su prethodno spremjeni, tj. postavljeni u varijable `MenuActor` i `ScoreActor` pa se iz tih varijabli dohvaćaju vrijednosti i uništavaju se funkcijom `DestroyActor`. Reference navedenih actora se postavljaju na ništa.

9.2 INTERAKCIJA S IZBORNIKOM

Dosad je bilo govora samo o implementaciji izbornika. U ovom dijelu bit će objašnjena interakcija igrača s izbornikom. Prva implementacija interakcije s izbornikom temeljila se na tzv. `WidgetInteraction` komponenti. Navedena komponenta na rukama igrača stvara vektore koji se nakon toga sudađaju s izbornikom. Komponenta `WidgetInteraction` započinje trima funkcijama: `GetWorldLocation` (za dohvaćanje lokacije), `GetForwardVector` (za dohvaćanje vektora) te `GetLastHitResult` (zadnja lokacija u kojoj je došlo do preklapanja `WidgetInteraction`a s korisničkim sučeljem). Na temelju svih ovih podataka izračunavamo gdje bi trebali postaviti pokazivač (eng. *pointer*). Prilikom odabira određene opcije, na izborniku se vidi plavi kružić koji nam pomaže u usmjeravanju prema željenoj opciji. Usmjeravanje kružića vršimo rukom te kada dođemo kružićem do opcije, pritiskom tipke kontrolera je izvršavamo. Ukratko, `WidgetInteraction` komponentu "lijepimo" na vektor i tada pomoću tog vektora vršimo željene radnje.

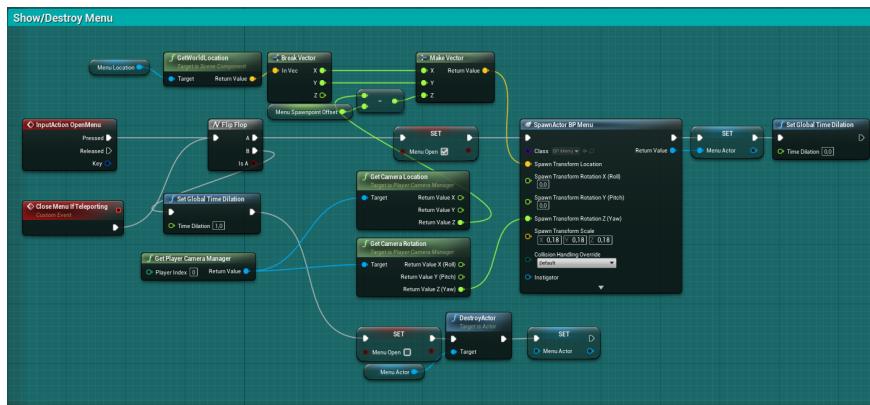
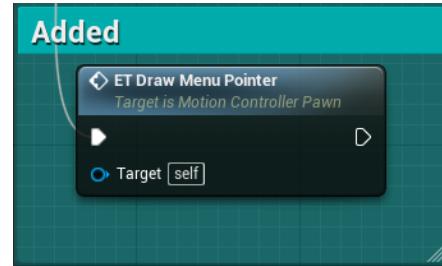
Nadalje postoje i određeni uvjeti za vidljivost pokazivača. Prvi je uvjet da je izbornik otvoren, dok je drugi da je udaljenost ruke (kontrolera) od izbornika najmanje 20. Razlog drugog uvjeta je taj što je implementacija izbornika izvršena na način da kad se rukom dođe blizu sfere izbornika, izbornik nestaje. Ako su oba uvjeta zadovoljena, tada će pokazivač na izborniku biti vidljiv. Ukoliko uvjeti nisu zadovoljeni, pokazivač će biti nevidljiv. Vidljivost pokazivača se postavlja pomoću funkcije `SetVisibility` gdje varijabla `NewVisibility` može poprimiti vrijednost `True` (vidljivost postavljena) ili `False` (za postavljanje nevidljivosti).

Nakon što smo objasnili kako igrač bira opcije izbornika, pogledajmo kako radi sama interakcija. Dakle, kada igrač prvi put otvoriti izbornik, opcije izbornika može birati samo putem desne ruke. Ukoliko se pokrene `InputAction FireLeft` ili `FireRight`, ispituje se je li izbornik već otvo-

ren. Ako izbornik nije otvoren, onda navedeni okidači neće ništa raditi, tj. pritisak tipke neće imati nikakvog efekta. S druge strane, ako je izbornik otvoren, a pritisnuli smo tipku na lijevom kontroler, provjeravamo je li varijabla `MenuController` postavljena na vrijednost nulu. U navedenu varijablu se pohranjuju integer vrijednosti te vrijednost nula predstavlja desni kontroler, dok vrijednost jedan predstavlja lijevi.

Dakle, ukoliko je vrijednost postavljena na desni kontroler, a stisnuli smo lijevi, onda ćemo postaviti kontroler s kojim upravljamo na lijevi zapisujući vrijednost jedan u varijablu `MenuController`. S druge strane, ako smo pritisnuli lijevi kontroler, a vrijednost u varijabli je već postavljena na lijevi kontroler, onda se pokreće funkcija `PressPointerKey`, zapravo se dozvoljava klik. Ovdje još nailazimo na funkciju `ReleasePointerKey` koja se pokreće kada igrač otpusti tipku na odgovarajućem kontroleru. Ista logika vrijedi i za provjeru desnog kontrolera. Prethodno opisani postupak nam omogućava promjenu ruke s kojom odabiremo opcije izbornika. Kao što smo napomenuli, pretpostavljena je ruka desna (prvi put kad se izbornik otvoriti upravljamo desnom rukom). Igrač treba pritisnuti tipku lijevog kontrolera želi li promijeniti ruku i tada će se upravljanje prebaciti na lijevu ruku.

SLIKE



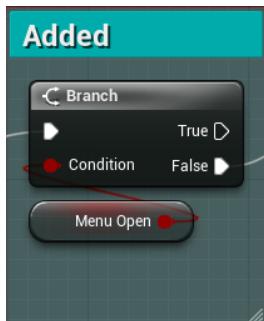
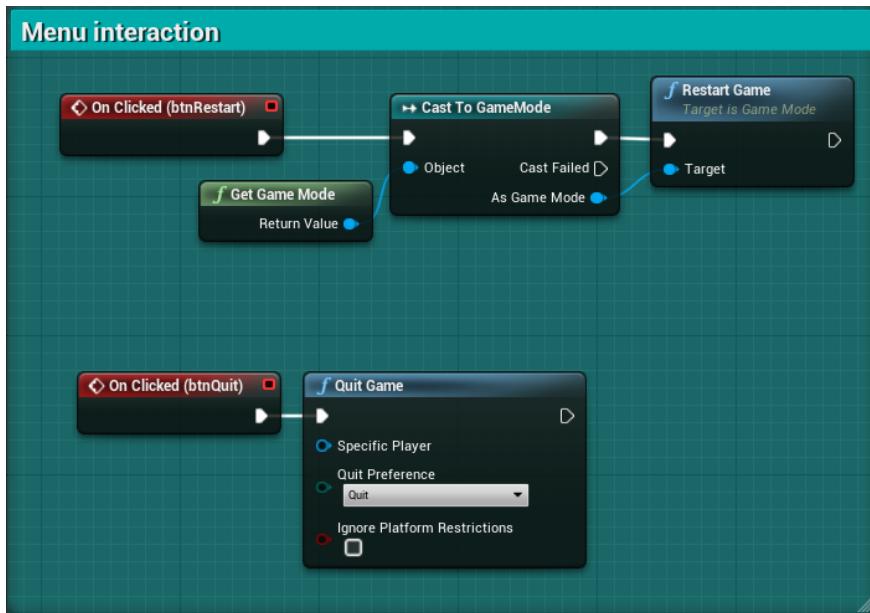
Samo mijenjanje ruku odvija se unutar `Interaction Hand Switching` odjeljka. Dakle, uzima se vrijednost iz varijable `MenuController` te ako je vrijednost postavljena na nulu, pomoću `GetWorldTransform` dohvaća se informacije o desnom kontroleru te pomoću funkcije `SetWorldLocationAndRotation` postavlja transform za `Widget Interaction`. Isti postupak vrijedi i za lijevi kontroler (ako je prosljadena vrijednost jedan), samo što će se uzimati informacije za lijevi kontroler.



10 SCOREBOARD API

U ovom poglavlju objasnit će se koncept koji se zove API (eng. *application programming interface*). Svi se mi spajamo na internetsku mrežu pomoću određenog uređaja (mobilni, računalo, laptop...). Kod spajanja šalju se određeni zahtjevi i čekujemo određene odgovore. Dakle, mi kao korisnici šaljemo zahtjeve prema određenom poslužitelju putem API-ja, poslužitelj ih obrađuje te nam API vraća podatke u obliku odgovora.

U nastavku projekta ništa ne ovisi o API funkcionalnosti pa ukoliko čitatelj ovog priručnika smatra da nema dovoljnog predznanja iz dijela web tehnologija, slobodno može preskočiti ovo poglavlje i nastaviti s razvojem igre.



Nakon uvodnih pojašnjenja, prebacimo se na naš projekt i zamislimo sljedeću situaciju. Igrač je izgubio život, dakle njegovi *health pointi* pali su ispod nule (na nulu) pa možemo reći da je igrač izgubio život. U većini igara, pa tako i u ovoj, nakon što igrač otvori izbornik, prikazuje se rezultat koji je ostvario tijekom igranja. Ovdje u igru dolazi API. Preko funkcije `GetPlatformUserName` dohvata se sustavno ime korisnika (ime Windows korisnika ili ime računala). Ovo smo izveli na taj način kako ne bi dodatno komplificirali proces uvođenjem virtualne tipkovnice za unos korisničkog imena nego jednostavno povučemo korisničko ime sustava. Nakon toga, u varijablu `Url`, koja je polje stringova koji se spajaju, dodaje se korisničko ime (dohvaćeno funkcijom `GetPlatformUserName`) tako da se pomoću funkcije `SetArrayElem` korisničko ime zapiše u polje pod indeks 2. Na isti način, u polje pod indeksom 4 dodajemo postignuti rezultat (eng. *score*) igrača dohvaćen iz privremene varijable `Score` (kad neprijatelj umre, poziva se funkcija `UpdateScore` koja inkrementira i printa string). Primjer zapisa u varijabli `Url` je

```
["http://localhost/data.php?username=", "User", "&score=", "0"]
```

Pomoću funkcije `JoinStringArray` spajamo sve komponente polja, pri čemu nismo postavljali nikakav dodatni separator nakon čega dobivamo jedan cijeli string.

Kako bismo uopće mogli koristiti funkcionalnosti za spajanje na Internet poput pristupa web servisima i slično, potrebno je instalirati jedan dodatak (eng. *plug-in*). Dodatak se zove `VaRest`, besplatan je te ga se može preuzeti s Epic Store platforme (*Marketplace* > u tražilicu napisati `VaRest` > odabratи ponuđenu opciju > klik na `Install to Engine`) > odabratи verziju enginea na koji se želi instalirati (u ovom se projektu koristi verzija enginea 4.25.4). Nakon što instalirate `VaRest` i vratite se u Unreal Engine okolinu, vidjet ćete *popup* prozor u kojem će pisati da je dostupan novi dodatak. Nakon toga pojavit će se dvije opcije: `Configure` i `Dismiss`; te kada se odabere `Configure`, otvorit će se prozor s dodatcima te omogućujemo željeni dodatak (u ovom slučaju `VaRest`) pritiskom na potvrđni okvir (eng. *checkbox*). Posljednja stvar koju je potrebno napraviti je ponovno pokrenuti UE4 nakon omogućavanja dodatka.

Vratimo se u API `Send` odjeljak. Desnim klikom miša u tražilicu upiše se pojam `GetVaRestSubsytem` i postavi se u okolinu. Iz navedene komponente će se izvlačiti određeni REST objekti poput JSON-a. Funkcija `ConstructJsonObject` služi upravo za ono što joj naziv i govori – kreira se novi JSON objekt. Pomoću funkcije `Call URL` pozivamo URL adresu koja je stvorena prilikom kreiranja stringa. Kod ove funkcije treba imati na umu da je potrebno staviti određeni *callback* jer će se funkcija prilikom izvođenja buniti da ne postoji nikakav *event*. Međutim, u našem primjeru nije potreban nikakav event pošto se funkcija samo pokušava spojiti na proslijedeni URL i spremiti vrijednost na API pa zato je i u primjeru taj događaj nazvan *nothing*.

Sada se pozabavimo prikazom rezultata igre. Stvaranje *actora* nazvanog `BP_Scoreboard` gotovo je isti kao i stvaranje izbornika. Nakon stvaranja *blueprinta* dodaje se nova `widget` komponenta pod imenom `Scoreboard` kojoj postavimo visinu i širinu (opcija `Draw Size` > upišemo vrijednosti *X* i *Y*), nakon toga podesimo `Collision Responses` parametre (na `Ignore` postavimo `TriggerArea`, `Projectile` i `EnemyProjectile`, na `Overlap` postavimo opcije `Camera`, `WorldStatic`, `WorldDynamic`, `Pawn`, `Vehicle`, `Destructible`, dok na `Block` postavimo opciju `Visibility`). Pod `Blend Mode` smo odabrali opciju `Transparent`.

Koristimo dva `widget actora` – jedan koristimo kao `widget` komponentu u drugom. Jedan `widget actor` je glavni gdje se prikazuje *scoreboard*, a drugi je jedna linija tog *scoreboarda* i onda se za svaki par korisnika i rezultata stvara jedna *score* linija u `WG_ScoreLine` koja se automatski dodaje u `WG_ScoreBoard` jer je to njegova komponenta (pri čemu i dalje imamo samo

jednu komponentu `WG_ScoreLine`). Sada ćemo objasniti kako izgledaju navedeni *widget actori*, pri čemu ćemo prvo objašnjavati *widget WG_ScoreLine*. Prvo, dodali smo `Horizontal Box` element, za razliku od izbornika kojem je početni element bio `Canvas Panel`. Unutar navedenog `Horizontal Box` elementa dodaju se tri nova `Horizontal Box` elementa koja sadrže `LineIndex` element (za ispis rednih brojeva uz imena korisnika, počevši od jedan prema dalje, prepostavljeni na 1), `LineUsername` (koji sadrži ime korisnika, prepostavljeni na „`username`“) i `LineScore` (osvojeni rezultat navedenog korisnika, prepostavljeni na „`000000`“). Navedeni elementi poslužit će kao *placeholderi* te im je moguće postaviti vertikalno ili horizontalno poravnanje, postaviti veličinu (opcija `Size`), dodati tekst (opcija `Text`). Uz navedene elemente postavljene su privremene vrijednosti unutar tih `textboxova` koje nisu pretjerano važne jer će se one mijenjati sukladno dobivenom odgovoru od servera. Što se tiče `Event Grapha` prethodno objašnjjenog widgeta, potrebno je postaviti opciju `Expose on Spawn` na vrijednost `True` (označiti `checkbox` kraj opcije) i označiti `visible` varijable s lijeve strane (redom pisane: `LineIndex`, `LineScore`, `LineUsername`, `index`, `Username`, `Score`). Nakon toga, postoji novi widget, `WG_Scoreboard`. Uz određene elemente putem naziva (`Scoreboard`) u element `ScoreList` pridružit ćemo `WG_ScoreLine`, tako da se izvrši funkcionalnost ispisa rezultata.

Prije nego što nam se prikažu rezultati igre, potrebno je izvršiti cijeli niz postupaka. Prilikom kreiranja `WG_Scoreboarda`, tj. njegova konstruiranja stvara se JSON objekt preko `VaRest` podsustava, pomoću funkcije `CallURL` poziva se određeni URL te se vrši obrada `RestCallback eventa` (odgovora). Pomoću funkcije `GetresponseObject` uzimamo zahtjev. Iz rezultata navedene funkcije se pomoću `GetArrayField` funkcije izvlači polje naziva `scores` te svaki vraćeni element polja gledamo kao objekt (funkcija `AsObject`) i onda iz tog objekta uzimamo korisničko ime i rezultat korisnika. U funkciji `CreateWGScoreLineWidget` stvara se novi *score line* (imajući na umu inkrementaciju indeksa, stvara se redni broj, korisničko ime i rezultat koji će se postaviti u `WG_ScoreLine` *widget*) te ga se pomoću funkcije `AddChild` pohranjuje u `ScoreList` varijablu. Slijedi primjer ispisa koji se dobije kao odgovor od servera.

```
{
  "scores": [
    {
      "username": "Huzz",
      "score": 255
    },
    {
      "username": "Haso",
      "score": 55
    }
  ]
}
```

```

    },
    {
        "username": "Pero",
        "score": 11
    },
    ...
]
}

```

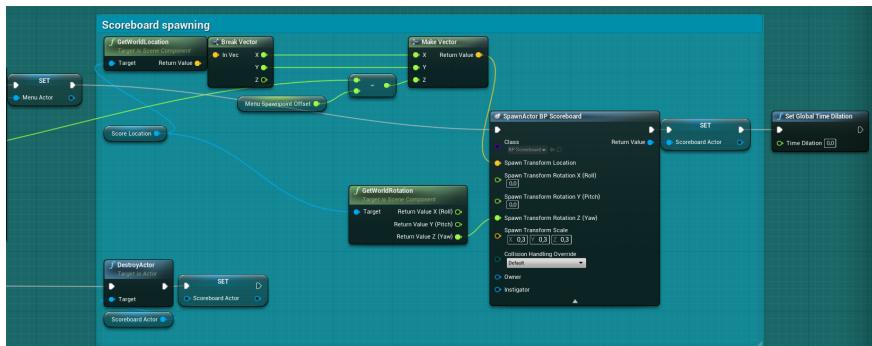
Iz primjera se vidi da server vraća podatke u JSON formatu. U JSON-u su podatci oblikovani prema principu ”ključ”: ”vrijednost” pri čemu ključ mora biti string tip, a vrijednost može biti string, boolean, broj, polje ili null vrijednost. U ovom primjeru server nam vraća podatke o uspjehu pojedinih igrača. Prvi par je oblika `"username": "korisnicko_ime"`, dok je drugi par oblika `"score": "rezultat_korisnika"`.

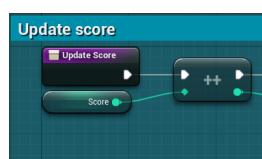
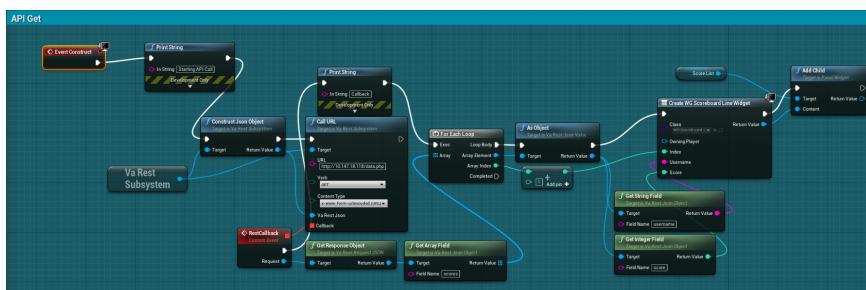
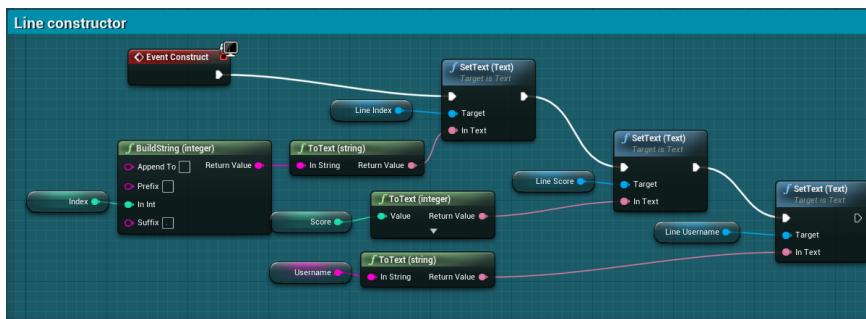
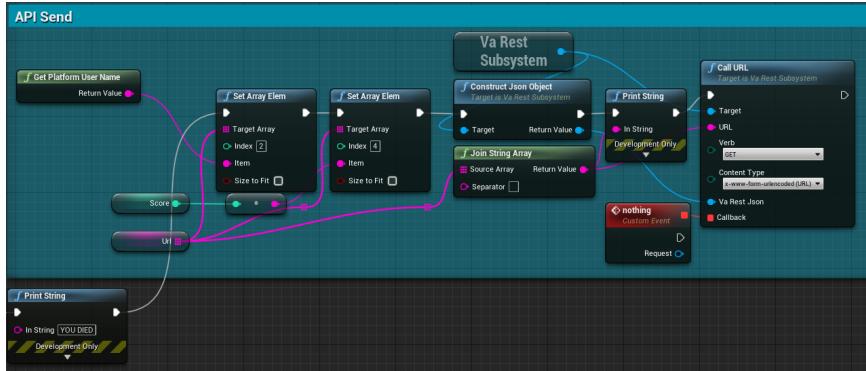
Tablica rezultata (prikazano u odjeljku *Scoreboard spawning*) stvara se na sličan način kao i izbornik, samo što se koristi ScoreLocation varijabla u kojoj je zapisana lokacija tablice rezultata. Po želji, ScoreLocationArrow komponenta postavlja se u *viewportu*.

Za kraj ovog poglavlja potrebno je dati par napomena. Prvo, potrebno je imati instaliran web poslužitelj na računalu. Preporučujemo instalaciju XAMPP ili WAMP paketa. U navedenim paketima dostupni su Apache web poslužitelj, PHP programski jezik i određeni sustav za upravljanje bazom podataka (najčešće je to MySQL ili MariaDB, ovisno o izabranom paketu, a također se nudi i phpMyAdmin alat pomoću kojeg se upravlja MySQL ili MariaDB sustavima).

Drugo, potreban nam je programski jezik koji će vršiti određene radnje na poslužitelju. Najčešće je to PHP (i mi smo ga koristili u našem projektu), a pošto on već dolazi u XAMPP ili WAMP paketu, logično je da ćemo uzeti upravo taj jezik.

SLIKE





11 SPREMANJE POSTAVKI

U ovom poglavlju učit će se o spremanju korisničkih preferenci i načinu na koji se iste ostvaruju. U jednom od prethodnih poglavlja detaljno smo raspravljali o vizualnom izgledu izbornika, unutarnjoj logici i sl. Raspravljat ćemo o potvrđnim okvirima (eng. *checkbox*) koji su sastavni dio izbornika pri čemu će se navedeni potvrđni okviri zamaskirati u tipke (opcije *Training mode* i *Movement mode*).

Zašto *checkboxove* maskiramo u tipke? Skaliranjem veličine *checkbox*, tj. potvrđni okvir postaje pikseliziran i ne skalira se vektorski pa smo mu promijenili stil, gdje smo u odjeljku *Style* pod istoimenom opcijom promijenili *Check Box Type* u *Toggle Button*. Potrebno je i uključiti opciju *isVariable* tako da se u grafu može raditi s navedenim tipkama. Pritisom na neku od navedenih tipki, doći će do određene promjene.

Dolaskom do promjene stanja u checkboxu, tj. promjene njene *Bool variable*, pokrenut će se event *OnCheckStateChanged*. Prije nego što krenemo detaljnije objašnjavati navedeni događaj, objasnit ćemo ukratko što se događa kada se pritisne gumb *Quit*. Osim što se ovim gumbom izlazi iz igre, iz klase *MotionControllerPawn* (glavna klasa ovog projekta) se ovim gumbom također poziva vlastiti događaj *SaveUserPreferences*. U navedenom se pokreće funkcija *LoadGameFromSlot* koja je opet sama po sebi razumljiva (učitava igru iz utora (eng. *slot*)). Naziv utora je *UserPreferencesData* i njega kreiramo, a ako ne postoji, putem funkcije *CreateSaveGameObject* gdje pod opcijom *Save Game Class* odabiremo *SG_UserPreferences*. Navedeni actor (tipa je *SaveGame*), tj. novo kreirani objekt nije ništa drugo nego spremnik, kontejner koji u sebi sadrži dvije varijable, *TrainingModePreference* i *MovementTypePreference*. Najčešće moramo nadjačati (eng. *override*) ono što je prije zapisano u njemu, ukoliko nismo imali potrebe kreirati navedeni utor.

Dakle, postojeći utor *castamo* u *SG_UserPreferences*, pod uvjetom da nismo imali potrebe kreirati novi objekt (neuspjeli *casta* znači da ne postoji utor naziva *UserPreferencesData* pa se pokreće funkcija *CreateSaveGame*) i postavljamo *RuntimeDataPreferences* (referenca na *SG_UserPreferences*). Postavljanje se vrši u oba slučaja, i ukoliko *cast* prođe, i ukoliko ne prođe. Bitno je još napomenuti da je *RuntimeDataPreferences* referenca na objekt tipa *SG_UserPreferences* koji u sebi sadrži dvije varijable:

- *TrainingModePreference*
- *MovementTypePreference*

Objasnimo malo jednostavnije prethodno opisani postupak. Postoje dva stanja:

- Training Mode
- Movement Type Preference

Training Mode je mod u kojem su upaljene sfere koje su vidljive u prostoru što omogućava igraču da se lakše snalazi prilikom okidanja moći i sl. Movement Type Preference je opcija koja mijenja možemo li se kretati ili ne, ovisno o tome ima li igrač tzv. motion sickness.

Te dvije vrijednosti su pretpostavljene na neku vrijednost, ali korisnik nakon prvog korištenja bira kakve postavke hoće i onda se one pospremaju svaki put prilikom izlaska u odgovarajući utor.

Kada govorimo o učitavanju opcija, postupak je gotovo isti kao i kod spremanja, samo što se postavljaju variabile **Training** i **MovementType**. Navedene variabile koriste se za vrijeme dok je aplikacija pokrenuta i razlikuju se od onih opisanih u prethodnom dijelu tako da se kod spremanja iz varijabli **Training** i **MovementType** se preko **SaveGamea** spremaju vrijednosti u utor **UserPreferences**, dok se kod učitavanja vrijednosti iz utora **UserPreferences** preko **SG_SaveGamea** sprema u **Training** i **MovementType** varijable. Važno je napomenuti da se učitavanje opcija pokreće u trenutku pokretanja igre. S druge strane, spremanje opcija pokreće se prilikom pokretanja izlaska iz igre, tj. pritiskom na tipku *Quit*.

Na početku poglavlja bilo je govora o događaju **OnCheckStateChanged**. Kada se pokrene navedeni event (tj. kada u izborniku odaberemo tipku **Training mode** ili **Movement mode**), doći će do promjene vrijednosti varijabli (**MovementType** ili **Training**, ovisno o pritisnutoj tipki) i pokretanja novih evenata (**TurnStandingModeOn/Off** i **TurnTrainingModeOn/Off**). Ti događaji zapravo rade na vrlo sličan način. U **TurnTrainingModeOn/Off** provjeravamo je li varijabla **Training** postavljena na **False** vrijednost te ukoliko jest, postavimo je na **True** vrijednost i obratno. Isti princip vrijedi za **StandingMode** samo što se ovdje provjerava varijabla **MovementType**. Postavke teleportiranja, kretanja i rotiranja mijenjaju se u **MovementType** varijabli, a **Training** je pomoć pri učenju pokreta ili svladavanju koncepata igre. Za kraj ovog odjeljka, navedena su mjesta na kojima se pozivaju i postavljaju vrijednosti varijable **MovementType**: unutar odjeljka **Switch Standing Mode** (postavljanje i dohvaćanje vrijednosti varijable), u odjeljku **Player Rotation** (dohvaćanje vrijednosti varijable), **Tick Collision Capsule** (dohvaćanje vrijednosti varijable), **Handler Controller Input** (dohvaćanje vrijednosti varijable), **Save Game Preferences** (dohvaćanje vrijednosti varijable) te se varijabla postavlja prilikom učitavanja preferenci, dakle odjeljak **Load Game Preferences** (logika u ovom odjeljku se kreće izvoditi prilikom pokretanja događaja **BeginPlay**, dakle postavljaju i učitaju varijable **Training** i **MovementType**).

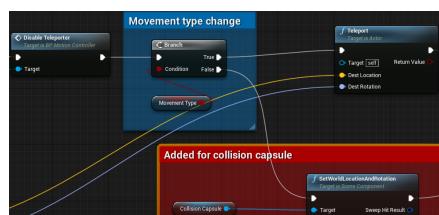
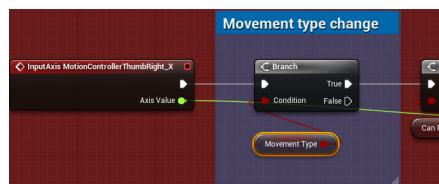
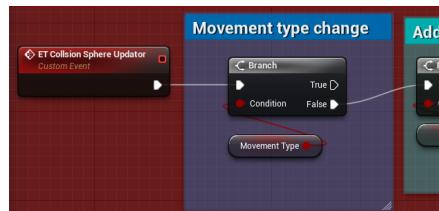
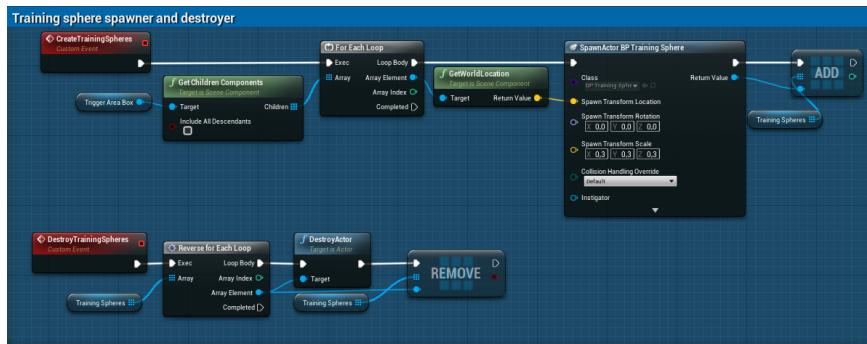
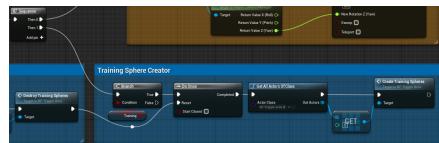
Nakon što smo opisali postupke biranja opcija, opisat ćemo i što se događa ako odlučimo odabrati trening. Prvenstveno, potrebno je dodati novi *blueprint* i materijal za trening sfere. Dakle, kreira se materijal naziva *M_TrainingSphere*. Koristimo običan vektor parametar pri čemu postavljamo *Base Color* i *Alpha* pridružimo u *Opacity*. Također, u odjeljku *Material*, pod opcijom *Blend Mode* izabrano je svojstvo *Translucent* tako da materijal bude proziran. Nakon toga, kreiramo *blueprint* *BP_TrainingSphere*. Odabrati opciju *Add Component* i iz popisa izabrati *Sphere*. S desne strane u postavkama,

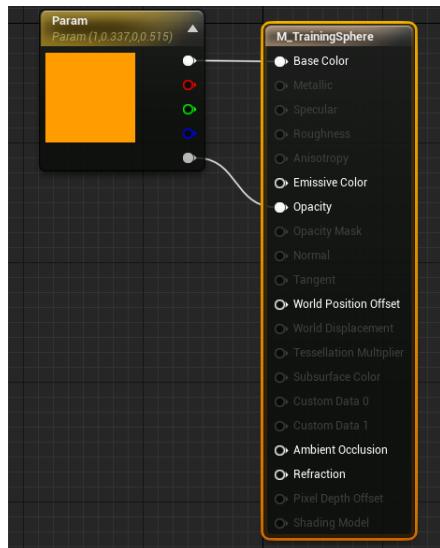
potrebno je postaviti **Collision Presets** na **Custom** i sve stavke postaviti na **Ignore**.

Također, potrebno je izabrati materijal **M.TrainingSphere** u odjeljku *Materials*. U odjeljku *Training Sphere Creator* prikazano je kreiranje sfera za trening. Dakle, ukoliko je varijabla **Training** postavljena, pokrenut će se **Custom Event CreateTrainingSpheres** (prilikom pritiska na tipku). Prvo ćemo iz **Trigger Area** Boxa funkcijom **GetChildrenComponents** dohvati svu „djecu“ (npr. **Trigger Area** Box sadrži **FrontLeftSphere**, **TopLeftSphere**, **TopRightSphere** komponente koje nazivamo djecom) i za svaku dohvaćeno dijete, tj. element kreiramo **Training Sphere** koji je zapravo vizualno reprezentacija onog što će se dogoditi kad stisnete tipku. Pritom se novokreirana sfera skalira i doda u polje, kako bi ih se moglo kasnije lakše izbrisati.

Kada se želi pozvati uništavanje, tj. brisanje sfere pokreće se event **DestroyTrainingSpheres** (događaj se pokreće prilikom otpuštanja tipke). Navedeni događaj se poziva u odjeljku **Training Sphere Destroyer** i u njemu se zapravo dohvaća svaki element **Training Sphere** polja te se pomoću funkcije **DestroyActor** uništava i briše njihovu referencu. Navedeni događaj će se pokrenuti kada igrač otpusti tipke na kontroleru jer je cilj da sfere budu vidljive samo kada kreiramo pokret, a ne cijelo vrijeme. Vrijedno je još napomenuti da će uništavanje sfera dovesti do reseta kreiranja sfera, tj. sfere će se moći ponovno kreirati pošto je samo kreiranje postavljeno unutar **Do Once** strukture. Dakle ono je moguće provesti samo jednom dok se funkcija ne resetira.







12 MAPA

Kako bi igra dobila dodatnu nijansu finoće, dodali smo i mapu na kojoj će se igra odvijati. Mape su vrlo važna komponenta svake igre jer imaju jedan dodatan čar koji nadopunjuje igru. Mapa, odnosno okolina u kojoj se odvija igra, ima raznih, od snježnih krajolika do pustinjskih oaza dok ćemo mi u našem projektu koristiti postojeću mapu koja izgleda poput gladijatorske arene. Mapu smo uzeli na način kao što smo uzeli i skin gotovog lika, dakle putem Epic Storea. Sada je potrebno kreirati Level1 (desni klik na content browser -; Create -; Level). Iz MotionControllerMap označimo NavMeshBoundsVolume i mesh poda koje kopiramo i stavljamo u našu novokreiranu mapu na način da zalijepimo kopirano u word outliner. Navmesh nam je potreban kako bismo odredili područje po kojem se AI može kretati. Bez navedenog, kretanje ne bi bilo moguće. Jedna napomena prilikom rada s NavMeshBoundsVolume. Znalo se desiti da je kretanje AI-a podbacilo, tj. AI likovi se nisu htjeli kretati. Zbog toga je potrebno pritisnuti tipku p da bi se prikazala vizualizacija navmesha, tj. bit će vidljiva zelena površina koja će prikazivati prostor po kojem se neprijatelji mogu kretati. Treba provjeriti obuhvaća li zelena površina sav onaj dio koji želimo da bude obuhvaćen, tj. sav onaj dio po kojem želimo da se AI kreće. Ukoliko zelena površina nije obuhvatila željeno područje, potrebno je malo podići NavMeshBoundsVolume i kombinacijom tipki ctrl + z vratiti na prijašnje mjesto. Ukoliko ste već prije radili s UE4 ili bilo kojim drugim alatom za razvoj igara, tada znate da je svjetlo u igri vrlo važno. Bez svjetla, u igri bi praktički prevladavao mrak te se ne bi moglo ništa vidjeti. Zbog toga, na mapu smo dodali tzv. DirectionalLight koji predstavlja određeno svjetlo u atmosferi, SkyAtmosphere kojim smo dodali oblake, atmosferu i sl., SkyDomeMesh koji predstavlja nebo te SkyLight koji reprezentira Sunce.

Raspored svjetla je napravljen tako da se vide i određene sjene pa je vizualni doživljaj podignut na još jedan nivo.

13 GOTOVI ELEMENTI I ANIMACIJE

Do ovog poglavlja pričali smo o više-manje o programiranju, unutarnjoj logici, kako određene stvari funkcioniraju i slično, dok ćemo se od sada bazirati na uljepšavanje igre, dakle dodavanje animacija, vizualnih i audio efekata itd. Iako mehanike rade, i igra je gotova završena, postavlja se pitanje bili netko igrao igru koja nema vizualno privlačnu pozadinu, glazbu, tj. zvučne efekte, u kojoj su likovi jednolični i slično. Zbog toga ćemo u narednih nekoliko poglavlja prikazati kako dodati animaciju, vizualne i audio efekte. Cijeli ovaj koncept oko uljepšavanja igre ćemo započeti pričom o dodavanju elemenata.

O elementima koji tvore igru unutar Unreal Enginea već je bilo riječi u uvodu ovog priručnika, međutim i ovdje ćemo napomenuti nekoliko stvari. Prva je da u Unreal Engineu postoje mogućnosti odabira gotovih elemenata (npr. na Marketplaceu pronađemo gotov prikaz kamena ili drva i sl.). S druge strane putem geometrijskih tijela (kocke, kugle, ...) moguće je kreirati jednostavnog neprijatelja ili metu za gađanje. Na primjer, za potrebe testiranja, neprijatelja smo oblikovali putem stošca i kugle i na njemu testirali funkcionalnosti pucanja.

Slična stvar vrijedi i za projektile. Kod projektila smo prilikom testiranja koristili raznobojne kuglice, po abilityju kojeg predstavljaju, dok smo navedene diferencijacije boja ostvarili tako što smo u RGBA parametar Boja postavili boju i pridružili je Base Color opciju materijala koji možemo mijenjati u postavkama projektila. Slična stvar vrijedi i za projektil neprijatelja koji uz navedeni parametar Boja ima još jednu varijablu koja se množi s parametrom Boja. Svi projektili imaju Emissive Color koji je isti kao i Base Color, jedino se između metaka razlikuje po svom intenzitetu. Također, materijalu se može dodati Opacity opcija tako da on bude proziran (implementirano u materijalu MT_TransparentSphere).

Na prethodno opisane načine kreiramo jednostavne materijale. Dakle, putem jednostavnih mesh objekata smo ispitivali i testirali razne funkcionalnosti. O meshu je već bilo riječi u uvodu ovog tutorijala, no možemo ovdje napomenuti da je mesh zapravo dio geometrije koji se sastoji od seta poligona, vrhova, bridova itd. koji kreiraju 3D objekt. Kao što možemo vidjeti iz ove jednostavne definicije, kocka ili kugla su zapravo mesh.

Kao što smo već napomenuli, u Unreal Engineu možemo importati gotove mesheve (npr., gotovi kamen, ili drvo, kuću i sl.). Kako bi pronašli navedene elemente, potrebno je posjetiti Epic Store platformu i otvoriti Marketplace. Svaki mjesec se nude besplatne stvari koje možete koristiti u svojim projektima (pod opcijom Free odabrati Free For The Month, pronaći stvari koje želite, dodajte ih u košarice i vaše su, možete ih koristiti kad kod želite).

S druge strane, ako odaberete opciju Free -i Epic Games Content pronaći će te mnoštvo besplatnog sadržaja kojeg možete ukomponirati u svoju igru. Npr. ukoliko u tražilicu pokraj Help opcije upišete pojam gideon (naš glavni wizard u igri), možete ga odabratи, klikom na Add To Project dodajete ga u željeni projekt. Bitno za naglasiti je da smo sve elemente s Epic Stora početno ubacivali u zasebne projekte kako bi mogli bolje shvatiti kako određeni element funkcionira. Npr. za već spomenutog wizarda kreirali smo novi projekt u kojem smo pregledavali njegove animacije i efekte. Bit kreiranja zasebnog projekta je izbjegavanje migriranja elemenata koje nećemo koristiti jer bi na taj način gomilali nepotrebne stvari u projektu. Dodavanjem u zasebni projekt možemo pregledati cijeli projekt i vidjeti koji točno dio (animaciju, mesh, vizualni efekt) želimo migrirati i onda taj dio dodamo u projekt.

Pošto smo već spomenuli wizarda, objasnit ćemo rad na njemu. U njemu smo gledali animacije lika dohvaćenog s Epic Stora. Riječ je o Gideonu, liku iz video igre Paragon, čiji se sadržaj koristi kod kreiranja igara unutar Unreal Enginea. Naime, Paragon je bila igra koju su razvili unutar Epic Gamesa, međutim nije opravdala očekivanja te su je otkazali, a sav sadržaj te igre postao je dostupan, putem Marketplacea, svakome tko radi u Unreal Engineu. Međutim, vratimo se na našu temu. Prva stvar koju smo dodali je jedan skin za Gideona (Characters -i Heroes -i Gideon -i Skins), koji je mesh. Ako želimo dodati mesh sa svim njegovim teksturama, materijalima i animacijama, kako bi lik u našoj igri izgledao isto kao onaj lik kojeg smo preuzezeli s Marketplacea, desnim klik na Gideon_Undertow, odabratи opciju Asset Actions -i Migrate.

Pokazat će se prozor sa stvarima povezanim s likom, stisnemo OK te biramo datoteku u koju će se podaci migrirati. Bitno je da se podaci migriraju u Content datoteku (inače će se pojavitи poruka pogreške). Odabirom Content datoteke i pritiskom na tipku Select Folder dolazi do migracije podataka u željeni projekt. Sad se može postaviti pitanje zašto migrirati podatke unutar Content datoteke. Odgovor je taj što struktura direktorija mora ostati ista, dakle ukoliko se neki mesh u samostalnoj verziji nalazi unutar direktorija Content -i , tada se i u drugim projektima mora nalaziti na istoj putanji. Da rezimiramo, prvo smo pronašli elemente koji nam trebaju, da bi onda na opisani način migrirali samo one stvari koje nam trebaju u naš projekt. Stvari koje su nam posebno bitne: svi skinovi sadržani unutar mape Skins, dok su iz mape Animations bitne animacije Death Back, Idle, Jog_Fwd i Primary_Attack_A_Medium_Montage i Primary_Attack_A_Medium.

Jednom kada smo navedene stvari uvezli u naš projekt, tada ih je bilo potrebno uređiti. Konkretno, otvorili smo našeg čarobnjaka, s lijeve strane odabrali komponentu Mesh (Inherited) i s desne strane, u odjeljku Mesh, odabrali njegov Mesh. Nakon toga, materijali u odjeljku Materials će se

sami postaviti, dok je naš zadatak bio da ručno, u odjeljku Animation, postavimo set njegovih animacija. Problem s Skeletal Meshevima i animacijama je taj što su oni povezani, dakle ne možete koristiti jedan Skeletal Mesh, a koristiti animacijsku klasu povezanu s drugim Skeletal Meshom.

Prethodno opisani postupak se odnosi na wizarda. Sada ćemo objasniti kako izgledaju „djeca“ wizarda. Ako se prisjetite, na početku smo rekli da postoji glavna klasa neprijatelja, nazvana wizard i da je u njemu implementirana sva logika za pucanje i slično. Mi ćemo imati još nekoliko neprijatelja, a svi oni nasljeđuju logiku koja je kreirana unutar wizarda pa tako ako pogledate njegov Event Graph vidjet će te da je prazan. Razlika je u drugačijem skinu, tj. meshu i potrebnim materijalima (koji se automatski povuku). Kreiranje navedenog neprijatelja (nazvanog BP_WizardDevil) se odvilo tako što smo kliknuli desnim klikom kliknuli na BP_Wizard model i odabrali opciju Create Child Blueprint Class. Nakon toga, dobiveno „dijete“ smo nazvali po želji (dakle BP_WizardDevil) i promijenili smo mu mesh. Na isti način smo kreirali još dva neprijatelja. Ovime smo postigli raznolikost, tj. da neprijatelji ne budu isti, već da se oni razlikuju po svojem izgledu. Da rezimiramo, kreirali smo jednog osnovnog neprijatelja, čiju logiku nasljeđuju njegova djeca te su oni po tome isti, dok se razlikuju prema izgledu i tagovima.

13.1 ANIMACIJE

Dosad smo pričali o pronalasku i ubacivanju likova u igru, kako ga ponaći na Epic Storeu, kako migrirati podatke i slično. Pri tome smo u jednom dijelu spomenuli i animacije, tj. animacijske klase. U ovom ćemo se poglavljju bazirati upravo na animacije.

Dakle, o ubacivanju animaciju u igru je već bilo riječi na primjeru neprijatelja, tj. wizarda. Animacije, zajedno s likom, njegovim meshom možemo preuzeti s Epic Storea. Odabirom lika, u našem slučaju wizarda i biranjem opcije Mesh (Inherited) s desne strane će se pojaviti postavke pri čemu je jedan od odjeljaka nazvan Animation (o ovome smo već pričali u prošlom poglavljju). Animation odjeljak se sastoji od animacijske klase i još nekih komponenti na koje se sad nećemo pretjerano osvrnati. Kliknemo li ikonu tražilice pokraj naziva animacijske klase, otvorit će nam se prikaz u kojem vidimo blueprint za animaciju te ga možemo otvoriti. Jednom kad ga otvorimo, vidjet ćemo da postoji jako puno toga što je već otprije urađeno. Također, određene stvari smo samostalno modificirali pa je tako u Added komentaru prikazano da u trenutku kada se wizard stvori u igri starta se funkcija MontagePlay i montaža (eng. montage) koja se pokreće je LevelStart_Montage. Navedena montaža je zapravo animacija, koju smo preuzeли zajedno s ostalim podacima i ako je pokrenete vidjet će te lika koji iz jednog položaja prelazi u drugi, možemo reći iz nekog pognutog položaja se uzdiže

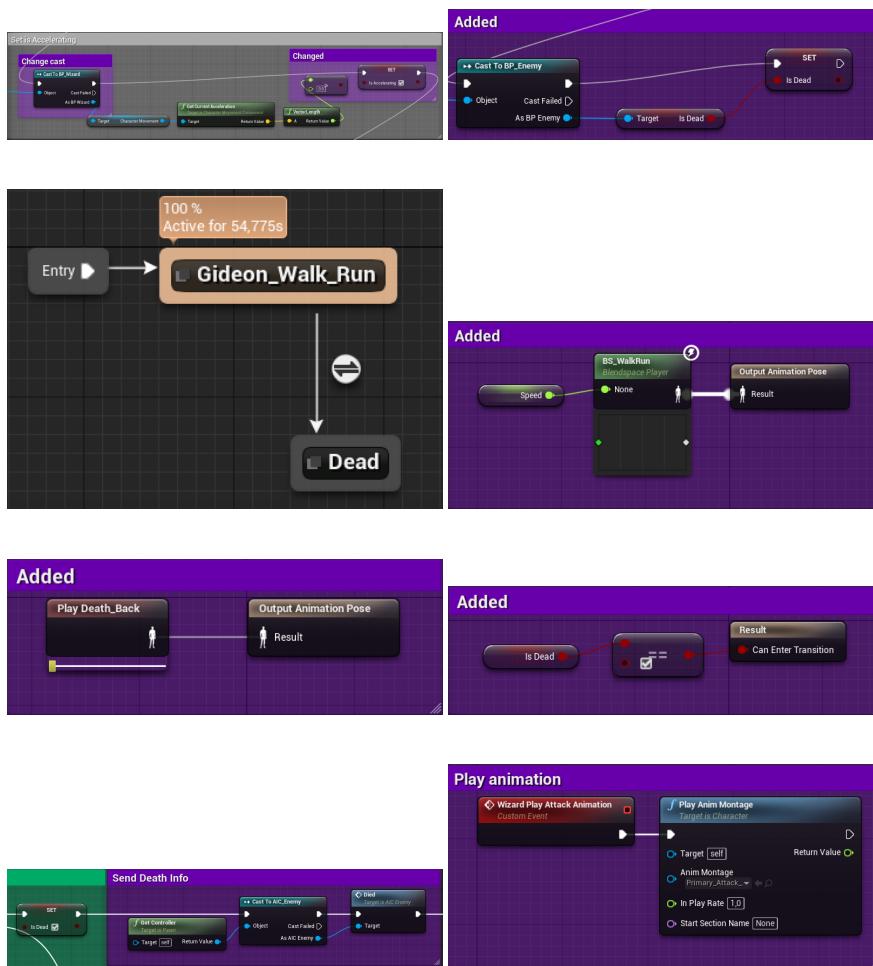
i stoji uspravno. Funkciju MontagePlay smo iskoristili kako ne bi došlo do utjecaja na Animation Blueprint jer bi zbog funkcije Play Animation animation blueprint prestao raditi (jer bi bio nadjačan).

Uz navedeno, još smo modificirali dio vezan uz smrt, tj. oduzimanje cijelog života. Naime, neprijatelj će biti mrtav jednom kada njegov HP padne na nulu, odnosno ispod nule. Tada će se varijabla IsDead postaviti na vrijednost true i na taj način se dobiva informacija o tome da je neprijatelj mrtav. Kada smo ponovili ovaj dio, prebacimo se u AnimGraph Gideon_AnimBlueprinta i pogledajmo odjeljak Ground Locomotion (ostali dijelovi, dakle grafovi ispod odjeljka Ground Locomotion su bitni za Gideona kao first person lika tako da ćemo se fokusirati na Ground Locomotion odjeljak). Vidjet ćemo jedan zanimljiv graf u čijem je središtu napisano Gideon_Walk_Run. Gotovu implementaciju animacijskog grafa smo promjenili, tj. napravili nanovo da bude jednostavnija kako bi bila lakša za shvatiti. Prijašnja implementacija je sadržavala puno drugih stanja poput Idle, JogStart, Run i slično, dok smo u ovoj implementaciji izbacili većinu tih animacijskih stanja te spojili Idle i Run stanje u trenutno postojeće Gideon_Walk_Run. Kad se otvori Gideon_Walk_Run, vidljiv je jednostavan graf s varijablom Speed, blend spaceom BS_WalkRun te OutputAnimationPoseom. Svaki od vidljivih animacijskih stanja (eng. Animation State) ima pridružen OutputAnimationPose, koji prikazuje odredene animacije, dok će blend space biti objašnjen u nastavku.

Još jedna zanimljivost koju je vrijedno spomenuti je blend space. Blend space je spajanje animacija (ako je 1D onda se spaja animacija samo po jednoj osi ovisno o vrijednosti određenog faktora (u ovom slučaju brzina), na taj način od kreirane dvije animacije dobivamo cijeli spektar animacija između te dvije). Recimo, možemo vidjeti kako će se igrač kretati kada je brzina kretanja postavljena na nulu (tada je postavljena Idle animacija, dakle lik će mirovati), kako kada je postavljena na neki veći broj (postavljena Run animacija, dakle lik će trčati) i slično. Dodane su samo ove dvije animacije, dakle Idle i Run, dok bi u savršenom slučaju trebalo umetnuti i animaciju hodanja, međutim navedena animacija nije bila uključena u skinuti paket pa je prijelaz između stanja mirovanja i trčanja malo nezgrapan, ali opet izgleda sasvim dobro. Bitno je napomenuti da se brzina blend spacea prosljeđuje putem varijable Speed i onda se na temelju prosljeđene vrijednosti odvijaju gore navedene animacije. Navedeni postupak će se odvijati sve dok neprijatelj ne umre (dok se varijabla IsDead ne postavi na vrijednost true). Tada će iz stanja Gideon_Walk_Run preći u stanje Dead (uvidite da prijelaz iz stanja Dead u Gideon_Walk_Run nije moguć) te će se pokrenuti zadana animacija (Death_Back). Prilikom postavljanja BS_WalkRun blend spacea potrebno je postaviti određene parametre. Unutar odjeljka Axis Settings postavi se parametar Maximum Axis Val na vrijednost 300.0, unutar Blend Samples

odjeljka ubacimo Jog_Fwd animaciju na zadnji čvor i Idle animaciju na prvi čvor. Moramo napomenuti da smo prebacili samo potrebne animacije, dok je u sklopu Gideon paketa sadržano puno više animacija. Dakle, prilikom migracije podataka, označili smo samo one animacije koje su nam trebale, dok smo ostale odbacili. Bitno je još napomenuti da aimoffset animacije nisu naizgled bitne, ali da se ne smiju izbrisati jer animacijski blueprint neće raditi bez njih.

Nadalje, potrebno se prebaciti u Gideon_AnimBlueprint na event graph. Ovdje se mora dodati nekoliko stvari. Prva je odjeljak Set Is Accelerating gdje se povratna vrijednost iz funkcije Cast to BP_Wizard spaja direktno s varijabljom IsAccelerating koju je potrebno postaviti na true.



14 VIZUALNI EFEKTI

U ovom poglavlju objasnit ćemo vizualne efekte (skr. vfx). Vjerujemo da više-manje svi znate što su to vizualni efekti. Neka jednostavna definicija kaže da su vizualni efekti zapravo manipulacija slikom koju gledatelj gleda. Postoji mnogi alati i programi kojima se može manipulirati vizualnim efektima. Što se tiče Unreal Engine, on također nudi navedenu opciju, pri čemu ima više editora za vizualne efekte, najpoznatiji od njih su svakako Cascade i Niagara. Cascade smo proučavali više nego Niagara (koja je noviji i kompleksniji editor). U nastavku ćemo objasniti kako raditi s Cascadeom. Razlog korištenja Cascedea je taj kako bi razumijeli efekte koje smo našli gotove pa da ih možemo manipulirati na način koji nama odgovara.

Što se tiče Cascade editora za efekte, u njemu možemo pronaći odjeljak koji se zove Emitters. Kao što se vidi, u našem projektu postoji podosta emittera (hrvatski prijevod bi bio odašiljač, međutim pošto je prijevod dosta grub, nećemo ga koristiti, već ćemo se držati izvornog termina emitter). Svaki od tih emittera mora imati svoju teksturu i tip. Kako točno izgleda kreiranje novog emittera, pokazat ćemo na primjeru. Odabirom opcije New Particle Sprite Emitter kreiramo novi emitter i on će se pojaviti s desne strane već postojećih emittera. Vidjet ćemo da mu je trenutna tekstura postavljena na default element, tj. default teksturu, koja izgleda kao plusić. Za promjenu teksture potrebno je odabrat Required opciju odmah nakon imena emittera te otici u odjeljak Emitter i nakon toga pod Material opcijom promijeniti DefaultParticle u željenu teksturu. U Required odjeljku je također važno napomenuti stavku Loop. Ako je navedena stavka postavljena na vrijednost nula, tada će emitter cijelo vrijeme raditi, a ako je postavljena na neki broj veći od nula (navedeni broj označava koliko će se puta emitter ponavljat), tada će odraditi koliko je već zadano prolaza i više neće raditi ništa. U odjeljku Spawn definiramo koliko će se elemenata kreirati, spawnati, dok u odjeljku Lifetime definiramo koliko će dugo trajati. Initial Size je potrebno postaviti, dok opciju Initial Velocity nije, a također postoji i opcija Color Over Life. Navedene opcije se automatski generiraju, dok je moguće postaviti i neke druge poput rotacije, ubrzanja, svjetla i slično.

Što se tiče emittera, postoje nekoliko vrsta (prikazano u odjeljku TypeData). Postoji obični emitter, koji se stvori odmah i on se zove CPU emitter (najobičniji, jako je modularan, radi na procesoru te ima puno mogućnosti s kojima možemo raditi, međutim ne smije se preopteretiti jer može utjecati na performanse sustava). Nakon toga postoji AnimationTrail emitter koji za sobom vuče određenu animaciju (kada pokrenete igru i stisnete ruke, tada će te vidjeti nešto slično plamenu sa strane vaše ruke, to je zapravo emitter na koji je postavljena animacija, malo je teže ispitati radi li jer je potrebno imati otvoreno puno komponenti da bi vidjeli funkcioniira li is-

pravno, također potrebna je animacija da bi ga mogli uređivati), slijedi ga Beam emitter (emitter se sastoje od više komponenata, na izlaz stavljamo beam koji uzima usmjerenost drugog emittera i onda nastavlja sebe u tom smjeru). Nakon toga, na red nam dolazi GPU Sprite (jači od procesorskog, ali je manje modularan i nema pristup svemu čemu ima pristup procesor, radi s fizikama jer je moćan, ima samo jednu baznu točku, tj. ne zna kad je izvan ekrana, problem se rješava dodavanjem bound area pomoću kojeg zadajemo dokle je on unutar ekrana na vidljivom području). Ribbon emitter je vrsta emittera koja prati neki drugi emitter (npr. ako kreiramo emitter koji će bacati u zrak neki objekt poput konfeta, onda bi kreirali novi ribbon emitter koji bi pratio naš objekt i primjenjivao svoj emitt na trag objekta (vodopad, projektil za kojem ide neki trag, uzme i mijenja svoj način emitanja ovisno o tome za što smo ga zakvačili, vatromet kad eksplodira na jednom mjestu). Još ćemo spomenuti i Mesh emitter (emitira mesheve, emitter obično stvara 2D elemente, pa ako je potrebno stvoriti 3D elemente, onda koristimo Mesh emitter koji emitira 3D elemente). Kako bi bolje razumijeli koncept emittera, preporučujemo da pogledate kako smo konfigurirali postavke emittera (poput Lifetime, Initial Size i slično) te da sami provjerite koje se još sve postavke nude, a koje nismo uključili u naš projekt.

Sada ćemo objasniti materijale koje smo pridružili emittерима. O materijalima smo već puno pričali u ovom priručniku tako da vjerujemo da ste pohvatali određene osnove pa nećemo previše ulaziti u detalje. Kreiramo materijal MT_VFX2. Za razliku od dosadašnjih materijala, u ovom će te vidjeti da postoji nešto više logike. Na početku, materijal će imati bijelu boju. Nakon toga, na materijal stavljamo masku koja ima zatamnjivanje odozdo prema gore gdje je zapravo na dnu postavljena boja u cijeloj svojoj jakoći dok kako idemo prema gore boja opada te je u konačnici postavljena na vrijednost nula, što će reći crnu boju. Nakon množenja navedenih vrijednosti faktorom 0.5, dobivamo obratnu vrijednost, tj. donji dio će biti zatamnjivan, dok će u gornjem dijelu biti postavljena potpuna boja. Nakon toga koristimo eksponent kako bi dobili više crne boje.

Vizualni efekti koje koristimo u našem radu su već gotovi, dakle preuzeli smo ih i dodatno modificirali tako da odgovaraju našoj svrsi. Recimo, preuzeli smo materijal M_Wave03. Navedeni materijal smo dodali na sfere za trening (prilikom stvaranja sfera pridružujemo im navedeni materijal) tako da one vizualno izgledaju ljepše, tj. da imaju određenu boju. Naša zamisao je bila da se vidi razlika prilikom interakcije, pa da sfere promijene boju iz plave u žutu. Dakle, kada se pojave sfere za trenining, one će biti plave boje. Kada igrač postavi ruku na tu sferu i ostvari interakciju s njom, promijenit ćemo sferu u žuto. U tu svrhu kreira se Material Instance materijala M_Wave04 (desni klik -> create material instance) i modificirali njegovu boju, tj. pos-

tavili je na žutu. Dakle, kako bi laički rekli, nije potrebno izmišljati toplu vodu. Ukoliko već postoje gotovi materijali, a dostupni su za korištenje, najbolje je preuzeti ih i modificirati prema svojim potrebama. Na taj se način skraćuje vrijeme potrebno za izradu projekta, što nam omogućuje da se fokusiramo na neke druge stvari.

Isto tako, ponekad su neke stvari višak i bolje ih je maknuti iz projekta. Na primjer, P_Gideon_Burden_Projectile particle, koji reprezentira projektil, je imao pojedine elemente u sebi koje su iz nekog razloga bile usmjerene na VR Origin. Kad bi se emitter stvorio, trag koji je bio iste boje i ponašanja kao i emitter bi se vukao i spojio s VR Originom. Ovo je greška unutar Unreal Engine okoline. Razlog ovome je vjerojatno taj što su emitteri napravljeni da rade s jednom kamerom, a problem kod nas je što postoje dvije točke s kojih gledamo igru. Jednostavnim debugom, tj. pregledavanjem svakom emittera smo provjeravali koji od njih stvara probleme i kad smo pronašli krivce, jednostavno bi ih izbrisali.

Sada se možemo zapitati koja se točno logika odvija iza određenih vizualnih efekata. Pogledajmo prvo primjer koji smo opisali prethodnom paragrafu, a tiče se promjene boje sfera prilikom interakcije s njima. Dakle, ukoliko kod trening sfere dode do BeginOverlap evenata, tj. ako dode do preklapanja, a objekt s kojim se sfera preklopila ima tag player, dakle riječ je o igraču, pokreće se funkcija SetMaterial (za postavljanje novog materijala). Navedena funkcija će promijeniti boju iz plave (materijal M_Wave03) u žutu (M_Wave04). Nakon odgode od jedne sekunde, opet će se pokrenuti funkcija SetMaterial i materijal se vraća na onaj stari (M_Wave03).

14.1 RUKA

U poglavlju smo također spomenuli što se dogada kada stisnemo šaku (pojavit će se trag, sličan plamenu, sa strane šake). Međutim, prvo moramo stisnuti šaku. U tu svrhu koristimo dodatak koji radi s kosturom ruke. Dakle, potrebno je odabratи ruku, tj. otvoriti BP_MotionController, kliknuti na komponentu mesh i na ikonu tražilice pod meshes te će se otvoriti static mesh u content browseru. Dvoklikom na static mesh otvara se novi prozor u kojem se gore desno se izabere skeletal mesh, desni klik miša na nju i odabratи opciju Add_Socket. Nakon što smo mi napravili ovaj postupak, u popisu s lijeve strane ćete vidjeti novi socket koji smo kreirali naziva VFX_Base. Također smo kreirali još jedan socket naziva VFX_Tip. Ako želimo malo bolje pregledati ruku ili je bolje pozicionirati, možemo koristiti tipke w, a, s, d za micanje.

Nakon što smo kreirali sockete, možemo izaći iz Skeleton pogleda i prebaciti se u Animation pogled. U prozoru ispod prikaza šake, kliknemo desni klik miša i odaberemo opciju Add Notify. S desne strane će se otvoriti prozor i

u odjeljku Anim Notify posložimo određene vrijednosti. Prvo, PSTemplate željenog vizualnog efekta. Nakon toga, moramo upisati imena socketa. Opcije First Socket Name i Second Socket Name. I zapravo smo riješili sve što smo trebali. Potrebno je još napomenuti da će se opisani postupak pokrenuti ako je aktivno stanje MannequinHand_Right_Grab (zapravo cijeli postupak je sadržan unutar navedenog stanja).

Objasnit ćemo još jedan dio vezan uz stiskanje šake. Prebacimo se u MotionControllerPawn dio. Kada igrač stisne (eng. grip) šaku, kreiramo čestice (eng. particles). Čestice su vrlo karakterističan pojam u svijetu igara. Njih povezujemo s emitterima koji djeluju kao izvor za čestice. Pomoću njih simuliramo neke efekte iz stvarnog svijeta, poput plamena. U našem projektu, kada se desi event InputAction GrabLeft ili InputAction GrabRight, LeftHandMotionTrigger (točno mjesto na kojem se nalazi sfera s kojom vršimo interakciju s drugim sferama, tj. sfera s kojom triggereamo druge sfere) pridružujemo funkciji SpawnEmitterAttached te na navedenu sferu pridodajemo emitter zapisan u opciji Emitter Template funkcije SpawnEmitterAttached. Nakon toga, postavljamo referencu na emitter (postavljamo vrijednost variable ParticlesLeftReference). Nakon što otpus-timo stisk šake, funkcijom IsValid provjeravamo postoje li određene reference (dakle, varijabla ParticlesLeftReference sadrži neke vrijednosti), onda brišemo čestice funkcijom DestroyComponent, a referencu postavljamo na vrijednost „ništa“ (eng. none). Navedeni postupak je objašnjavao stisk šake na lijevoj ruci, međutim isti princip vrijedi i za pritisak šake na desnoj.

Za kraj ovog dijela, reći ćemo nešto o emitteru koji se pojavi kada smo pronašli ability. Dakle, kada pronađemo dobar ability, kreiramo čestice u odjeljku Create Charge Particles. Ovdje stvaramo isti emitter na obje ruke. Stvoreni emitter referenciramo stavljajući vrijednost odgovarajućim varijablama (Charge Left Preference i Charge Right Preference). Vjerujem da shvaćate, referenciranje se koristi kako bi mogli brisati kasnije (na otpuštanje šake) na način kako je to bilo opisano u prethodnom paragrafu.

14.2 PROJEKTILI

Za svaki projektil smo također postavili određene čestice. Prebacit ćemo se prvo u BP_FireProjectile. Dakle, kad se desi event Spawn Emitter (pokreće funkcija Apply Damage), tada se pokreće funkcija SpawnEmitterAtLocation u kojoj je posložen emitter na pogodak, točnije rečeno, kad projektil eksplodira, tada će se zapravo pokrenuti navedena funkcija. Vizualizirajmo si to ovako. Projektilom ćemo pogoditi neprijatelja. Kada ga pogodimo, projektil će se sudariti s neprijateljem, a sudar će okinuti pokretanje emitera koji će odaslati čestice te će se pojaviti efekt raspršivanja. Na navedeni način je svaki projektilu priložen emitter povezan s projektilom kad on eks-

plodira. Postoji jedan izuzetak pa ako otvorimo BP_IceProjectile, vidjet će te da se zapravo stvaraju dvije SpawnEmitterAtLocation funkcije. Razlog tome je što jedan od emittera nije vizualno odgovarao s drugim pa je malo promijenjena visina na kojoj će se on stvarati. Naravno, potrebno je primijetiti funkciju DestroyActor koja služi za uništenje projektila što je razumljivo, jer kad se projektil sudari s metom, odmah ga moramo uništiti, ali prije uništenja projektila, pokreće se emitter na opisani način.

Svaki od emittera na kojim smo radili ima glavni dio, glavnu česticu (eng. main particle) i trag (eng. trail) koji ga prati. U slučaju ledenog projektila, glavni dio se sastoji od leda koji izgleda kao strelica dok trag izgleda poput snježne oluje koja ga prati. Vatreni projektil kao glavni dio ima plamen, a trag koji ga prati čine dvije vrteće spirale. Dodavanje navedenih komponenti se čini tako da se klikne na Add Component (zeleni gumbić u gornjem lijevom kutu), iz popisa odaberemo opciju Particle System te s desne strane, u odjeljku Particles odaberemo opciju Template i postavimo odgovarajući predložak. Isto vrijedi kako za glavni dio, tako i za trag. Isti postupak smo kreirali i za zračni projektil (AirProjectile) koji je specifičan jer ima dvije trail komponente te za grmljavinski projektil (ThunderProjectile).

Za kraj dijela o projektilima, možda ste u Spawn Emitter događaju primijetili Impact Location. Njega dobivamo iz Apply Damage dijela, koji može pozvati navedenu varijablu i proslijediti je Spawn Emitteru. Tada će se na prosljedenoj lokaciji stvoriti emitter. Opisani projektili su oni koje ispaljuje igrač. Kada pričamo o projektilima koje ispaljuje neprijatelj, prebacit ćemo se u blueprint naziva BP_WizardAttack1Projectile. Kao što vidite, on je de facto isti kao i igračevi projektili. Jedina razlika je što nema trail komponentu, već je ona sadržana u glavnem dijelu.

14.3 ENEMY

Za kraj ovog poglavlja moramo objasniti vizualne efekte povezane s neprijateljem. Prvo što moramo napomenuti jest da se prilikom stvaranje neprijatelja stvara i emitter koji prikriva da se neprijatelj samo pojavio. Nadalje, što se tiče projektila, o njihovim vizualnim efektima smo već nešto pričali o potpoglavlju o projektilima. Međutim, nismo spomenuli kako će neprijatelj točno ispaljivati projektile. Jedan od načina je da se odabere animacija (Anim to Play) u kojoj neprijatelj ispaljuje projektile, pauziramo navedenu animaciju (Pause Anims), nakon toga postaviti poziciju (Initial Position) u animaciji na kojoj bi se ispaljivanje trebalo desiti i na navedenu poziciju dodamo element. Na opisani način, definirali smo vrijeme u animaciji kada će neprijatelj ispaljivati projektile. Prilikom stvaranja projektila, doći do i do stvaranja emittera koji je povezan uz projektil.

Što se tiče obrade smrti neprijatelja, logika je sadržana u odjeljku Death

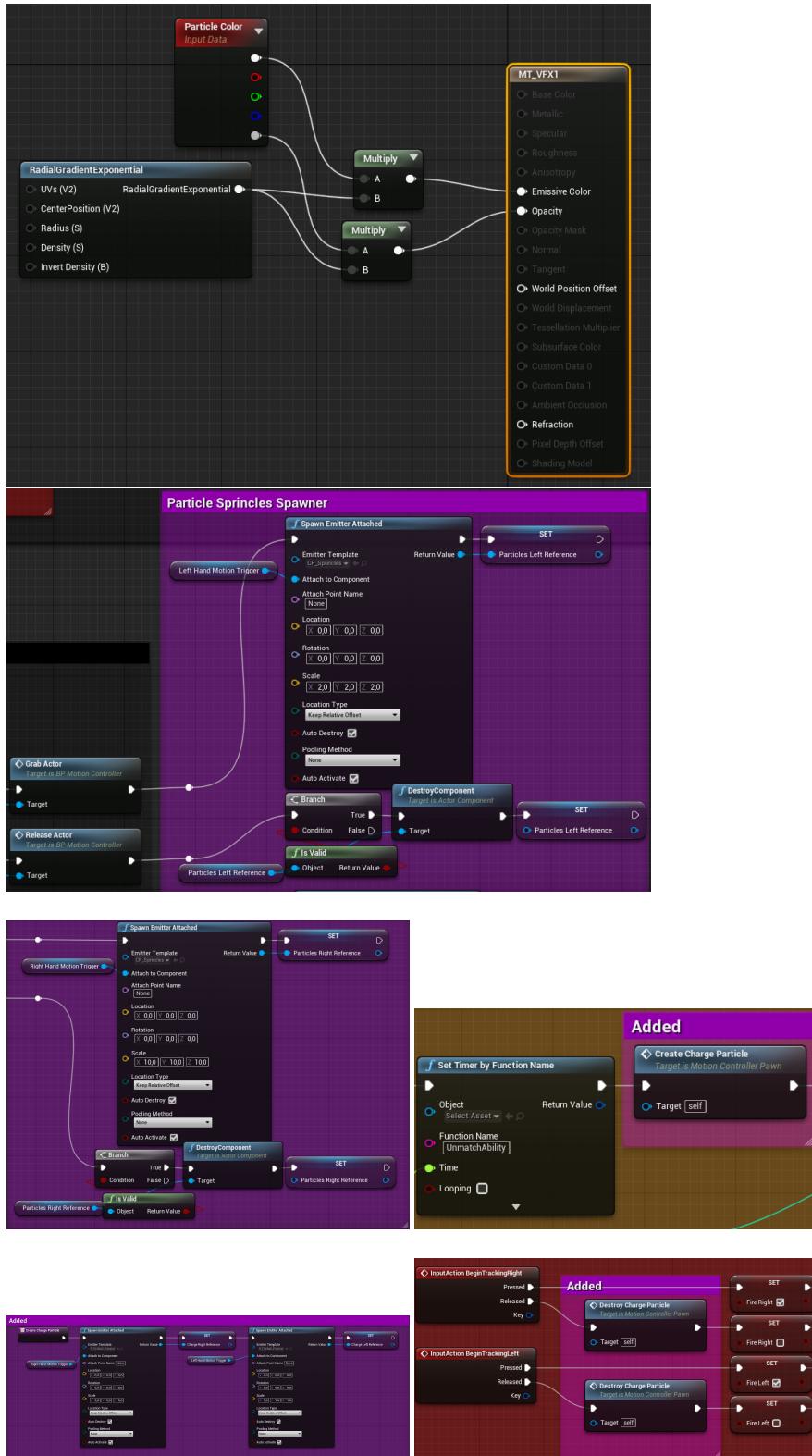
Handler. Na početku se nalazi odgoda od jedne sekunde jer je to vrijeme koje je potrebno da neprijatelj padne na pod, nakon što mu je oduzet sav život (pritom se pokreće određena animacija). Mogućnost kolizije se postavi na nulu jer bi inače kolizijska kapsula koja se nalazi oko neprijatelja i dalje bila aktivna te je na ovaj način deaktiviramo. Nakon još jedne odgode od sekunde stvara se novi emitter koji će se prikazati kada je neprijatelj na podu pri čemu se uzima neprijateljeva lokacija, uz malo modifikaciju Z-lokacije. Nakon toga dolazimo do varijable ScaleSave koja je potrebna da bi se moglo pratiti koliko je mesh velik, pri čemu se iz varijable Mesh dohvata parametar Relative Scale 3D te se spremi u varijablu ScaleSave. Nakon još malo odgoda, stvaramo još jedan emitter (vatra u sredini) i nakon toga dolazimo do koncepta vremenske crte (eng. timeline).

Vremenske crte u Unreal Engineu su funkcije koje se događaju na tick kad se pozovu. Ako ne želimo da se na svaki tick cijelo vrijeme nešto događa, provjerava i sl., kada dođemo do neke akcije, napravimo jedan timeline i pozovemo ga. Naš timeline se zove DeathGoDown i ako ga malo bolje pogledate, vidjet će te da je uključena opcija Play from Start. Ako otvorimo timeline (dva put kliknemo na DeathGoDown) vidjet ćemo jednostavan graf koji nam nije toliko bitan. Bitna nam je opcija Length koju postavimo na određenu vrijednost i onda će timeline trajati toliko koliko smo naveli u opciji Length. Nakon toga, na svako ažuriranje (update) postavit će se pozicija neprijatelja za onu poziciju na kojoj je sad umanjenu za jedan te će pomoći funkcije SetActorScale3D postaviti novi Scale 3D koji smo prije spremili u varijablu ScaleSave pri čemu vrijednost u navedenoj varijabli dijelimo s jakom malim brojem koji je veći od jedan tako da se na svaki tick neprijatelj malo smanji. Kada je cijeli navedeni proces odrađen, uništava se actor, tj. neprijatelj. Što se tiče vizualnih efekata, oni će se sami uništiti (kada završe) jer je opcija Auto Destroy (dostupna prilikom stvaranja emitera) postavljena na true vrijednost. Spomenut ćemo još kako prikazujemo zdravlje igrača. Nećemo ponovno objašnjavati kako funkcioniра zdravlje i HP jer je o tome već bilo riječi. Ovdje ćemo se fokusirati na funkciju HealthVisualUpdater. Dakle, ukoliko primimo štetu, ali ta šteta nije fatalna, još imamo života, onda se izvršava navedena funkcija. Funkciji prosljeđujemo HP, nakon toga čitamo iz varijable PercentageHealthForIndicator u kojoj je trenutno spremljena vrijednost 0.5 i dijelimo je s jedan. Vrijednost u navedenoj varijabli označava postotak na kojem želimo da se aktivira ostatak HealthVisualUpdater funkcije. U našem slučaju, kada igrač ima 50 ili manje posto preostalog života. Dakle, jedan dijelimo s pola i dobiveni rezultat množimo s trenutnim HP-om igrača te provjeravamo je li umnožak manji od maksimalne vrijednosti života. Sada ćemo pojasniti jedan element koji se zove DamageEffect. Njemu je svojstvo Color Grading -i Global -i Saturation postavljeno na crnu boju. Njegov Blend Weight postavljen je na nulu zato jer bi inače ekran bio crno bijelog spektra. Ukoliko je provjera

prošla (dakle, umnožak koji smo objasnili netom prije spominjanja DamageEffecta mora biti manji ili jedan maksimalnom životu), trenutni HP dijelimo s maksimalnim životom te dobivamo postotak u odnosu na jedan (jer Blend Weight mora biti u rasponu od jedan do nula). Dobiveni rezultat se postavlja kao vrijednost u Blend Weight odgovarajućeg DamageEffecta. Kako bi vratili život pokreće se funkcija HealthRegeneration koja regenerira HP, a to ima efekta na funkciju HealthVisualUpdater koja miče crnu boju s ekrana.

14.4 STVARANJE KAMENA

Moramo još opisati što se događa prilikom stvaranja samog kamena. Stvaranjem kamena stvaraju se i tri emittera. Jedan od emittera će stvoriti efekt prašine, nakon odgode od 0.75 sekunde se stvara drugi emitter koji će stvoriti šiljke ispred i sa strane kamena, dok je treći efekt isti onaj koji se desi prilikom raspadanja kamena, samo s malom modifikacijom (promjenjena tekstura i boja). Treći emitter će se pokrenuti nakon pet sekundi jer je to životni ciklus kamena. Dakle, ukoliko kamen unutar tih pet sekundi nije uništen od strane neprijatelja, tada će se pokrenuti treći emitter i kamen će se raspasti. Otprilike je jasno, ali moramo napomenuti, da će se emitteri stvoriti na istoj lokaciji na kojoj je kamen. Naravno, određeni parametri su dodatno podešeni, poput rotacije ili lokacije, što možete vidjeti na samom grafu. Još je bitno napomenuti da kad se kamen raspadne, odgodom ćemo učiniti to da se raspadnute krhotine kamena još dvije sekunde nalaze na podu te njihovu mogućnost kolizije funkcijom SetActorEnableCollision postavljamo na false. Nakon dvije sekunde, raspadnuti komadići kamena će propasti kroz pod i nestati (uništiti se).





15 ZVUČNI EFEKTI

Uz vizualne, za bolji doživljaj u igri koristit ćemo i audio efekte. Glazba, tj. zvukovi u igrama su vrlo važni jer pomoću njih možemo naglasiti da nam se



bliži opasnost, ili da smo blizu nekog važnog traga i slično. Zvukove možemo povezati uz razne akcije, poput ispučavanja projektila ili oduzimanja života i upravo će na tim stvarima biti fokus prilikom dodavanja audio efekata u igri.

Što se tiče ubacivanja zvukova u igru, postoje razne stranice na kojima su dostupni besplatni audio efekti. Jedna od takvih je Freesound (dostupna na



adresi <https://freesound.org/>). Po potrebi, pojedini efekti su modificirani tako da bolje zvuče u kontekstu u kojim ih koristimo. S druge strane, za neprijatelja smo koristili zvukove koji dolaze s Gideonom (neprijateljevo puštanje, primanje štete, smrt). Dakle, osim vizualnih efekata, među podacima je moguće pronaći i audio efekte. Bitno je još napomenuti da ćemo koristiti zvukove u WAV formatu (Unreal Engine obično koristi ovaj format). Druge formate, poput MP3 ili MP4, može učitati, međutim ne smatra ih direktno audio datotekama. Sve audio datoteke se nalaze na putanji Content -> Sounds.

Kako bi se zvukovi čuli u prostoru, ne samo na lijevo, odnosno desno uhu, potrebno je otvoriti audio datoteku i pod opcijom Attenuation -> Attenuation Setting postaviti određenu komponentu. Naravno, prethodno ju je potrebno kreirati (desni klik unutar Content browser -> Sounds -> Sound Attenuation), točnije potrebno je kreirati dvije navedene komponente koje će biti naziva SA_Natural i SA_Explosion. Nakon što se komponente kreiraju, moraju se podesiti određeni parametri pa s toga otvorimo SA_Natural komponentu i postavimo sljedeće vrijednosti: u odjeljku Attenuation Distance vrijednost parametra Attenuation Function potrebno je postaviti na Natural Sound, Attenuation at Max (dB) (opcija povezana uz glasnoću) na -25, Fallout Mode (zvuk cijelo vrijeme djeluje jednakim intenzitetom) na Continues, Attenuation Shape (kako se širi zvuk) na Sphere, Inner Radius (radijus na kojem se ne događa promjena glasnoće, dakle u radijusu od neke točke koja pušta zvuk u svim smjerovima se za određenu vrijednost zvuk neće stišavati, nego će se stišavati nakon što se pređe definirani radijus) na 400,0 te FallOut Distance (vrijednost nakon koje se zvuk neće više čuti) na 5000,0. U odjeljku Attenuation Spatialization potrebno je postaviti vrijednost parametra Spatialization Method na Binaural. Ostale vrijednosti nije potrebno mijenjati. Jednaku stvar je potrebno ponoviti i za SA_Explosion. Jedina razlika SA_Explosion komponente od prethodne je što je parametar

Attenuation at Max (dB) potrebno postaviti na vrijednost -15 (vrijednost je povećana jer kad se desi eksplozija, glasnije će se čuti), dok je Inner Radius parametar potrebno postaviti na vrijednost 1500,0. Još je potrebno napomenuti da se SA_Explosion (audio datoteke koje u nazivu sadrže sufiks Hit) dodaje na zvukove koji simuliraju eksploziju, dok se SA_Natural dodaje na sve druge zvukove koji u sebi imaju određen promjenu (audio datoteke koje u nazivu sadrže sufiks FlyingLoop), dok na audio datoteke koje u svom nazivu imaju sufiks Cast, ne dodajemo navedene Sound Attenuation komponente.

Bitno je objasniti još jednu komponentu, a to je sound cue. Sound Cue je audio komponenta koja omogućava aranžiranje i modificiranje zvukova. Ona se dodaje desnim klikom, odabirom opcije Sounds -i Sound Cue. Potrebno je dodati tri ovakve komponente koje se trebaju nazvati SQ_AbilityReady, SQ_MusicLoop i SQ_HeartBeat. U SQ_HeartBeat potrebno je dodati odgovarajuću WAVE datoteku (audio zapis otkucaja srca) i postaviti funkciju Looping te Output komponentu. Ovo je jedan od načina, drugi način je da se izabere navedena WAVE datoteka te unutar Sound odjeljka postavi vrijednost Loop. Tada nam navedena Looping funkcija ne bi bila potrebna i mogli bi spojiti WAVE datoteku direktno s Output komponentom. Nakon toga, u datoteci SQ_MusicLoop potrebno je dodati dvije WAVE datoteke, prva je A_MusicIntro (uvodni dio glazbe koji se ne ponavlja), dok je druga A_MusicLoop (stalna pozadinska glazba) , pri čemu je A_MusicLoop potrebno spojiti s funkcijom Looping. Zatim je potrebno dodati Concatenator komponentu koji služi za sekvenčijalno puštanje zvukova, tj. zvukovi se puštaju redom kako su unijeti. U našem slučaju prvo se pušta A_MusicIntro, dok će se nakon njega pustiti A_MusicLoop. Naravno, na kraju je potrebno izlaz spojiti s Output čvorom. Preostala nam je SQ_AbilityReady komponenta koja funkcioniра na sličan način kao prethodno navedene dvije (koristimo A_UntilRelease (zvuk koji traje sve dok se ne ispuca ili ne unmatcha ability) i A_AbilityFound zvuk (koji se pokreće prilikom pronalaska abilityja) te Output komponentu), međutim u ovom primjeru je potrebno dodati novi čvor imena Mixer. Mixer spaja stvari, tj. istovremeno pušta zvukove, pri čemu nije bitno koji je zvuk ušao prije, kao kod Concatenatora. Nakon što su kreirani navedeni sound cueovi, potrebno je oticiti unutar Footsteps direktorija i ondje kreirati još jedan sound cue naziv SQ_EnemyFootsteps. U navedenoj komponenti potrebno je dodati zvukove hodanja po blatu kao i zvukove normalnog hodanja pri čemu je dodano po pet takvih audio datoteka. Nakon toga, kreiraju se dvije Random komponente. Random komponenta nasumično uzima po jedan od zvukova (dakle, jedan zvuk hodanja po blatu te jedan normalan zvuk hodanja), stavlja ih u Mixer te proizvodi zvuk. Svim navedenim komponentama mora biti postavljen SA_Natural sound attenuation.

Prebacit ćemo se sada na Match Ability. Kada se pronade ability na način kako je to opisano u jednom od poglavlja, eventom PlayAbilitySoundEffect pokreće se zvučni efekt. U navedenom eventu prvo se funkcijom Deactivate deaktivira audio komponenta zapisana u varijabli AudioAbilityFound. Kad se otvorí varijabla, s desne strane u popisu, u odjeljku Sounds, pod opcijom Sound je vidljivo da je navedena varijabla povezana s SQ_AbilityReady zvučnim efektom. Nakon toga, opcijom SetSound se postavi zvuk koji se želi pokrenuti (pod opcijom New Sound) te se taj zvuk postavi u varijablu AudioAbilityFound te se funkcijom Activate aktivira zvuk tako da se odsvira željeni audio efekt.

Sljedeća stvar vezana uz zvuk se dešava prilikom stvaranja projektila. U FireAbility eventu opali se određeni ability te se pomoću funkcije PlaySound2D pokreće zadani zvuk. Dakle, prilikom ispučavanja vatre bit će pokrenut A_FireCast zvuk, prilikom ispučavanja zračnog projektila A_AirCast zvuk. Prilikom ispučavanja munje zadano je da se pokreće A_ThunderCast zvuk, dok će se ispučavanjem ledenog projektila pokrenuti A_IceCast zvuk. Što se pak tiče UnmatchAbility eventa, u tom djelu se stišava zvuk ako je prošlo vrijeme u kojem je ability mogao biti bačen. Dakle, ukoliko se unutar koda ability unmatcha, tada se i zvuk koji je aktivan prilikom pronalaska abilityja gasi.

BP_AirProjectile sadrži logiku za pokretanje zvukova koji su povezani uz zračni projektil. Navedeni projektil sadrži audio komponentu, tj. varijablu naziva Audio (navedeni varijabla se mora dodati u bazni projektil, dakle u BP_Projectileu) u koju pohranjujemo zvuk pomoću opcijom SetSound (u opciji New Sound se postavi SQ_AirLoop audio efekt) te aktiviramo zvuk zapisan u varijabli funkcijom Activate na BeginPlay događaj. Prilikom kolizije projektila s nekim drugim objektom pokreće se event SpawnEmitter te se funkcijom PlaySoundAtLocation izvodi zvuk A_AirHit koji se postavlja pod opcijom Sound. Nakon što se izvrši izvođenje zvuka, zvuk pohranjen u varijabli Audio se deaktivira funkcijom Deactivate te se pomoću funkcije DestroyActor uništava projektil. Što se tiče ostalih projektila, navedenih u prethodnom odjeljku uz dodatak neprijateljskog projektila, logika za zvuk se odvija na isti način, samo što pod funkcijom PlaySoundAtLocation postavljamo drugi zvuk.

Animacije smo susreli u prijašnjem poglavlju gdje smo detaljno objasnili kako funkcioniraju. U ovom poglavlju ćemo spomenuti kako možemo uposlitи animaciju da na vrlo jednostavan način upravlja nekim zvukovima unutar igre. Za tu svrhu objasnit ćemo animaciju Jog_Fwd. Prije toga, potrebno je desnom tipkom miša na animation notify track i odabirom opcije Add Notify -i PlaySound dodati novi notify (koji je zapravo okidač zvuka) i onda postaviti da se pušta tzv. sound cue kojeg je potrebno nazvati SQ_EnemyFootsteps. Dodavanje zvuka je napravljeno na vrlo jednostavan

način. Prolazimo animacijom i gledamo kada trebamo dodati zvuk. Konkretno, u ovom primjeru se gleda kada će lik zakoračiti na zemlju, odnosno staviti nogu na podlogu i tada dodajemo audio efekt. Još jedan primjer je vezan uz smrt neprijatelja. Smrt neprijatelja prati animacija Death_Back i navedenoj animaciji smo postavili zvuk na tri mjesta. Dakle, potrebno je proći animacijom i gledati kada neprijatelj prvi put primi štetu te ovdje postaviti zvuk SQ_GideonEffortDeath. Nakon toga pronaći mjesto u animaciji kad neprijatelj prvi put takne pod i postaviti zvuk SQ_EnemyFootsteps te na kraju pronaći kad neprijateljeve noge udare u pod i postaviti isti audio, dakle SQ_EnemyFootsteps. Animacija Primary_Attack_A_Medium također sadrži audio efekte (SQ_GideonEffortAttack i A_DarkCast) koji su postavljeni sukladno kretnjama neprijatelja. Navedene audio efekte se postave na sljedeći način: desna tipka miša na sound editor, odabratи opciju Add Notify -i PlaySound, kliknuti na stvoreni audio efekt te u opcijama s desne strane, unutar Anim Notify -i Sound izabrati odgovarajući zvuk. Na prikazane načine spojena je animacija i određeni zvuk te nas dalje nije briga kad će se taj audio efekt desiti, već sve to ostavljamo animaciji, koja će biti okidač za pokretanje zvuka.

Vezano uz neprijatelja, kada neprijatelj primi štetu (točnije, prilikom pokretanja AnyDamage događaja) funkcijom PlaySoundAtLocation pokreće se audio Gideon_Effort_Block (audio efekt koji dolazi s Gideon podacima, dakle preuzet s Epic Storea). Klikom na ikonu tražilice pokraj naziva audio efekta i odabirom navedenog audia pobliže se može vidjeti kako funkcionira Gideon_Effort_Block sound cue. Kao što je moguće vidjeti, gotova implementacija je vrlo slična našoj, osim što se u gotovoj implementaciji koristi Dialogue Wave dok mi koristimo običan WAVE. Dakle, svaki put kada neprijatelj primi štetu, nasumično, random će se pustiti drugi audio efekt. Još jedan dio je bitan u ovom dijelu, a to je potrebno funkcijom GetAllActorsOfClass dohvatiti sve neprijatelje (unutar parametra Actor Class postavi se BP_Enemy) te za svakog dohvaćenog neprijatelja, na njegovoј lokaciji (dohvaćenoj pomoću funkcije GetActorLocation) funkcijom PlaySoundAtLocation puštamo zvuk Gideon_Effort_Cheer jednom kada igrač umre, tj. ostane bez HP-a.

U ovom odjeljku, objašnjen je dio vezan uz audio efekte i regeneraciju zdravlja. O funkciji IsPlayerRecentlyDamaged je već bilo riječi, konkretno, u njoj se provjerava može li igrač regenerirati zdravlje te se stvara emitter na lokaciji PlayerHitPointArea koji prikazuje regeneraciju života (pri čemu je emitter skaliran kako bi bio kraći). Nakon toga, kreira se varijabla AudioHeal (koja je tipa Audio Component) koja se deaktivira, opcijom SetSound u varijablu se postavlja novi zvuk A_Heal te se varijabla aktivira. Navedeni zvuk se prekida (fade out) u dva slučaja. Prvi je unutar funkcije HealthRegeneration gdje se provjerava je li se igrač u potpunosti regenerirao (pomoću

funcije FadeOut zvuk zapisan u varijabli AudioHeal se utišava, pri čemu je fade out trajanje potrebno postaviti na dvije sekunde). U drugom slučaju, zvuk će se prekinuti prilikom zadobivanja štete (unutar PlayerHealthHandler odjeljka) pri čemu je fade out trajanje također postavljeno na dvije sekunde. Da rezimiramo, zvuk će se prekinuti, pritišati ili kad je HP pun (dakle, život je u potpunosti regeneriran) ili kad je zaprimljena šteta. Što se tiče zdravlja igrača, postoji još jedan dio koji je vezan uz audio efekte. Dakle, kada igrač ima manje od 50% HP-a, glazba se stiša, a igrač počne čuti otkucaje srca, tzv. hearthbeat. Prvo, potrebno je kreirati novi glavni kanal (u content browseru unutar Sounds direktorija desni klik miša i odabere se opcija Sound Class koja kreira novi kanal kojeg je potrebno imenovati SC_Changing). SC_Changing nije ništa drugo nego vlastito kreirani kanal. Ukoliko se otvorи neka od datoteka u Wave formatu, tada se može vidjeti da je njihov kanal postavljen na Master, koji je bazna klasa za sve zvukove u Unreal Engineu (prikazano unutar Sound -i Class). Razlog stvaranja novog kanala je dodjeljivanje zvuka u novi Sound Class, zatim se može manipulirati novostvorenim Sound Classom, što će u konačnici utjecati i na zvuk. Potrebna su dva kanala pa je potrebno kreirati SC_Damaged koji će se koristiti za otkucaje srca, dok će se SC_Music kanal koristiti za pozadinsku glazbu. Nakon toga, otvorimo sound cue SQ_HeartBeat te unutar Sound -i Class postaviti na SC_Damaged. Na isti način postupamo s SQ_MusicLoop gdje unutar Sound -i Class postavimo na SC_Music. Zadnja stvar koja je potrebna ovdje je nova audio komponenta tipa sound mix modifier koja će se zvati SMM_Music (desni klik miša unutar Sound direktorija -i Sounds -i Classes -i Sound Mix Modifier). Nakon što imamo navedene stvari, prebacujemo se u MotionControllerPawn event graph unutar odjeljka Set Sound Channels. Odjeljak je potrebno započeti funkcijom SetBaseSoundMix (postavlja bazni sound mix te se pokreće na BeginPlay event) u koju se, pod opcijom In Sound Mix, postavlja sound mix modifier SMM_Music. Nakon toga, dodaje se funkcija SetSoundMixClassOverride u kojoj se unutar opcije In Sound Mix Modifier mora postaviti određeni sound mix modifier. Navedena funkcija modifcira ili nadjačava željene postavke preko sound mix modifiera, a utječe na zvukove sound klase koju joj zadajemo. Zatim se funkcijom PlaySound2D pokreće zadani zvuk (SQ_MusicLoop). U grafu je nakon toga potrebno dodati još jednu funkciju SetSoundMixClassOverride u kojoj je vidljivo da je Volume postavljen na 0,01 (ako se navedena vrijednost postavi na nulu, tada se zvuk ne može pojačati, zbog toga je uzeta mala vrijednost) te se dodaje funkcija PlaySound2D u koju dodajemo zvuk SQ_HeartBeat (navedeni zvuk je cijelo vrijeme aktivan u pozadini, ali se ne čuje jer je jako tih) pri čemu je parametar Volume potrebno postaviti na vrijednost različitu od nule jer ako je postavljeno na nulu, tada funkcija neće ispravno raditi. Da rezimiramo. Kada igrač primi štetu i ima 50 ili manje od 50 HP-a, pokreće se funkcija SetSoundMixClassOverride te se prilikom

toga postavlja novi volume na 0,5 na pozadinsku glazbu i na 9 na otkucaj srca, što bi značilo da se pozadinska glazba utiša, dok se zvuk otkucaja srca poveća. Kako se igrač regenerira, i zdravlje mu ide prema 50 HP-a, ponovno je potrebno pokrenuti SetSoundMixClassOverride i postaviti glasnoću (volume) na stare vrijednosti, dakle pozadinsku glazbu na 2,5 dok otkucaje srca na 0,01 glasnoće.

Za kraj poglavlja o audio efektima, objašnjeno je povezivanje zvukova s kamenom. Za ovaj dio, potreban je DM_Rock blueprint. Prilikom stvaranja kamena (na događaj BeginPlay), pomoću funkcije GetActorLocation dohvaćamo njegovu lokaciju te na dobivenoj lokaciji, funkcijom PlaySoundAtLocation puštamo zvuk A_RockEmerge. Dakle, zvuk koji ćemo pustiti će simulirati podizanje kamena iz tla. Nakon nekog vremena, poslije posljednjeg emittera ponovno se pokreće funkcija PlaySoundAtLocation (prvo je potrebno saznati lokaciju kamena funkcijom GetActorLocation) pri čemu se pušta zvuk A_RockBreak (dakle, navedeni zvuk daje dodatnu vrijednost vizualnom efektu raspadanja kamena na komadiće).





16 PONOVNO POKRETANJE IGRE

Prije početka ponovnog pokretanja igre postoji jedan uvjet. Događaj `RestartFromMenu`, koji se nalazi na `widgetu`, pozove događaj `CloseMenuIfTeleporting` koji



zatvara otvoreni izbornik. Dakle, zatvoreni izbornik je uvjet za daljnji nastavak ponovnog pokretanja. Nakon toga postavimo varijablu `PlayerIsDead` na vrijednost `True` (jer prilikom ponovnog pokretanja igre zapravo umiremo i krećemo ispočetka). Kada varijablu postavimo ovako, to će biti znak da se neke stvari poput teleportacije, kretanja, otvaranja izbornika neće dozvoliti, dakle, onemogućit ćemo ih. Nakon toga, pozove se event `Reset Spawner`. U navedenom eventu dohvaćamo brojač koji je zapisan u varijabli `Timer`, pomoću funkcije `ClearAndInvalidateTimerByHandle` uništavamo dohvaćeni brojač te ga nakon odgode od šest sekundi ponovno pokrećemo funkcijom `SetTimerByFunctionName`. Ovakva implementacija osigurava da se prilikom završetka ponovnog pokretanja odmah ne stvaraju neprijatelji, već se pričeka nekog vrijeme. Nastavljujući logiku dalje, pokreće se funkcija `StartCameraFade`, pauzira se funkcija `IsPlayerRecentlyDamaged` (više ne gledamo navedenu funkciju), HP postavljamo na 100, ukoliko je zvuk za `heal` postavljen, funkcijom `Fade Out` ga pritišamo, maknemo.

Nakon odgode, pauziramo stvaranje novih neprijatelja te dohvaćamo trenutno ime razine na kojoj se nalazimo. Za nas je ovaj dio važan tako da znamo da ponovno stvaranje radi kao na testnoj razini, tako i na glavnoj razini. Ako kod provjere utvrdimo da se radi o testnoj razini, tada se postavljaju vrijednosti te mape, a ako nije riječ o probnoj razini, onda se sigurno radi o razini *Level1* jer nema drugih mapa. Što se tiče vrijednosti zapisanih u funkciji `SetWorldLocation` (u koju dolazimo ako se iz prethodnog uvjeta zaključilo da se radi o probnoj mapi), za *X* i *Y* lokaciju smo kopirali koordinate elementa koji je postavljen u svijet na lokaciju na koju se igrač treba teleportirati prilikom restartanja, dok smo *Z* lokaciju dobili pomoću funkcije `GetWorldLocation` koja kao cilj ima kolizijsku kapsulu.

Nakon navedenih postupaka, postavlja se vidljivost komponente `plane` za učitavanje. Učitavanje za korisnika smo implementirali pomoću dvije komponente: `LoadingText` (tekstualna komponenta) i `LoadingScreen` (*media*). `LoadingText` je tekst žute boje na kojem piše „Loading...“, ima zadani `DefaultTextMaterial`. `LoadingLoop` se mora učitati prije nego što se išta drugo radi s njim jer je on `MediaComponent`. Dakle, kad se razina (*Level1*) učita, prvo dodajemo `File Media Source` komponentu koja se nazove `MS_LoadingLoop`. Kada se komponenta otvorí, vidjet ćemo opciju `File > File Path` u kojem odabiremo video koji ćemo koristiti tako što ga postavimo unutar mapu `Content > Movies` (obavezno ga postaviti u mapu `Movies`). Takoder, pozicioniramo li se malo dolje, vidjet ćemo dio koji je potreban prilikom kreiranja razine. Ukoliko `Skylight` na bilo koji način

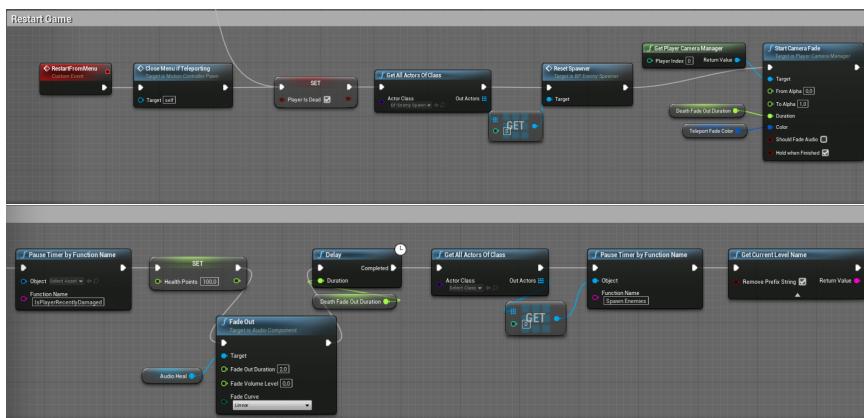
utječe na razinu, u *level blueprintu* ga treba isključiti, tj. pobrisati. Treba još napomenuti da ovaj dio nije potreban ukoliko u igri koristimo dinamično svjetlo.

Pošto smo u prethodnom odlomku spominjali pojam *media* (medij), ovde ćemo pojasniti kako ga uvesti u projekt. Nakon što smo kreirali **File Media Source** komponentu, dodajemo **Media Player** komponentu koja se nazove **MP>LoadingLoop** (prilikom stvaranja navedene media player komponente pojavit će se dijaloški okvir u kojem se postavlja pitanje želi li se koristiti **video output** kao tekstura, što potvrdimo pritiskom na *checkbox* i gumb **OK**). Kao što možete zaključiti, u navedenom **media playeru** potrebno je postaviti određeni medij (prije toga, potrebno je odabratи opciju **Loop** (unutar **Playback** dijela) te unutar **media playera** postavimo željeni video koji će se vrtjeti unutar petlje). Kreira se novi element koji se zove **MP>LoadingLoop_Video**. Također, kreiran je materijal naziva **MT>LoadingLoop** čija se tekstura neće vidjeti ako **media player** nije upaljen. Pošto smo spomenuli teksturu, vrijedi napomenuti da se kao tekstura u materijalu postavlja video MP4 formata. Kad ga upalimo, tada ćemo vidjeti kako materijal izgleda. Što se tiče navedenog materijala, MP4 format sam po sebi nema podatke o tome da bude transparentan pa smo mi „zavarali“ *editor* tako što smo pozadinu postavili da bude crna i onda smo RGB parametar povezali s **Opacity** parametrom, drugim riječima, tamo gdje ima boje, neka to ima *opacity* (vidljivost), a ono što je crno se neće vidjeti. Također smo materijalu stavili da ima odsjaj (**Emissive Color**). Navedeni materijal je bitan jer u **LoadingScreen** komponenti, u odjeljku **Materials**, moramo pridružiti neki materijal. Bitno je još napomenuti da je **LoadingScreen** uvijek prisutan u igri, međutim, pokazujemo ga samo kada je parametar **Visibility** postavljen na vrijednost **True**. Dakle, referenca na njega postoji cijelo vrijeme (koristimo apsolutnu adresu do njega u projektu) jer se on učita prilikom pokretanja igre, ali ga nigdje ne uništavamo. Za kraj, važno je napomenuti da se u *Level1 blueprintu* na **BeginPlay** pokreće funkcija **Open Source** koja u varijablu **LoadingLoop**, koja je tipa **media player** i kojoj je defaultna vrijednost postavljena na **MP>LoadingLoop**, postavlja **File Media Source** komponentu naziva **MS>LoadingLoop**.

Nakon što smo objasnili kako funkcioniра **LoadingScreen** i **LoadingText**, vratimo se na glavni dio. Stali smo kod funkcija **SetVisibility** gdje postavljamo vidljivost **LoadingScreena** i **LoadingTexta** na vrijednost **True**. S druge pak strane, vidljivost **UI_Scorea** (ispis trenutnog *scorea* korisniku) postavlja se na vrijednost **False**, tj. neće se vidjeti. Nakon toga slijede funkcije **SetSoundMixClassOverride** u kojima nadjačavamo zvuk, tj. ako smo imali HP manji od zadane vrijednosti, tada smo jače čuli otkucaj srca negoli glazbu unutar igre, a ovim funkcijama vraćamo zvuk na staro. Nasjavljajući dalje dolazimo do postavljanja opcije **Blend Weight** koju posta-

vimo na vrijednost nula (prisjetimo se poglavlja o vizualnim efektima, gdje smo rekli da se **Blend Weight** postavlja kako bismo zacrnili ekran). Funkcijom **StartCameraFade** se vraćamo iz alfa vrijednosti jedan u alfa vrijednost nula. Nakon što smo došli na postavljenu poziciju, uzimamo sve actore klase **Enemy** pri čemu provjeravamo vraća li funkcija **GetAllActorsOfClass** barem jedan rezultat, onda prodjemo kroz to i uništavamo *actora*, a ako ne, onda ignoriramo i idemo dalje. Ova provjera je potrebna jer ako restartamo igru, a nije stvoren niti jedan neprijatelj, **ForEach** petlja neće znati što da radi jer je u njoj funkcija **DestroyActor** koja ne može tada uništiti nikoga.

Nakon navedenog, opet se pokreće funkcija **StartCameraFade** (nećemo vidjeti ništa na ekranu), ovisno o mapi na kojoj se nalazimo postavit ćemo se na neku poziciju, ponovno pokreće **StartCameraFade** tako da vratimo sliku i nakon odgode od jedne sekunde pokreću se brojač za funkcije **HealthRegeneration** i **IsPlayerRecentlyDamaged**. Varijablu **IsDead** postavimo na vrijednost **False**, čime dajemo do znanja da više nismo mrtvi. Vrijednost **Regen** variable postavimo na **False** što znači da je se može opet ispitivati. Varijablu **SecondCount** postavimo na nulu, a varijablu **ScoreCount** na minus jedan zato što funkcija **UpdateScore** (naziv je malo zbunjujuć jer je funkcija trenutno u UI-ju, međutim prvotno je bila zamišljeno da je funkcija u printu) automatski povećava vrijednost za jedan pa se varijabla **ScoreCount** zbog toga postavlja na minus jedan. Navedena funkcija se pokreće kada uzmemo **MotionControllerPawn**. Logika iza ove funkcije jest sljedeća: uzimamo vrijednost zapisanu u **ScoreCounter**, povećavamo je za jedan, i postavljamo novu vrijednost u navedenu varijablu. Nakon toga, dobivenu vrijednost iz integer oblika pretvaramo u tekstni i postavljamo taj tekst u element **UI_Score**. Nadalje, vidljivost **LoadingScreen** i **LoadingText** komponenti se ponovno postavi na vrijednost **False**, dok se vrijednost vidljivosti **UIScore-a** postavi na **True**.





17 UVOD U IGRU

Uvod u igru uvijek se događa kad igrač uđe u igru. Takvo postavljanje u igru daje glatki prijelaz i daje igraču do znanja da je dio igre.

Što se tiče uvoda u igru (eng. *intro*), logika započinje na **BeginPlay** postavljanjem varijable **Intro** na vrijednost **True**. Postavljanjem ove varijable, zabranjuje se micanje, teleportacija, kao i rotacija (postavljanjem varijable **CanPlayerMoveOrRotate** na vrijednost **false**). Nakon odgode od tri sekunde (vrijeme koje je potrebno da se pripremimo za igranje igre, moglo je biti i duže, ali smatramo da su tri sekunde optimalne), stvori se novi *blueprint* naziv **Unreal Engine Logo**. Nastavno na prethodno poglavlje, također postoji plane s materijalom, ali u ovom djelu se poziva stvaranje

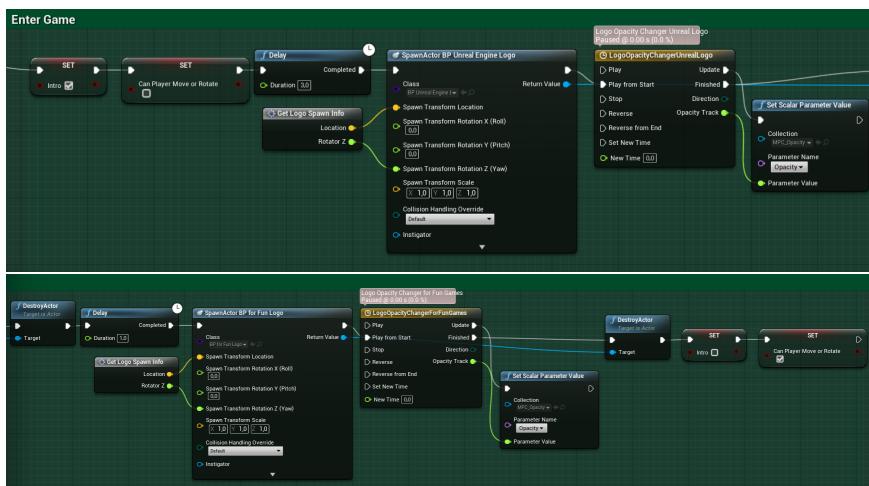


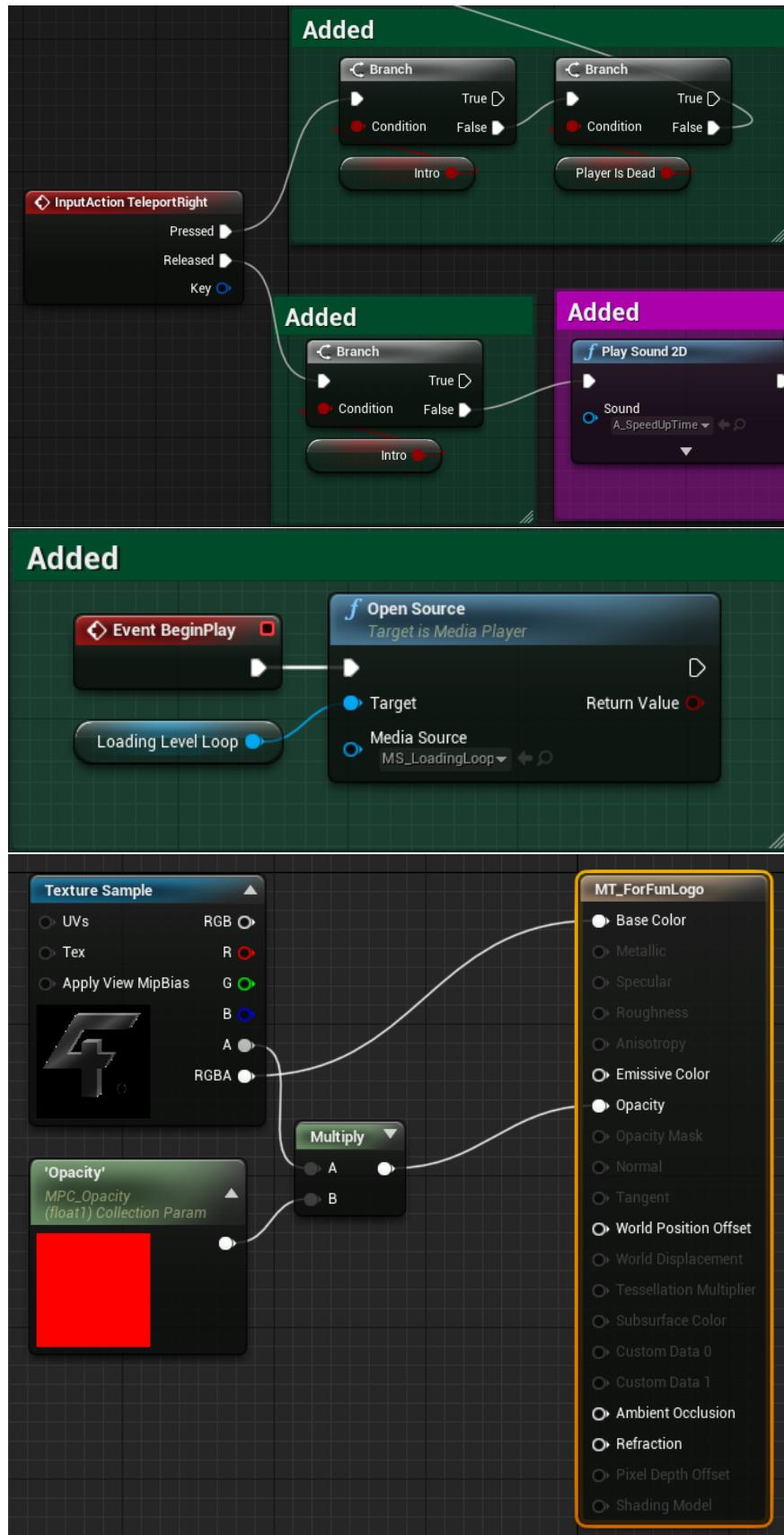
actora, dok smo u prethodnom poglavlju, kod *loadinga*, samo promijenili vidljivost. Međutim, i ovdje nailazimo na problem s *opacity* svojstvom jer ga ne možemo ručno slagati. Prije nego što dođemo do tog dijela, pozabavit ćemo se ubacivanjem slika za uvod.

Ubacili smo dvije slike (koje moraju biti u PNG formatu) koje možemo pronaći kao *texture* element pod nazivom TE_FunLogo i TE_UnrealEngineLogo. Također, vrijedi napomenuti da ubaćena slika bude RGB ili RGBA jer ako je nešto drugo (npr. *grayscale*) onda se neće dobro učitati u Unreal Engine. Isto tako, bilo bi dobro da slika ima i *Alpha* kanal za lakši rad sa slikom, iako to nije nužno. Nakon toga, navedene *texture* elemente stavimo na materijal. Kada otvorimo materijal, vidjet ćemo da parametar RGBA je **Base Color**, a *alfa* (*A*) vrijednost se množi postavljenim *opacityjem*. No, prije toga potrebno je dodati element **Material Parameter Collection** koji je nazvan MPC_Opacity. Otvaranjem navedenog elementa, prvo što ćemo vidjeti su skalarni (*float*) ili vektorski (trodimenzionalni) parametri. Tu ćemo postaviti jedan skalar koji ima naziv *Opacity* s vrijednošću jedan (zapisano u *float* obliku: 1.0). Nakon što se to kreira, *drag&dropom* ubacimo *MPC_Opacity* u materijale koje koristimo prilikom izrade uvoda. Drugi način je da se otvori materijal i u tražilicu upiše pojam **CollectionParameter** i zatim s lijeve strane, u odjeljku *General*, pod opcijom *Collection* izabere željena kolekcija i varijabla. Navedeni postupak nam

je bitan jer opacity ne možemo mijenjati dinamički, ali dinamički možemo mijenjati dodani parametar. Naime, mi želimo koristiti *timeline* pa nam je to važno.

Dakle, stvorimo actora s **GetSpawnLogoInfo** (uzme *X* i *Y* lokaciju izbornika te *Z* visinu i rotaciju po *X* i *Y* rotiranu za 90 kamere, što se vraća kao *output*) od kojeg dobivamo rotaciju i lokaciju te navedeno pridružimo s funkcijom koja stvara *actora*. Nakon toga koristimo timeline **LogoOpacityChanger**. U njemu je konfiguirirano da se u prvih 0.25 sekundi *opacity* postavi na vrijednost jedan te se nakon 2.5 sekunde vrijednost smanjuje prema nuli sve dok ne dođe do treće sekunde. Dakle, navedeni *timeline* traje tri sekunde. Zatim se pokreće funkcija **SetScalarParameterValue** u kojoj se postavlja kolekcija **MPC_Opacity**, parametar koji postavljamo se zove **Opacity** te iz timelinea prosljeđujemo **Opacity Track** parametru funkcije koji se zove **Parameter Value** (bitno je za napomenuti da se parametar ažurira svakim tickom). Nakon što je to gotovo, uništava se *actor*, čekamo jednu sekundu, na isti način ponavljamo postupak za drugi materijal (pri čemu je važno naglasiti da se ne mogu koristiti timelineovi istog naziva, dakle timeline koji smo koristili za prvi materijal (**LogoOpacityChangerUnrealLogo**) nije moguće koristiti kod drugog materijala, već se navedeni timeline treba preimenovati u **LogoOpacityChangerForFunGames**), pošto je uvod završio postavljamo vrijednost varijable **Intro** na **False** te dozvoljavamo kretanje i rotaciju igrača (postavljanjem varijable **CanPlayerMoveOrRotate** na vrijednost **true**), kao i sve druge funkcionalnosti koje su onemogućene prilikom pokretanja uvoda. Za kraj ćemo navesti gdje se točno provjeravaju **Intro** varijable: u **MotionControllerPawn Event** grafu (gdje se dohvaća vrijednost varijable prilikom pokretanja **InputAction TeleportLeft**, **InputAction TeleportRight** i **OpenMenu** događaja (prilikom pritiska ili otpuštanja tipke koja je povezana uz navedeni input action)). Ovime smo završili dio vezan uz uvod.





18 KUHANJE, PEČENJE I ZAKLJUČAK

Naravno, kuhanje i pečenje se ne odnosi na kulinarske vještine, nego na načine *deployanja* aplikacije. Zanimljivo je što se često programeri programskih jezika odluče za neortodoksne izraze za takve pojmove, npr. u Pythonu je analogni pojam **freeze**.

Prvi korak u procesu kuhanja je postavljanje prepostavljenje razine na Level1. u Project → Maps & Modes te zakvačiti opciju *Start in VR*.

Pečenje počinje tako da se u Window → Project Launcher. Tamo se dodaje novi profil za *baking*. Potrebno je i promijeniti i *Building Configuration* na *Shipping*.

Kuha se *by the book* i u tim opcijama se odabere Windows. Malo niže je potrebno i napraviti *release* te upisati ispravnu verziju nove aplikacije.

Pred kraj opcija *Cooker build options* je potrebno također označiti kao *Shipping*.

Nakon što je sve to gotovo potrebno je izabrati mjesto na disku gdje će se stvoriti instalacijske datoteke. Pokreće se kreirani profil i pričeka se da sve završi.

Nakon što je igra dovršena, potrebno ju je monetizirati. Najbolji način za monetizaciju igre je pomoću poznatih digitalnih distribucijskih platformi za igre poput Steama. U ovom dijelu bit će opisan proces releaseanja igre na Steam, Epic Games Store i Oculus Store.

18.1 STEAM

Prva stvar u procesu releasanja igre na Steam je postanak Steamworks partnerom. Kako bi se osoba registrirala na Steamworks potrebno je otici na stranicu Steam Direct (<https://partner.steamgames.com/steamdirect>) i slijediti navedene korake. Prvi korak je ispunjavanje informacija o bankovnom računu i poreznom statusu.

Nakon ispunjavanja svih potrebnih informacija potrebno je uplatiti 100 USD prije releasa prve igre. Uplaćeni iznos može se vratiti ako Steam od igre zaradi barem 1 000 USD preko prodanih jedinica i in-app purchasea (unutar-aplikacijskih kupovina?).

Nakon što su ispunjene sve informacije, uplaćeni novci i nakon što je korisnik potvrdio svoj identitet, može se započeti s procesom postavljanja igre na Steam. Prva stvar koju je potrebno napraviti ako je ovo prvi put da se igra stavlja na Steam jest izrada Store Pagea. Store Page je stranica koju ima svaki izdavač koji se nalazi na Steam. Na Store Pageu se mogu vidjeti sve igre koje je izdao određeni izdavač. Prvi korak u izradi Store Page je

Street Address Street address must be the current location of the individual or company entered in the Legal Name field above, and will be used for any appropriate notifications.

City

State/Province **Postal Code**

Country **United States**

Payment Notification Email Set the email where you would like payment notification emails sent. This could be your own email or your company's finance email alias for example. This can be changed in the future by editing your company details.

Preferred Payment Email Language: **English** This will be the preferred language we send payment notification emails in.

Fax Number If you provide a fax number, we may give you written notices via fax.

Continue

Legal Name
This is really, really important to enter correctly. Carefully read all instructions below. You will be unable to release your product via Steam until this name matches all records.
The name you enter below must be the legal entity that owns or has rights to publish the game, software or video ("content") and is the legal entity that will be signing the Steam Distribution Agreement. The legal name you enter here must match the name as written on official documents with your bank and on United States IRS tax documents or foreign tax documents if applicable. You will need to enter this name again as your bank account holder and the legal name associated with a tax payer identification number in the following steps.
If you don't have a company name and you are the sole owner of your content, please fill in your full name as the Legal Name and your own address as Street Address. If you co-own the content with other individuals, you must form a legal entity to own and receive payments for your content.
The Legal Name here is for internal use. If you have a DBA or "friendly name" that you wish to show to customers on your store page, you will be able to enter that separately when creating your store page.

Legal Name (As written on bank documents)

Company Form
Company form must match your entity formation documents. An example of what to enter in this box are: "A Quebec limited liability partnership" or "A Washington State corporation" or "A Sole Proprietorship". If you own the content as an individual, indicate "Sole Proprietorship".
Note: We are unable to work with partnerships that exists outside the US, if that partnership is taxed at the individual partner(s) versus the partnership level.
If you have a partnership registered in the United States or if your partnership exists outside the US and the partnership is taxed at the partnership level, then we can support your partnership. However, due to the complexity of obtaining proper tax documentation we cannot enter into a Steam Distribution Agreement with partnerships that exist outside the US, if that partnership is taxed at the individual partner(s) versus the partnership level.

Company Form

Slika 121: Informacije o Steamworksu

izrada Community Grupe u kojoj će se nalaziti ime izdavača i logo. Nakon toga se ide na opciju Creator Homepage Setup gdje se izrađuje Store Page. Prilikom setupa, Store Page se asocira s izrađenom Community grupom. Nakon izrade Store Page se može urediti po želji.

Nakon izrade Store Page potrebno je vratiti se na Steamworks stranicu i krenuti s postupkom stavljanja igre na Steam. Prvo što je potrebno napraviti jest slanje igre na Steam pomoću Steampipe. Pritom je potrebno odabratи platforme na kojima će se igra moći igrati i jezike na kojima će igra biti dostupna. Tijekom postavljanja igre za slanje dobit će se app ID koji će jednoznačno označavati aplikaciju.

Poslije slanja igre na Steam potrebno je odrediti informacije koje su bitne za releasanje igre. Informacije se dijele na Sore, Community i Depots. Informacije o Depotsima bi trebale biti dovršenje slanjem igre dok je ostale tek potrebno izvršiti. Store informacije su informacije poput bitne za kupce. To su osnovne informacije o igri, datum izlask, „system requirements“, cijena igre, trailer, screenshoti igre, podaci o developeru i publisheru. Što se tiče Community dijela, potrebno je samo dodati tri slike veličine 184x69, i dvije slike veličine 32x32. Jedna manja slika je za ikonu koja se koristi za pokretanje igre preko desktop-a, jedna je slika koja predstavlja igru na

wishlistama i jednu će korisnici Steama moći vidjeti svaki put kad dođu na stranicu posvećenu toj igri.

Postavljanjem svih informacija navedenih u prošlom poglavljiju, igra je spremna za review proces prilikom kojeg će zaposlenici Steama provjeriti da je igra sigurna i da je inače sve u redu s njom. Proces provjere obično traje od jednog do pet dana. U slučaju da je igra koja se releasea prva, potrebno je pričekati do 30 dana kako bi se mogli provjeriti bankovni i porezni podaci izdavača. Također, Store page i Coming soon stranicu za igru potrebno je postaviti najmanje dva tjedna prije releasea igre za kupovinu. Kada prođu ti periodi igru se može releasati na Steam.

18.2 EPIC GAMES STORE

Releasanje igre na Epic Games Store ima svoje prednosti i nedostatke u odnosu na Steam. Najveća prednost je činjenica da Steam uzima 30% prihoda od prodaje igri i in-app purchasea + još 5% ako je igra izrađena pomoću Unreal Enginea, dok Epic Games Store uzima samo 12%. Glavni nedostatak Epic Games Storea je činjenica da postoji nešto veća mogućnost odbijanja postavljanja igre na tu platformu.

Kako bi izdavač postavio svoju igru na Epic Games Store potrebno je ispuniti obrazac na stranici <https://www.epicgames.com/store/en-US/about>.

Nakon što se ispune sva potrebna polja, obrazac se šalje na razmatranje. Nakon što završi razmatranje izdavač saznaće hoće li se naći mjesto za njegovu igru u Epic Games Storeu ili ne. Ako se pronađe mjesto izdavač će objaviti igru uz pomoć zaposlenika Epic Games Storea. Ako se ne dozvoli izdavanje igre onda je to kraj priče i igra neće biti izdana na tu platformu.

Dakle, za razliku od Steama koji garantira postavljanje igre na store (u slučaju da je igra sigurna), ne postoji garancija da će se igra objaviti na Epic Games Store. Naravno pri odluci o izdavanju igre treba uzeti u obzir i činjenicu da Steam naplaćuje izdavanje prve igre, a Epic ne, kao i činjenicu da Epic uzima manje provizije od svake transakcije. Osim toga, na Epic je lakše postaviti igru gledajući tehničku stranu, jer setup za postavljanje igre na Steam potrebno odraditi samostalno, dok će postavljanje igre na Epic biti odrđeno uz suradnju s Epic Games Storeom.

18.3 OCULUS STORE

Oculus Store je platforma na kojoj se nalaze isključivo VR igre. Prema tome, upravo je ta platforma sjajna za release takvih igri. Kao i za Epic Games Store, Oculus Store zahtjeva detaljno pregledavanje igre i ispunjavanje mnogih uvjeta kako bi se igra uopće prodavala na platformi.

Store Submission

To request your game be added to the Epic Games store please complete the following form.

Email *

First Name *

Last Name *

Company *

Company Website *

Game Description *

Video Link *

Position / Title

Planned Ship Date *

dd.mm.gggg.

Current Engine *

Current Engine

Country/Region *

SUBMIT

The form consists of several input fields: Email, First Name, Last Name, Company, Company Website, Game Description, Video Link, Position / Title, Planned Ship Date (dd.mm.gggg.), Current Engine (dropdown menu), and Country/Region (dropdown menu). A large blue 'SUBMIT' button is located at the bottom of the form.

Slika 122: Epic games submission

Oculus Store prvo provjerava ispunjava li igra određene smjernice vezane za dozvoljeni sadržaj u igrama. Te su smjernice provjeravaju imaju li u igri govora mržnje, glorificiranja terorizma, pornografije i slično. Nakon toga se gleda ispunjava li igra tehničke zahtjeve. Tehnički zahtjevi koje mora ispuniti su ispravan format slika i drugog artworka, mogućnost slušanja

audia pomoću oculusa, kompatibilnost s nekim aplikacijama, ispravan rad, način na koji moraju biti postavljeni gumbi, minimalne performanse, mora biti sigurna i mora se moći igrati na određene načine (bez zvuka, podrška za daltonizam i slično). Osim toga, ako se koriste podaci korisnika, mora se točno odrediti koji se podaci koriste.

Valja napomenuti da je prilikom izrade slika bitno da one budu određene rezolucije, i png formatu i da se jasno može vidjeti tekst.

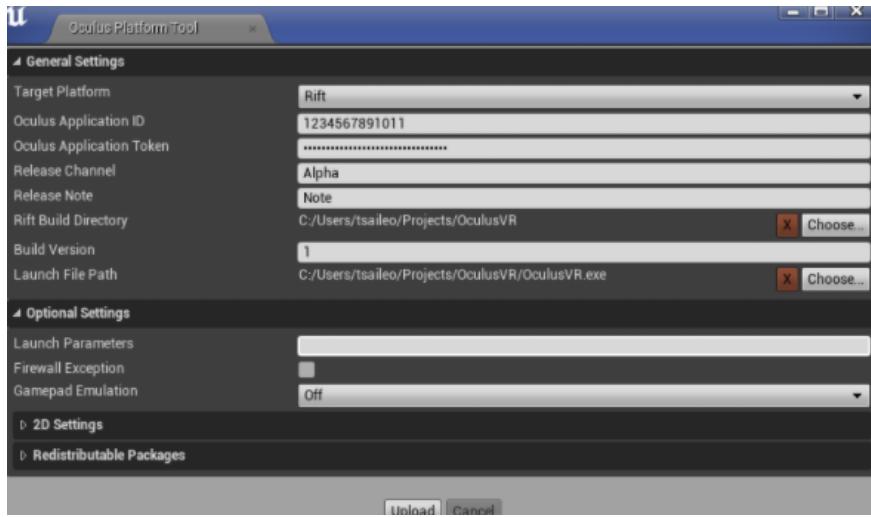
Store Assets Checklist		
Download templates here: ocul.us/storetemplates		
Asset	Dimensions	Pixels
■ Hero	10:3	3000 x 900px
■ Landscape	16:9	2560 x 1440px
■ Square	1:1	1440 x 1440px
■ Portrait	7:10	1008 x 1440px
■ Mini	3:1	1080 x 360px
■ Logo (Transparent)		9000max x 1440max
■ 5 Screenshots	16:9	2560 x 1440px
■ Trailer Video		Min: 1080p ~ Max: 2k (MP4/H.264/AAC format)
■ Trailer Cover	16:9	2560 x 1440px
■ Mobile Icon	1:1	512 x 512px Mobile-only (Gear / Go)
■ PC Icon	1:1	256 x 256px One .ico file containing 6 sizes. 96 x 96px 64 x 64px 48 x 48px 32 x 32px 16 x 16px
■ Cubemap		12,000 x 2000px (Mobile only - Optional)

Slika 123: Oculus format

Kako bi se mogao vidjeti tekst on mora biti centriran i mora biti dovoljan razmak od teksta do ruba slike. Od poslanih slika 5 treba biti screenshota igre na kojima ne bi smjelo biti nikakvog teksta. Trailer za igru može biti od 30 sekundi do 2 minute dugačak, mora biti najmanje full HD, a najviše 4K, a logo koji nije oculusov smije biti samo na njegovom početku.

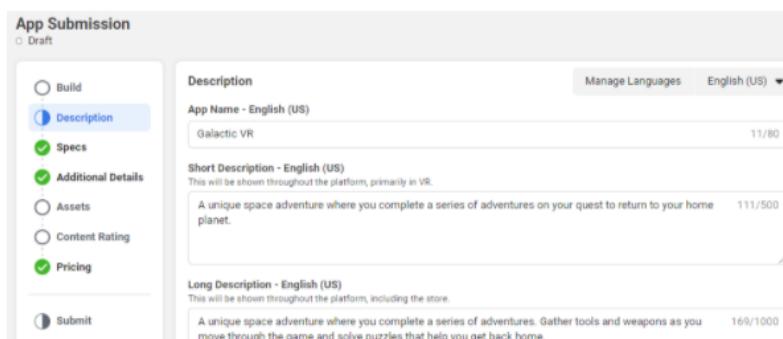
Tek pošto su ispunjeni svi zahtjevi ima smisla prijaviti igru. Prvi korak u relesanju igre je stvaranje app pagea Oculus Storeu. Nakon toga je potrebno poslati igru u produkcijски kanal što se, između ostalog, može napraviti

putem Unreal Enginea koristeći Oculus Platform Tool u kojem se samo trebaju ispuniti svi podaci.



Slika 124: Oculus upload

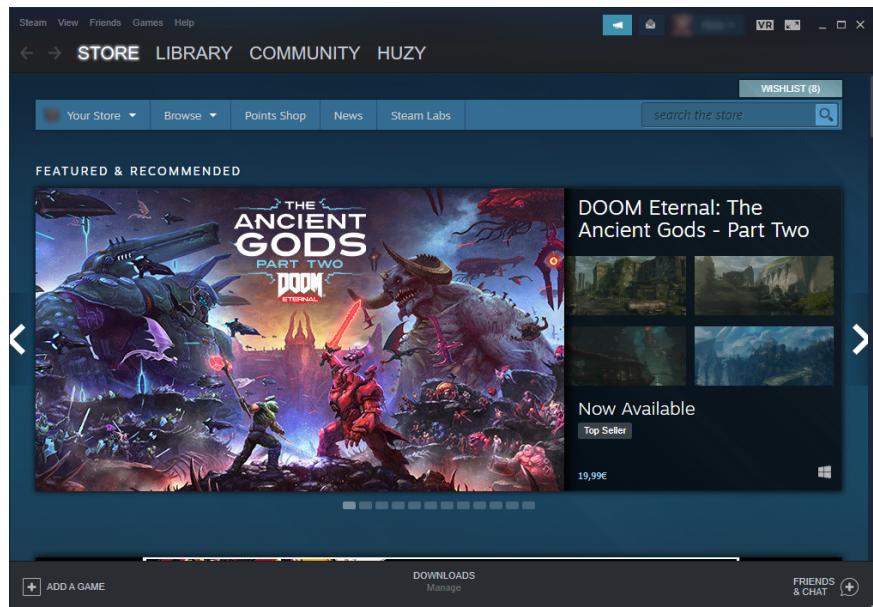
Nakon toga je potrebno ispuniti potrebne podatke. To se radi na način da se otvori App Submission u navigaciji te se ispunii sve što je potrebno. Nakon toga se obrazac šalje i čeka se na odgovor Oculusa.



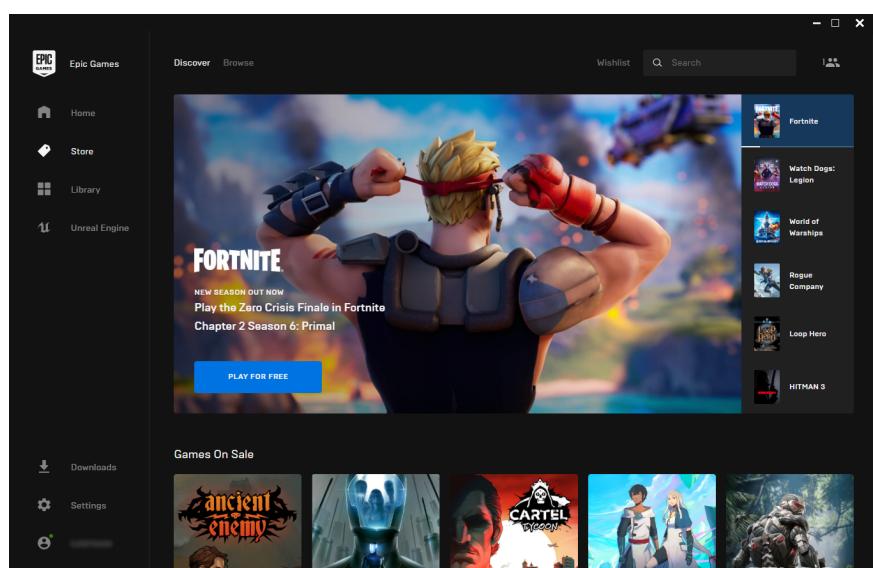
Slika 125: Oculus Send

U slučaju da se dobije potvrda igra se može relesati na Oculus Store s tim da kao i Steam Oculus Store uzima 30% priliko svake transakcije.

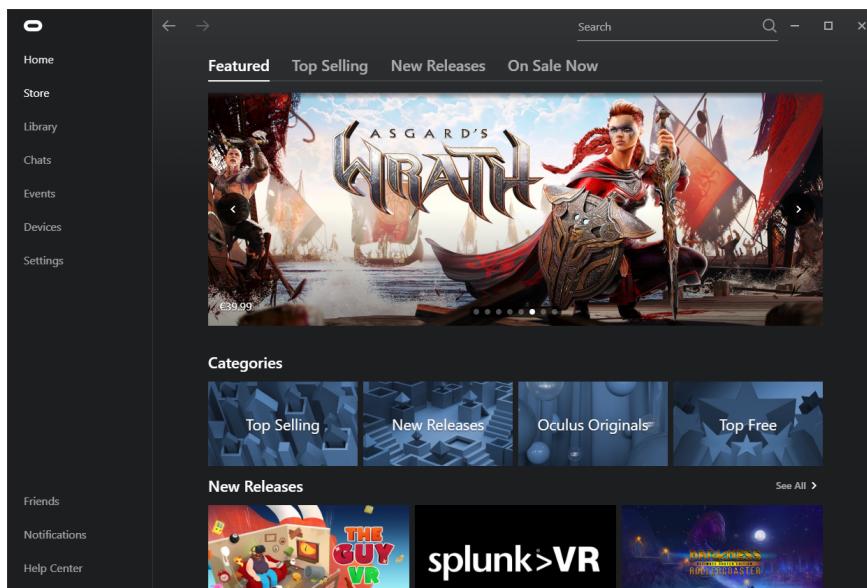
18.4 PRIKAZ SVIH SPOMENUTIH TRGOVINA



Slika 126: Prikaz Steam storea



Slika 127: Prikaz Epic storea



Slika 128: Prikaz Oculus storea

KAZALO

actor, 12, 20, 56
animacija šake, 40
Apex Destruction, 79

bacanje kamena, 78
BP_MotionController, 26
brisanje, 114

checkbox, 111
compile, 47
content browser, 12
Controller, 18

damage, 96
dodavanje actora, 56
događaj, 17

event, 17

fade in, 21
fade out, 19
force feedback, 27

git, 16
gljivice, 21, 23
Google Cardboard, 18
gravitacija, 30
grip, 28
gumb, 111

Head Mounted Display, 18
health, 96, 100
HMD, 18
HTC Vive, 18
hvatanje, 28

indikator mesta teleportacije, 20, 25
inside-out praćenje, 25
instalacija, 10
izračun luka, 36

kamen, 78
kompajliranje, 47
kontroler, 22

kretanje, 17
luk, 29, 36
light-house praćenje, 24, 41
line of sight, 90
lokacijska osviještenost kontrolera, 24

mehanike, 9
mrak na oči, 19

navigacijski sustav, 31
nevidljiv zid, 20

Oculus Rift, 18
odredište teleportacije, 32
okoštavanje, 28
opcije kretanja, 112

pawn, 20
percepcija, 86
place actors, 12
PlayAreaBounds, 26
pod, 33
postavke, 17, 111
preporuka broja FPS-a, 29
PSVR, 17, 18, 21
putanja projektila, 30

raspad kamena, 78
regeneracija, 100
rigging, 28
room-scale, 25
rotacija mesta teleportacije, 23
rotacija zapešća, 41
ruka, 26
rukovanje kontrolerom, 22
RumbleController, 27, 35

Samsung Gear VR, 18
SaveGame, 113
skaliranje, 26
spline, 29, 37
spremanje, 113

spremanje postavki, 111
stablo ponašanja, 85
sustav zaštite, 40

tangenta, 39
teleport cylinder, 26
teleportacija, 17, 19, 22, 30
traženje poda, 33
trening mod, 112

umjetna inteligencija, 84
uništavanje, 114
Uvod, 7
uvodenje materijala, 78

Valve Index, 18

varijsable, 47
vibracije, 27, 35
vidokrug neprijatelja, 90
viewport, 12
visina, 17
vizualizacija luka, 29
VR, 10

walk speed, 90
world outliner, 12
WorldStatic, 30

zdravlje, 96, 100
zrcalna slika, 26

šaka, 40