Konstrukcija video igara u programskom paketu Unreal Engine

Terzanović, Mateja

Master's thesis / Diplomski rad

2023

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:217:944605

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-07-11**

Repository / Repozitorij:

Repository of the Faculty of Science - University of Zagreb

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Mateja Terzanović

KONSTRUKCIJA VIDEO IGARA U PROGRAMSKOM PAKETU UNREAL ENGINE

Diplomski rad

Voditelj rada:
prof. dr. sc. Mladen Jurak

Zagreb, veljača, 2023.

Ovaj diplomski rad obranjen je dana ____________________ pred ispitnim povjerenstvom u sastavu:

1. ____________________________, predsjednik
2. ____________________________, član
3. ____________________________, član

Povjerenstvo je rad ocijenilo ocjenom _______________.

Potpisi članova povjerenstva:

1. ____________________________
2. ____________________________
3. ____________________________

Mami, tati i bratu hvala na ljubavi, podršci i strpljenju.
Filipu hvala na ustrajnom ohrabrivanju, na brizi i nadi u svijetlu budućnost.

Sadržaj

Sadržaj **iv**

Uvod **1**

1 Upravitelj igre **3**
1.1 Video igre . . . 3
1.2 Što je upravitelj igre? . . . 4
1.3 Kratka povijest upravitelja igara . . . 4
1.4 Arhitektura upravitelja igre . . . 5

2 Oblikovni obrasci u razvoju video igara **17**
2.1 Objektno orijentirano programiranje . . . 17
2.2 *Game Loop* . . . 20
2.3 *Update method* . . . 23
2.4 *Component* . . . 24
2.5 *Observer Pattern* . . . 26
2.6 *Factory Method Pattern* i *Abstract Factory* . . . 27
2.7 *Prototype* . . . 30

3 Unreal Engine **33**
3.1 Uvod . . . 33
3.2 *Unreal Engine 5* . . . 33
3.3 *Blueprint* i *C++* . . . 37
3.4 Korištenje *Blueprint*-a u *Unreal Engine*-u . . . 39
3.5 Korištenje *C++*-a u *Unreal Engine*-u . . . 42

4 *Survival Wizardry* **47**
4.1 Uvod . . . 47
4.2 *Game Mode* . . . 47
4.3 *Player Controller* . . . 51

4.4 *Wizard* 53
4.5 *Health* 55
4.6 *Staff* 57
4.7 *Spells* 57
4.8 *Projectiles* 57
4.9 *Enemies* 59
4.10 *Experience Gem* 61
4.11 *Health Pack* 61
4.12 *Spawner* 62
4.13 Korisničko sučelje 63

Zaključak **67**

Bibliografija **69**

Uvod

Igre postoje oduvijek. Od davnina ljudi, pa i životinje, zaokupljali su se raznim igrama, a današnje su igre samo novi oblik ove drevne vrste socijalne interakcije. Većina nas ima dobru intuiciju o tome da li se nešto može smatrati igrom ili ne. Danas postoje razne vrste igara – društvene igre, kartaške igre, razne igre koje smo kao djeca igrali na ulici poput igre skrivača, lovice ili igra policajaca i lopova te nešto novija pojava u društvu - video igre.

Brojni stručnjaci pokušali su dati formalnu definiciju igre, no one su se ispostavile suviše komplicirane te često ne obuhvaćaju čitav aspekt igara koje danas poznajemo. Zanimljiva je definicija Bernarda Suitsa koji kaže da je igranje igre dobrovoljni trud za svladavanje nepotrebnih prepreka. Raph Koster u [28] igru opisuje kao interaktivno iskustvo koje igraču pruža niz uzoraka koji progresivno postaje sve teži te kojeg igrač treba naučiti svladati. Koster smatra da su učenje i usavršavanje vještina suština zabave u igrama.

Oko igara se razvila i cijela grana matematike, *teorija igara*, koja proučava strateške situacije u igrama s dobro definiranim skupom pravila u kojima uspjeh jednog racionalnog igrača, koji želi maksimizirati svoju korist, ovisi o odlukama drugih racionalnih igrača.

Vidimo da postoje razne definicije, tipovi i aspekti proučavanja igara. Ovaj rad bavi se video igrama, čiji je opis dan u prvom poglavlju. Točnije, fokus rada je razvoj video igara korištenjem *Unreal Engine*-a koji spada pod vrstu softvera koje nazivamo upravitelji igara. Definicija upravitelja igara i kratak pregled njegovih komponenti nalazi se u drugom poglavlju. Zatim se u trećem poglavlju objašnjavaju osnovni koncepti objektno orijentiranog programiranja, a u četvrtom poglavlju se opisuju oblikovni obrasci najčešće korišteni u razvoju video igara. U petom poglavlju opisan je *Unreal Engine*, njegovi osnovni elementi te specifičnosti razvoja video igara u *Unreal Engine*-u korištenjem jezika *C++* i vizualnog skriptnog jezika pod nazivom *Blueprint*. Šesto i posljednje poglavlje opisuje video igru napravljenu kao praktični dio ovog rada.

Poglavlje 1

Upravitelj igre

1.1 Video igre

Video igra je igra koja koristi ulazne informacije dobivene od korisnika preko nekog ulaznog uređaja poput tipkovnice, miša ili kontrolera kako bi generirala vizualnu povratnu informaciju na nekoj vrsti ekrana. Kako se svakodnevno objavljuje sve više i više video igara, s gomilom novih žanrova te raznim inovacijama, brojne video igre počinju sve više odudarati od nekih dosadašnjih shvaćanja video igre. U definiciji video igara zna se nalaziti zahtjev da igra sadrži stanje pobjede ili poraza, ali neke novije video igre nemaju niti taj uvjet. Ipak, moglo bi se reći da sve igre zahtijevaju nekakvu vrstu interakcije s korisnikom, stoga to smatramo njihovom osnovnom karakteristikom, sve drugo je diskutabilno.

U [25], video igre se opisuju kao meke interaktivne računalne simulacije u stvarnom vremenu temeljene na agentima (eng. *soft real-time interactive agent-based computer simulations*). Kada govorimo o video igrama, najčešće zamišljamo nekakav 2D ili 3D virtualni svijet u kojem je glavni lik pod kontrolom igrača. Virtualni svijet je aproksimacija, tj. pojednostavljeni prikaz svijeta (pravog ili izmišljenog) postignuta njegovim matematičkim modeliranjem. Taj matematički model se može kompjuterski manipulirati i njega smatramo simulacijom svijeta. Prethodno smo već naglasili da su video igre interaktive, a kako se reakcije na događaje u igrama uglavnom odvijaju u stvarnom vremenu, kažemo da su video igre interaktivne simulacije u stvarnom vremenu. Nadalje, u većini igara imamo nekolicinu entiteta koji međusobno interaktiraju i njih nazivamo agentima, stoga kažemo da je igra bazirana na agentima. Preostaje još jedino objasniti što znači da su one meke simulacije. U svakom sustavu u stvarnom vremenu imamo razne rokove, primjerice, ekran se mora osvježiti određen broj puta u sekundi zbog fluidnosti, a simulacija fizike se mora osvježavati još i češće. Meki sustav u stvarnom vremenu je onaj u kojem neki propušteni rok nema katastrofalne posljedice. Video igre zasigurno spadaju pod meke sustave jer propušteni rok kod video igara nema nikakve značajne posljedice na igrača.

1.2 Što je upravitelj igre?

Prema [2], upravitelj igre (eng. *game engine*) je softver koji programerima pruža paket alata i značajki kako bi mogli profesionalnije i efikasnije razvijati video igre. Što je točno taj paket alata i od čega se sve sastoji jako varira od jednog upravitelja igara do drugog, a nepostojanje jasne granice između upravitelja igre i same video igre dodatno otežava njegovo egzaktno definiranje. Neke osnovne funkcionalnosti koje pruža upravitelj igre su: iscrtavanje grafike, detekcija kolizija, upravljanje fizikom, upravljanje zvukom, animacijama, verzioniranjem, upravljanje memorijom itd. Upravitelji igara često pružaju i paket alata za vizualni razvoj igara kroz integrirano razvojno okruženje (eng. *integrated development enviroment, IDE*) kako bi se osigurao još brži i jednostavniji razvoj video igara. Upravitelja igre dakle možemo smatrati svojevrsnom platformom koju možemo koristiti za razvoj video igara ili pak nekih drugih interaktivnih aplikacija.

1.3 Kratka povijest upravitelja igara

Kako bi bolje razumjeli upravitelje igara i njihovu svrhu, pogledajmo kratko kako je razvoj video igara izgledao kroz povijest. Prije pojave upravitelja igara video igre su se svaki put morale razvijati od nule. Trebale su se optimizirati i prilagoditi specifičnom hardveru za koji su bila pravljene, pogotovo dok su u pitanju bile arkadne igrice. No čak i kada je prikaz na ekran prestao biti problem, ostao je problem iznimno male memorije. Stoga nešto poput upravitelja igre, koji bi zahtijevao puno memorije za to vrijeme, nije bilo moguće. Igre su se dizajnirale pomoću čvrsto kodiranog (eng. *hard-coded*) skupa pravila i gotovo ništa se nije moglo prenijeti iz jedne igre u drugu.

Izraz upravitelj igre (eng. *game engine*) se počeo pojavljivati sredinom 90-ih s pojavom pucačina u prvom licu (eng. *first person shooter, FPS*), iako su i prije toga postojali sustavi koji bi se mogli smatrati upraviteljima igara, primjerice *Nintendo* je softver koji je koristio za trkaću igru *Excitebike* kanije koristio u razvoju popularnog platformera *Super Mario Bros* (1985.). Jedna začetnica spomenutog žanra video igara je i dobro poznati *Doom* tvrtke *id Software*. *Doom* je razvijen s poprilično jasnom granicom između osnovnih dijelova softvera poput prikazivanja grafike ili detekcije kolizija i dijelova koji se direktno tiču igračeva iskustva poput pravila igre ili izgleda komponenti u igri. Pomoću upravitelja igre video igre *Doom* (eng. *Doom engine*), koji je kasnije preimenovan u *id Tech 1*, *id Software* je razvio i brojne druge video igre. Prepoznata je vrijednost ovakvog odvajanja upravitelj igre od sadržaja igre pa su se igre počele razvijati upravo s ovim pristupom na umu. Jedna takva igra je i *Unreal* iz 1998. tvrtke *Epic Games* čime je nastala prva verzija revolucionarnog *Unreal Engine*-a. Tijekom godina razvijali su se brojni upravitelji igara, oni privatni koje koriste isključivo zaposlenici kompanija za razvoj svojih proizvoda, ali i

oni koji su se komercijalizirali i postali dostupni široj javnosti na korištenje, među kojima su svakako najpoznatiji *Unity* i *Unreal Engine*.

Većina upravitelja igara je pažljivo dizajnirana kako bi pokretala točno određenu vrstu igara na točno određenoj hardverskoj platformi. Čak i najopćenitiji upravitelji igara prilagođeni svim platformama nisu pogodni za razvoj igara u svim žanrovima. Razlog za to je što je za razvoj efikasnog softvera potrebno nešto žrtvovati kako bi nešto drugo radilo bolje, a takve odluke se baziraju na tome koja je namjena tog softvera. Primjerice upravitelj igre namijenjen za prikazivanje manjih zatvorenih prostora funkcionirat će znatno drugačije nego onaj namijenjen prikazivanju velikih otvorenih površina. S današnjim brzo rastućim razvojem hardvera i grafičkih kartica ova razlika između upravitelja igara namijenjenih određenim žanrovima sve je tanja.

Kako se tehnologija upravitelja igara sve više razvija, tako se širi i područje njegove primjene pa se tako osim u razvoju video igara upravitelji igara danas koriste i za razvoj raznih interaktivnih aplikacija poput arhitektonskih vizualizacija, marketinških demonstracija i simulacija koje se mogu koristiti za treniranje raznih vještina.

1.4 Arhitektura upravitelja igre

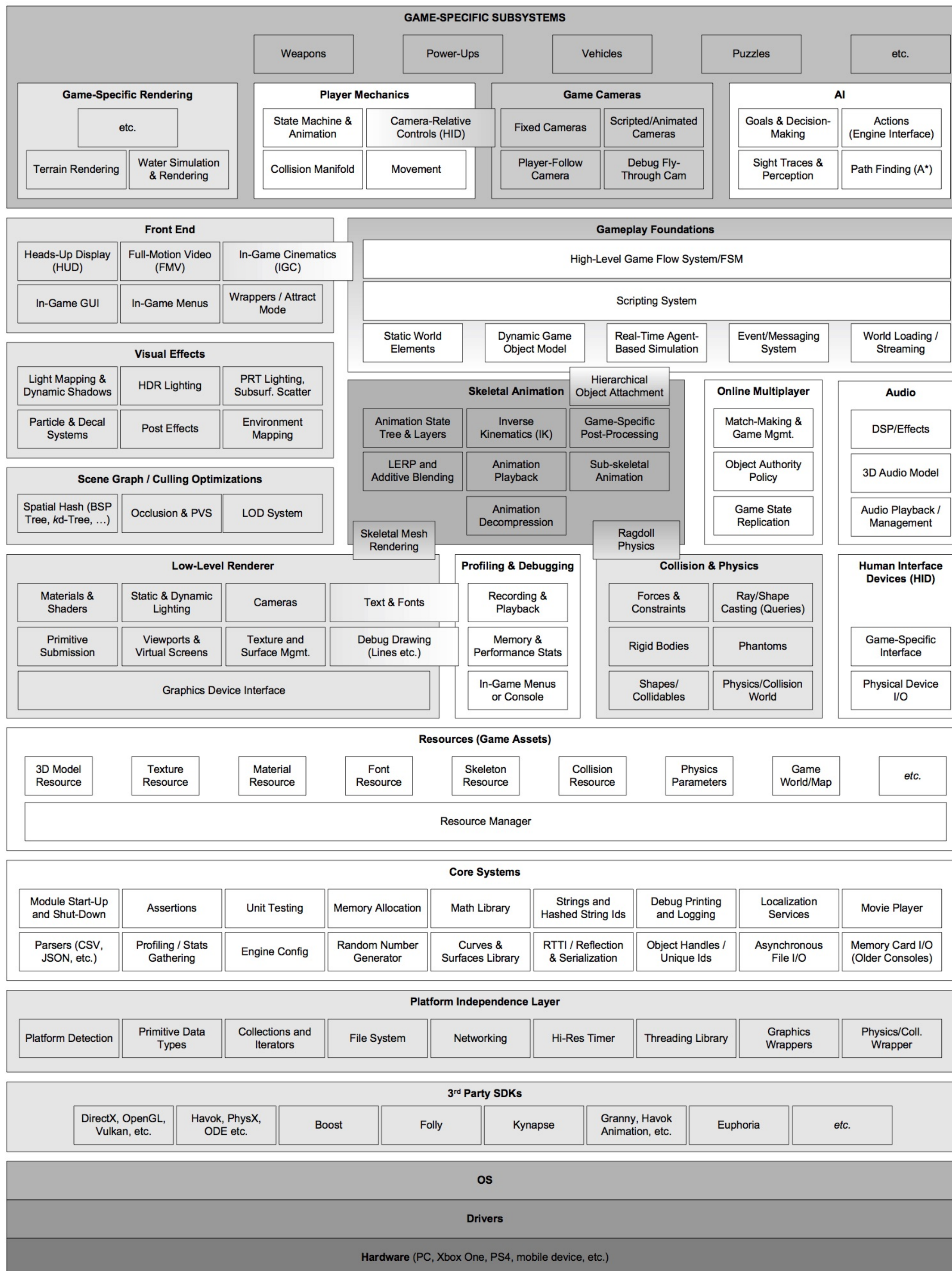
Pogledajmo na trenutak sliku 1.1 na kojoj je prikazana arhitektura upravitelja igre. Slika je poprilično velika i ako čitamo s A4 papira, teško je išta vidjeti. Slika ovdje stoji kako bi se stekao dojam o tome koliko su uistinu veliki sustavi koje ovdje nazivamo upravitelji igara. Ono što na slici možemo primijetiti jest da je sustav izgrađen u slojevima. Obično viši slojevi ovise o nižima, ali ne i obrnuto kako bi se izbjegla kružna ovisnost koja može uzrokovati nestabilnost u ovako velikim sustavima. Sada ćemo prema knjizi [25] dati pregled pojedinih komponenti od kojih se sastoji upravitelj igre.

1.4.1 Računalno sklopovlje

Kako bi uopće mogli pokretati i koristiti video igru, nužno je da imamo pristup nekom uređaju koji ju podržava, stoga je razumno da kao prvi i najniži sloj arhitekture upravitelja igara imamo računalno sklopovlje. Naziv računalno sklopovlje, odnosno hardver (eng. *hardware*), odnosi se na opipljivi dio računala. Sloj računalnog sklopovlja predstavlja ciljani računalni sustav ili konzolu na kojoj će se video igra pokretati, a to su primjerice stolno računalo, *PlayStation*, *Xbox*, pametni telefon itd.

1.4.2 Upravljački programi

Upravljački programi (eng. *drivers*) pružaju sučelje prema sklopovlju računala kako bi operacijskom sustavu i ostalim računalnim programima omogućili pristup funkcionalnos-



Slika 1.1: Arhitektura upravitelja igre (slika iz [25])

tima hardvera bez poznavanja svih pojedinosti o točno određenom hardveru koji se koristi. Oni dakle definiraju način komunikacije između operacijskog sustava i računalnog sklopovlja, upravljaju s hardverskim resursima i apstrahiraju operacijskom sustavu i ostalim gornjim slojevima arhitekture detalje hardverskih uređaja. Na različitim računalima koriste se različiti upravljački programi i zahvaljujući njima sklopovski različita računala s istim operacijskim sustavom se mogu koristiti na jednak način.

1.4.3 Operacijski sustavi

Operacijski sustav (OS) je skup osnovnih programa koji upravlja izvođenjem programa na računalu, omogućuje korisnicima i programima jednostavno korištenje sredstava računala kroz razna aplikacijska programska sučelja (eng. *application programming interface, API*) i kroz grafičko korisničko sučelje (eng. *graphical user interface, GUI*) te je on konstantno aktivan na računalu. Neki operacijski sustavi poput Microsoft Windowsa dijele procesorsko vrijeme između nekoliko programa koji se izvode na računalu, stoga igrica koja se koristi na takvim operacijskim sustavima nikada ne smije pretpostaviti da ima potpunu kontrolu nad hardverom. Što se tiče konzola, u početku nisu imale operacijski sustav sam po sebi, ali u novije vrijeme moderne konzole također uvode operacijske sustave čime se smanjuje razlika između razvoja igrica za konzole i osobna računala.

1.4.4 Paketi za razvoj softvera i međusoftveri trećih strana

Većina upravitelja igara poseže za raznim paketima za razvoj softvera (eng. *software development kits, SDKs*) i međusoftverima (eng. *middleware*) trećih strana, specijaliziranim za određene funkcionalnosti. Navest ćemo samo neke od njih. Kako igre dosta ovise o apstraktnim tipovima podataka i algoritmima za manipulaciju njima, upravitelji igara često koriste dodatne biblioteke koje im pomažu s rješavanjem zadataka i s performansama poput biblioteka *Boost*, *Folly* i *Loki*. Također koriste pakete za 3D grafiku kao što su *OpenGL*, *DirectX* i *Vulkan*, koje koriste za renderiranje grafike i komunikaciju s grafičkom procesorskom jedinicom. Nadalje, upravljanje fizikom i detekciju kolizija pružaju primjerice paketi *Havok*, *PhysX* i *Open Dynamics Engine*, a postoje i razni paketi za animaciju poput paketa *Granny*, *Havok Animations* ili *ObrisAnim*.

1.4.5 Sloj neovisnosti o platformi

Većina kompanija kada izdaje igricu želi da ona bude dostupna što široj publici, stoga je važno da se video igra može koristiti na različitim platformama. Iz ovog razloga, većina upravitelja igara sadrži sloj neovisnosti o platformi kojim se komponentama upravitelja apstrahiraju detalji specifične platforme. To se postiže takozvanim omotavanjem funkcija

(eng. *wrapping*) koje se razlikuju od platforme do platforme. Ovime dobivamo programsko sučelje koje je konzistentno na svim ciljanim platformama. Korisno je i omotavanje grafičkih ili fizičkih biblioteka, bile one ovisne o platformi ili ne, kako bi se omogućio kasniji potencijalni prelazak na drugačije biblioteke.

1.4.6 Temeljni sustavi

U sloj koji nazivamo temeljni sustavi (eng. *core systems*) svrstavamo razne pomoćne sustave koje upravitelj igara treba. Jedan primjer temeljnog sustava je sustav za upravljanje memorijom. Standardna dinamička alokacija može biti iznimno spora, stoga upravitelji igara implementiraju vlastiti sustav za alokaciju memorije kako bi osigurali bolje performanse. Nadalje, video igre u suštini koriste jako puno matematike, stoga upravitelji igara implementiraju razne matematičke biblioteke koje im omogućuju operacije s vektorima i matricama, razne geometrijske operacije, rješavanje sustava jednadžbi, numeričku integraciju i sve ostalo što je potrebno za razvoj video igara. Također, koristili oni vanjske pakete za strukture podataka i algoritme ili ne, upravitelji igara često zahtijevaju dodatne funkcionalnosti zbog čega implementiraju vlastite pakete za upravljanje strukturama podataka i algoritmima. Postoji još mnogo sustava koje možemo svrstati u ovaj sloj, više njih prikazano je na slici 1.2.

Slika 1.2: Temeljni sustavi upravitelja igre (slika iz [25])

1.4.7 Podsustav za upravljanje resursima

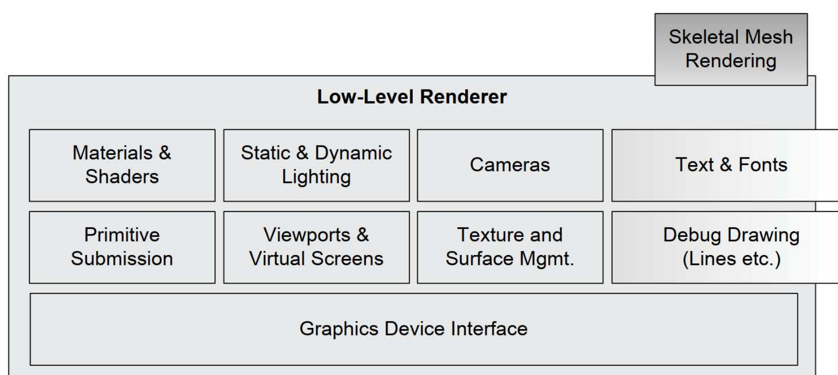
Svaki upravitelj igara u nekom obliku sadrži upravitelja resursima (eng. *resource manager, asset manager, media manager*). Kako su video igre po prirodi multimedijski sustavi, nužno je da upravitelj igara može upravljati velikom količinom resursa (eng. *resources, assets, media*) različitih tipova. Resursi koje koristi upravitelj igara su primjerice teksture, materijali, animacije, poligonske mreže (eng. *polygon mesh*), audio zapisi, sjenčari (eng. *shaders*), kolizijske primitive (eng. *collision primitive*), parametri fizike itd. Upravitelj resursima tijekom izvršavanja programa osigurava da se resursi po potrebi učitaju u memoriju i da se iz nje obrišu kada više nisu potrebni. Također, upravitelji igara često omotavaju sučelje izvornog datotečnog sustava i stvaraju vlastito kako bi bili neovisni o platformi i kako bi dodali nove funkcionalnosti.

1.4.8 Podsustav za iscrtavanje

"Najjeftiniji trokuti su oni koje nikada ne nacrtaš". (iz [25])

Mehanizam iscrtavanja (eng. *Rendering Engine*) jest jako velik, kompleksan i važan dio upravitelja igara. Može se realizirati na mnogo različitih načina, ali ipak većina modernih sustava za iscrtavanja dijeli osnovne koncepte. Često se koristi višeslojna arhitektura opisana u nastavku.

Zadatak iscrtavača niže razine je da prikupi podatke o geometrijskim primitivama (poligonske mreže, liste linija, liste točaka, tekst i sve ostalo što želimo iscrtati) koje je potrebno iscrtati i iscrta ih što je brže moguće. Dijelovi ovog sloja prikazani su na slici 1.3.



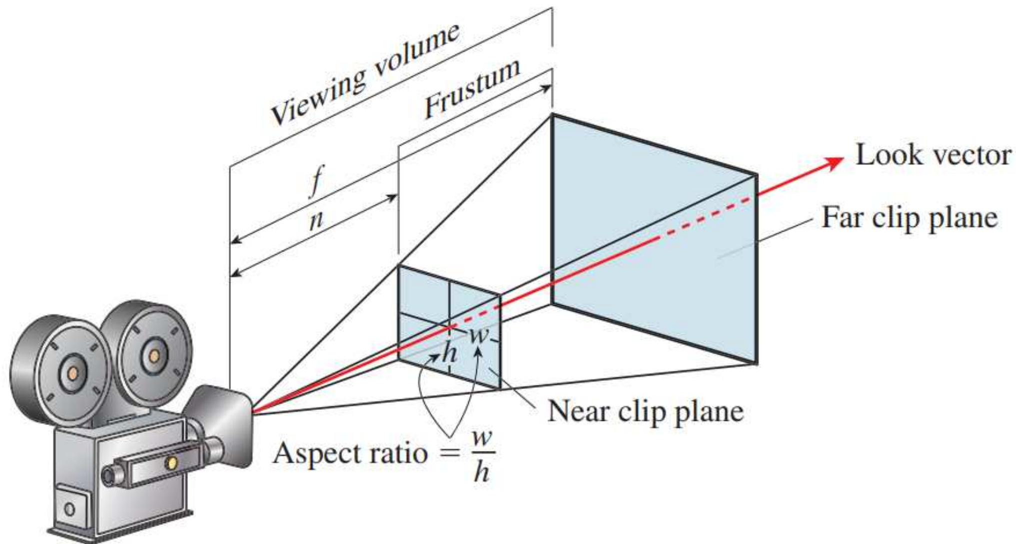
Slika 1.3: Iscrtavača niže razine (slika iz [25])

Sučelje grafičkih uređaja (eng. *graphics device interface*) je komponenta zadužena za numeriranje i inicijalizaciju grafičkih uređaja, pripremu ploha za renderiranje itd.

Iscrtavač niže razine upravlja stanjem grafičkog hardvera i sjenčarima (eng. *shaders*), programima koje izvršava grafičko sklopovlje, preko sustava za materijale i sustava za dinamičko osvjetljenje. Svakoj primitivi koju je potrebno iscrtati pridružen je materijal - opis vizualnih svojstava primitive. On sadrži texture (bitmape pridružene primitivi koje uglavnom sadrže informaciju o bojama) koje koristi primitiva, informaciju o tome koje sjenčare koristiti prilikom iscrtavanja i parametre koji kontroliraju funkcionalnosti grafičkog uređaja. Nadalje, svaka primitiva je pod utjecajem nekoliko dinamičkih osvjetljenja koji određuju kakve kalkulacije za osvjetljenje će se primijeniti na primitive.

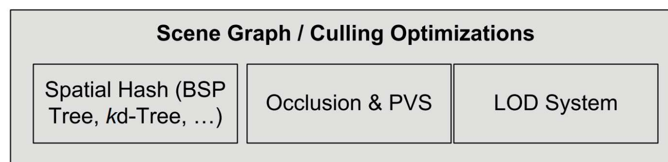
Iscrtavač niže razine također provodi obrezivanje primitiva od frustum (krnju piramidu) (eng. *frustum clipping*) kamere koji je prikazan na slici 1.4. Plohe sa slike okomite na smjer gledanja kamere nazivaju se bliža i dalja ravnina rezanja.

Primitive koje se najčešće koriste su trokuti i to iz nekoliko razloga: trokut je najjednostavniji poligon, uvijek je planaran, trokuti ostaju trokuti i nakon primjene transformacija (eventualno ako se gleda preko ruba može se dobiti linija) te su grafički hardveri bazirani na rasterizaciji trokuta (postupak transformiranja trokuta u piksele ekrana).



Slika 1.4: Frustum kamere (slika iz [21])

Sljedeća komponenta mehanizma za iscrtavanje prikazana na slici 1.5 je zadužena za ograničavanje broja primitiva koje je potrebno iscrtati primjenom nekog oblika određivanja vidljivosti. Ovdje ćemo navesti nekoliko osnovnih.

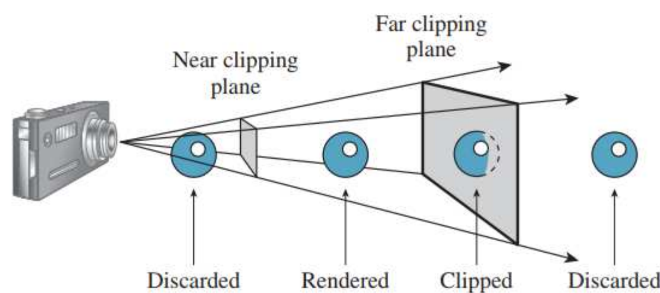


Slika 1.5: komponenta za određivanje vidljivosti (slika iz [25])

Prva metoda je izbacivanje iz frustuma (eng. *frustum cull*) i prikazana je na slici 1.6. Objekti koji se u potpunosti nalaze izvan frustuma se izbacuju.

Čak i kada objekti leže unutar frustuma oni mogu prekrivati jedan drugoga. Otklanjanje objekata koji su u potpunosti zaklonjeni drugim objektom naziva se odstranjivanje prekrivanja (eng. *occlusion culling*).

Sljedeća metoda koristi takozvane portale. Svijet igre podijeli se na poluzatvorena područja međusobno povezana otvorima poput vrata i prozora koje nazivamo portali. Za iscrtavanje scene s portalima prvo iscrtamo područje koje sadrži kameru. Onda za svaki portal unutar danog područja proširimo vidno područje tako da dodamo volumen kojeg čine plohe koje idu iz izvorišta kamere kroz svaki rub portala. Sadržaj susjednog područja



Slika 1.6: Izbacivanje iz frustuma (slika iz [21])

se zatim obrezuje od taj volumen, odnosno iscrtava se samo sadržaj koji je vidljiv kroz portal.

Odstranjivanje preklapanja u velikim okolinama može se provesti računanjem potencijalno vidljivog skupa (eng. *potentially visible set*). To se može provesti tako da se okolina podijeli u područja i svakom se području dodjeli lista područja koja su vidljiva kada se kamera nalazi unutar danog područja.

Primjena nekog oblika određivanja vidljivosti može značajno ubrzati vrijeme potrebno za iscrtavanje scene, pogotovo ako su u pitanju iznimno veliki virtualni svjetovi.

Mehanizmi iscrtavanja često imaju odvojenu komponentu, ili više njih, zaduženu za upravljanje vizualnim efektima poput sustava čestica (za prikazivanje dima, vatre i slično), dinamičkih sjena, mapiranja osvjetljenja, raznim efektima koji se primjenjuju nakon što se scena iscrta poput korekcija boja, antialijasiranja (zaglađivanje linija koje izgledaju nazubljeno radi korištenja diskretnog skupa pixela za njihov prikaz) itd.

Ovdje također spadaju i sustav odgovoran za puštanje unaprijed snimljenih videa tijekom video igre te sustav koji omogućava koreografiranje filmskih sekvenci unutar same igre (eng. *in-game cinematics*)

1.4.9 Podsustav za profiliranje i otklanjanje grešaka

Profiliranje je dinamička analiza programa koja koristeći informacije prikupljene tijekom izvršavanja programa mjeri primjerice vremensku i prostornu složenost programa, trajanje i učestalost funkcijskih poziva itd. Cilj profiliranja je određivanja koji dijelovi programa se mogu optimizirati.

Programeri se tijekom razvoja video igre često susreću s raznim problemima i greškama tijekom izvođenja igre i tu im u pomoć priskaču alati za pomoć pri otklanjanju grešaka (eng. *debugging tools*).

Upravitelji igara često uključuju razne alate za profiliranje i otklanjanje grešaka kako bi programerima olakšali razvoj igara. Mehanizam za prikazivanje rezultata analize na

ekranu tijekom izvođenja igre ili za spremanje rezultata u tekstualnu datoteku, mehanizam za utvrđivanje potrošnje memorije za pojedine podsustave, mehanizam snimanja događaja u svrhu pronalaženja grešaka samo su neke od mogućnosti koje upravitelji igara mogu implementirati.

1.4.10 Podsustav za kolizije i fiziku

Još jedan iznimno važan dio upravitelja igre je sustav za detekciju kolizija. Bez njega ne bismo mogli detektirati sudare objekata, čime bi izgubili i međudjelovanje tih predmeta. Objekti bi prolazili jedni kroz druge, ne bismo mogli detektirati dodirivanje tla, udaranje u zid, itd., stoga je nužno da svaki upravitelj igara ima nekakav oblik detekcije kolizija.

Sustav za detekciju kolizija obično je usko povezan s upraviteljem fizike (eng. *physics engine*). Ono što se u svijetu video igara naziva "fizikom" zapravo je samo dio fizike koji se bavi dinamikom krutih tijela, tj. upravitelje igara zanima utvrđivanje utjecaja sila na idealizirana čvrsta tijela. Upravitelj fizike uvelike koristi sustav za detekciju kolizija kako bi što vjernije prikazao razna fizička ponašanja. U novije vrijeme video igre sve više uključuju kompleksne simulacije fizike poput mehanike fluida u stvarnom vremenu i simulacije tijela koja se mogu deformirati, ali to dolazi s visokom cijenom.

Pisanje sustava za detekciju kolizija i za simulaciju fizike je iznimno zahtjevan i dugotrajan proces. Stoga, većina upravitelja igara koristi softvere trećih strana za ovu namjenu, primjerice *Havok* ili *PhysX*. *Unreal Engine* koristi *PhysX* tvrtke NVIDIA. *PhysX* kruta tijela naziva akterima (eng. actors), svaki akter sadrži informacije o obliku odnosno geometrijskoj strukturi aktera i transformaciju koja opisuje poziciju i orijentaciju aktera u virtualnom svijetu. Često se jednostavni geometrijski oblici koriste za aproksimaciju volumena koji se koristi za detekciju kolizija.

1.4.11 Podsustav za animacije

Postoji nekoliko tipova animacija koje se koriste u video igrama, ali danas najviše prevladava takozvana skeletna animacija. Ona omogućava postavljanje kompleksnih 3D mreža na jednostavan sustav kostiju tako da se vrhovi miču zajedno sa skeletom.

Iluziju pokreta u video igrama dobivamo postavljajući tijelo u niz poza koje se zatim prikazuju brzinom od 30 do 60 poza u sekundi. U skeletnoj animaciji, poza skeleta direktno određuje poziciju vrhova 3D mreže pridružene skeletu. Skelet se stavlja u pozu time da rotiramo, translatiramo i skaliramo njegove zglobove. Poza svakog zgloba reprezentira se matricom, a poza skeleta je zapravo skup poza svih njegovih zglobova pa je predstavljena poljem matrica.

Sustav za skeletnu animaciju povezan je s mehanizmom za iscrtavanje tako što sustav za animaciju određuje pozu skeleta i šalje skup matrica pridružen toj pozi mehanizmu

za iscrtavanje pomoću kojih on zatim transformira svaki vrh. Ovaj proces pridruživanja vrhova 3D mreže skeletu u određenoj pozi naziva se *skinning*.

Sustav za animaciju i sustav za fiziku su također usko povezani ako se koriste tzv. krpene lutke (eng. *rag dolls*). Kada lik u igrici umre ili padne u nesvijest, njegovo tijelo postane „mlitavo" i u toj situaciji želimo da tijelo reagira fizički korektno sa svojom okolinom. Za to koristimo krpenu lutku koja je zapravo skup krutih tijela spojenih na mjestima zglobova kojima upravljamo preko sustava za fiziku.

1.4.12 Uređaji za interakciju s korisnikom

Svaka video igra obrađuje ulazne podatke dobivene od korisnika preko nekog ulaznog uređaja poput tipkovnice, miša ili kontrolera. Ponekad se na uređaj za interakciju s korisnikom šalje i nekakav izlazni učinak, primjerice kada se kontroler zatrese tijekom određene radnje u igrici (kada primimo udarac i slično). Upravitelji igara uglavnom skrivaju detalje konkretnih uređaja od kontrola implementiranih u igri te je korisniku često omogućeno da prilagodi mapiranje fizičkih kontrola na određene funkcije u igri.

1.4.13 Audio podsustav

Zvuk je u video igrama jako bitan, ali je nažalost često podcijenjen. Kvalitetni zvučni efekti i pozadinske melodije mogu kompletno promijeniti doživljaj igre. Kvalitetan 3D audio sustav bi trebao što vjerodostojnije reproducirati zvukove koje bi igrač čuo da se uistinu nalazi u virtualnom svijetu video igre. Audio upravitelj je također odgovoran i za zvukove koji ne potječu iz virtualnog svijeta kao što je pozadinska glazba, zvukovi menija, glas naratora ili glas lika igrača. Takve zvukove nazivamo 2D zvukovima jer ne ovise o udaljenosti.

Teorija obrade signala je grana matematike koja je temelj gotovo svakog aspekta digitalne audio tehnologije

1.4.14 Podsustav za online igre s više igrača

Postoje četiri osnovna tipa igara s više igrača: Igre s više igrača na jednom ekranu, igre s više igrača na podijeljenom zaslonu, umrežene igre s više igrača i masivne online igre s više igrača.

Podržavanje igara s više igrača značajno utječe na većinu komponenti upravitelja igara. Naknadno mijenjanje upravitelja igara kako bi podržavao igre s više igraća je zahtjevno, stoga se preporučuje odmah implementirati tu funkcionalnost.

Kod umreženih igri s više igrača stanjem igre obično upravlja jedan stroj, ali se ono mora prenijeti i na sve ostale uređaje kako bi svi igrači imali prikazano jednako stanje

igre. Fizika koja ne utječe na tijek igre se može simulirati zasebno na svakom klijentskom računalu, ali fizika koja utječe na tijek igre mora se simulirati na serveru i precizno prenijeti na sve klijente.

1.4.15 Temelji za proces igranja

Kada bi imali liniju koja odvaja upravitelja igara od specifične igre, ovaj sloj stajao bi točno na toj liniji - neki ga svrstavaju na jednu stranu, neki na drugu. Ovaj sloj služi kao spojnica implementacije igre (pravila igre, mehanike igrača, ciljevi u igri itd.) i prethodno opisanih dijelova upravitelja igre. Pruža razne sadržaje pomoću kojih se logika igre i elementi virtualnog svijeta mogu jednostavnije implementirati. Iako je razlika među upraviteljima igara najuočljivija u ovom sloju, postoje neke zajedničke točke koje ima većina upravitelja igara.

Objekti u igri moraju komunicirati jedni s drugima, stoga je jedan od podsustava kojeg sadrži većina upravitelja igara takozvani sustav za događaje (eng. *event system*). U sustavu vođenom događajima, objekt koji okida događaj stvara strukturu podataka koju kratko nazivamo samo događaj (eng. *event*) i ona sadrži sve informacije o događaju koji se pokrenuo. Ta struktura se pošalje odredišnom objektu tako što se pozove njegova funkcija za upravljanje događajima.

Nadalje, većina upravitelja igara koristi sustav za skriptiranje. Skriptni jezik je jezik više razine, relativno jednostavan za korištenje, koji omogućava pristup raznim funkcionalnostima upravitelja igre. Korištenje skriptiranja pojednostavljuje razvoj logike i sadržaja igre.

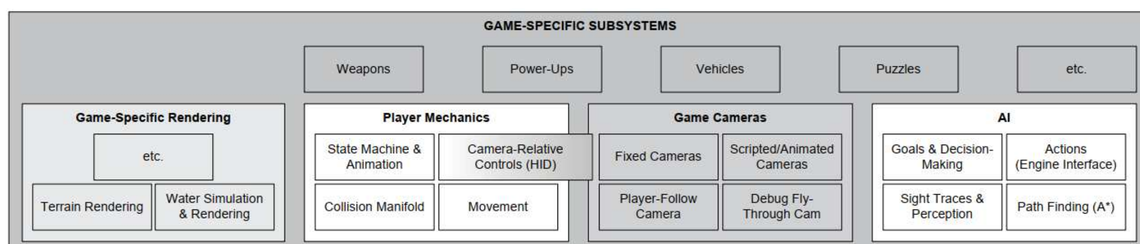
Sustav za upravljanje nivoima (eng. *levels*) učitava sadržaj virtualnog svijeta. Često se podaci o nivou prenose u memoriju tijekom izvođenja igre stvarajući iluziju ogromnog kontinuiranog svijeta.

Važan dio je podsustav koji pretvara apstraktni model igre koji je vidljiv u uređivaču svijeta u konkretnu implementaciju tog modela. Ovo je tipično provedeno na jedan od dva načina od kojih je prvi usmjeren na objekte gdje je svaki objekt tijekom izvršavanja reprezentiran instancom neke klase koja ima vlastite metode i atribute, a drugi način je usmjeren na svojstva, odnosno svaki objekt je reprezentiran s jedinstvenim identifikacijskim kodom, a svojstva objekata nalaze se u raznim tablicama podataka, po jedna za svako svojstvo.

1.4.16 Podsustavi specifični za igru

Ovaj podsustav ne spada pod upravitelja igre, nego pod samu video igru. Na slici 1.7 su prikazani razni sustavi koji su specifični za pojedinu igru. Oni su važni za igračevo iskustvo, a neki standardniji su primjerice mehanike igrača, sustavi kamera unutar igre i umjetna inteligencija za kontrolu likova koji nisu igrač. Postoji još ogroman broj manjih

sustava specifičnih za igru poput sustava za oružja, sustav za uređivanje lika, sustav za vozila, zagonetke itd.



Slika 1.7: Podsustavi specifični za igru (slika iz [25])

Sustav za umjetnu inteligenciju obično nije bio dio upravitelja igara, ali kako su uočeni brojni uzorci koji se pojavljuju u gotovo svakom sustavu za umjetnu inteligenciju, on polako postaje standardni dio upravitelja igara.

Poglavlje 2

Oblikovni obrasci u razvoju video igara

Oblikovni obrasci (eng. *design patterns*) su rješenja za standardne programerske probleme. To su zapravo ideje, koncepti rješavanja tih problema, ne i njihova konkretna implementacija, stoga se prilikom korištenja svaki oblikovni obrazac treba prilagoditi specifičnostima konkretnog projekta. Glavna svrha korištenja oblikovnih obrazaca je postizanje fleksibilnosti te mogućnost višestruke upotrebe dijelova koda. Iako oblikovni obrasci pružaju pametna i efikasna rješenja za razne programerske probleme, njihovo korištenje nije nužno uvijek najbolji izbor za svaki program. Prilikom donošenja odluka potrebno je „izvagati” njihove prednosti i mane, razmisliti da li bi nam korištenje nekog oblikovnog obrasca značajno poboljšalo kod i olakšalo razvoj programa ili bi time samo bespotrebno dodali kompleksnost koju smo mogli izbjeći korištenjem nekog jednostavnijeg pristupa. Postoji jako puno oblikovnih obrazaca, neki su jednostavni i direktni, neki kompleksni i apstraktni. Oblikovni obrasci značajan su dio razvoja svih video igara pa tako i onih razvijenih pomoću *Unreal Engine*-a. Neki su korišteni već u samom izvornom kodu upravitelja igara, a neke sami biramo implementirati prilikom razvoja vlastite video igre. U nastavku je navedeno nekoliko oblikovnih obrazaca koji se često koriste prilikom razvoja video igara. Prva tri oblikovna obrasca najvažnija su za razvoj video igara pa ih i *Unreal Engine* uvelike koristi. Nakon njih slijedi nekoliko obrazaca koji su također česti u razvoju video igara i drugih softvera, ali uz navedene postoji još mnogo obrazaca. Opis svakog od njih je van dosega ovog rada. Više informacija može se pronaći u [29].

2.1 Objektno orijentirano programiranje

Iako nije nužno, video igre se uglavnom razvijaju po objektno orijentiranom pristupu. Kako je *Unreal Engine* pretežno objektno orijentiran softver sa svojih bezbroj klasa koje se međusobno nasljeđuju, objasnit ćemo ukratko osnovne pojmove vezane uz objektno orijentirano programiranje.

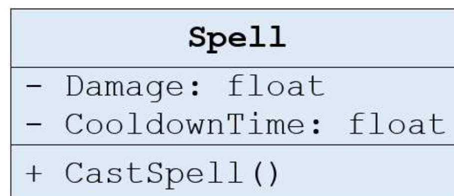
2.1.1 Objekti i klase

Osnovni pojmovi u objektno orijentiranom programiranju su klase i objekti. Klase predstavljaju vrstu objekata dok su objekti instance klasa. Objekti se sastoje od svojstava još nazivanih i atributima te metoda, odnosno procedura ili funkcija. Programi u objektno orijentiranom programiranju se sastoje od objekata koji su u međusobnoj interakciji, a metodama pridruženim objektima manipuliramo njihovim podacima.

Pogledajmo primjer klase koja je pojednostavljeni primjer klase korištene u video igri implementiranoj u sklopu ovog rada. Za grafičko prikazivanje klasa se standardno koriste UML (*Unified Modelling Language*) dijagrami. Na vrhu se nalazi ime klase, s „–" su označena privatna svojstva i metode, a s „+" javna.

```
1  class Spell
2  {
3      public:
4          Spell(float DamageValue, float CooldownTimeValue);
5          void CastSpell();
6      private:
7          float Damage;
8          float CooldownTime;
9  };
```

Isječak koda 2.1: Primjer klase



Slika 2.1: UML dijagram za klasu Spell

Klasa `Spell` predstavlja novu, korisnički definiranu vrstu podataka koju možemo koristiti u svom programu. Unutar programa možemo stvoriti instancu te klase te na njoj pozivati metodu `Castspell()` kako bi dobili ponašanje koje je definirano unutar klase, primjerice, ispaliti projektil koji nanosi štetu u vrijednosti varijable Damage (neprijatelj može također biti instanca neke druge klase koja predstavlja sve neprijatelje).

Sada ćemo objasniti osnovne koncepte objektno orijentiranog programiranja: enkapsulaciju, nasljeđivanje i polimorfizam.

2.1.2 Enkapsulacija

U objektno orijentiranom programiranju možemo grupirati podatke i funkcije u samostalne entitete koji su zaduženi za očuvanje vlastitog integriteta. Ključne riječi `public` i `private` koje vidimo na prethodnom primjeru označavaju vrstu pristupa svojstvima i metodama. Svojstva i metode definirani iza ključne riječi `public` mogu se dohvaćati iz cijelog programa, dok se one navedene nakon ključne riječi `private` mogu dohvaćati samo unutar klase koja ih definira, odnosno skrivene su ostatku programa. Postoji još i vrsta pristupa pod nazivom `protected` koja definira da se svojstva i metode mogu dohvaćati i unutar klasa koje nasljeđuju (nasljeđivanje je objašnjeno u sljedećem odjeljku) danu klasu, ali ne i u ostatku programa. Ovime se nameće enkapsulacija jer se dio koda koji se nalazi pod private labelom skriva, odnosno enkapsulira, od ostatka programa koji koristi ti klasu. Enkapsulacijom se skrivaju detalji implementacije ostatku koda čime se osigurava veća stabilnost koda, odnosno manja mogućnost njegova narušavanja. Uglavnom je praksa da se svojstvima objekta ne pristupa direktno već isključivo koristeći njegove javne metode.

2.1.3 Nasljeđivanje

Recimo da imamo klasu A i da želimo stvoriti objekt koji ima sve što se nalazi u klasi A, ali nam treba i još ponešto. Upravo tome služi koncept nasljeđivanja. Možemo stvoriti novu klasu B koja nasljeđuje klasu A i tada ona sadrži sva svojstva i metode koje sadrži i klasa A, može ih modificirati ako joj klasa A to dozvoljava i dodavati vlastite. Klasu A zovemo bazna klasa ili klasa roditelj, a klasu B izvedena klasa ili klasa dijete. Također, kažemo da klasa B jest klasa A jer instancu klase B možemo koristiti tamo gdje se očekuje instanca klase A pošto ona sadrži sve što sadrži i klasa A. Ovaj koncept koristimo ako želimo implementirati više vrsta podataka koji imaju dosta toga zajedničkog. Ako imamo klasu `Spell()` koja predstavlja sve čarolije, možemo implementirati posebne podvrste čarolija koje nasljeđuju klasu `Spell()`. Tada one također imaju svojstva `Damage` i `CooldownTime` te metodu `CastSpell()`, ali mogu definirati nešto drugačije ponašanje, drugačiju implementaciju metode `CastSpell()` ili dodati neke nove metode i svojstva.

2.1.4 Polimorfizam

Polimorfizam nam omogućuje da pozivamo metode iz različitih izvedenih klasa koje pripadaju istoj hijerarhiji, bez direktnog znanja o kojoj točno klasi je riječ. Recimo da imamo dvije izvedene klase koje nasljeđuju klasu `Spell` – `ForwardProjectileSpell` i `RadialProjectileSpell`, koje imaju različite implementacije metode `CastSpell()`. Unutar klase `Staff` spremljeni su pokazivači na instance klasa `ForwardProjectilespell` i `RadialProjectileSpell`, ali su spremljeni u obliku pokazivača na klasu `Spell`. Ipak,

prilikom poziva metode `CastSpell()` na pojedinoj instanci, pozvat će se točna verzija metode `CastSpell()` ovisno o tome kojeg je tipa instance na koju pokazivač pokazuje. Kako bi se ovo postiglo potrebno je metodu `Castspell()` unutar klase `SpellBase` proglasiti virtualnom.

2.2 *Game Loop*

Game Loop, odnosno petlja igre, je najvažniji oblikovni obrazac u razvoju video igara i koriste ga gotovo sve video igre. U prethodnim poglavljima već je spomenuto kako je interaktivnost najvažnija značajka video igara. Ono što razlikuje video igre od većine drugih interaktivnih softvera jest to što igra nije blokirana dok čeka unos korisnika. Čak i ako korisnik ne pritišće tipkovnicu ili miš, na ekranu se odvijaju razne animacije te ako igricu nismo pauzirali, neprijatelji će nas i dalje proganjati. S druge strane, primjerice u slučaju neke web aplikacije, standardno se ništa neće događati ako korisnik ne klikne na nešto, tj. takve aplikacije pasivno čekaju korisnikov unos. Ovaj obrazac, kako i samo ime govori, sastoji se od petlje. Ta petlja se izvršava dokle god je igrica pokrenuta. Pojednostavljena verzija petlje igre izgleda ovako

```
1  while( true )
2  {
3      processInput();
4      update();
5      render();
6  }
```

Isječak koda 2.2: Osnovna petlja igre (preuzeto iz [29])

Konkretne implementacije petlje igre su mnogo kompliciranije, ali uglavnom sve imaju osnovne dijelove prikazane u gornjem isječku koda. Prvo se obrade potencijalni unosi korisnika pozivom metode `processInput()`, zatim se pomoću metode `update()` ažurira stanje igre, simulacija fizike, umjetna inteligencija (ako je prisutna u video igri) i drugi sustavi koji zahtijevaju ažuriranje. Konačno se igra iscrta na ekran pozivom metode `render()`. Navedeni koraci se ponavljaju, u ovom pojednostavljenom slučaju beskonačno mnogo puta jer imamo petlju čiji je uvjet uvijek istinit, ali u stvarnoj implementaciji bi uvjet petlje bio nešto konkretniji i provjeravao bi da li treba završiti igru.

Jedan prolazak kroz petlju igre obično zovemo otkucaj (eng. *tick*) ili kadar (eng. *frame*) pa tako brzinu igre, odnosno brzinu izvršavanja petlje igre, mjerimo brojem otkucaja, odnosno kadrova u sekundi (eng. *Frames Per Second*, FPS). Što je FPS veći, to igra bolje teče i izgleda glađe, tj. nema zastajkivanja. Brzina izvršavanja petlje igre ovisi o brzini računala

koji pokreće video igru te o kompleksnosti i veličini koda koji se izvršava svakim prolaskom kroz petlju. Kako se današnje igre pokreću na mnogo različitih računala s različitim performansama, nužno je da se igre razvijaju neovisno o broju kadrova u sekundi kako korisnici koji igraju na puno bržem ili puno sporijem računalu ne bi imali mnogo drugačije iskustvo tijekom korištenja video igre.

Problem pojednostavljenog pristupa prikazanog u isječku koda 2.2 je u tome što nemamo nikakvu kontrolu nad brzinom izvođenja petlje, ona će se izvršavati što brže može. Problem se stvara kada brzina izvođenja petlje nije konzistentna sa stvarnim protokom vremena. Zamislimo primjerice da imamo projektil koji putuje stalnom brzinom u određenom smjeru. Njegovu poziciju mijenjamo unutar metode `update` i ako svakim prolaskom kroz petlju igre pozivamo metodu `update`, koja ne zna koliko je stvarnog vremena prošlo od zadnjeg prolaska kroz petlju, tada ta metoda svaki put pomakne projektil za jednak iznos u smjeru njegovog kretanja. Posljedica ovoga je što će projektil od točke A do točke B stići mnogo brže na brzom računalu, dok će se na sporom računalu kretati sporo uz moguće vidljivo zastajkivanje što uzrokuje veliku razliku u iskustvu igranja korisnika na različitim računalima. Prva verzija poboljšanog koda bi bila da na kraju petlje procesor odspava ako je od početka iteracije prošlo manje vremena od nekog predefiniranog fiksnog vremenskog perioda potrebnog za ažuriranje. Time možemo kontrolirati da se petlja ne izvršava suviše brzo, ali to rješava samo dio problema. Druga verzija rješava problem mjerenjem proteklog vremena od prethodne iteracije unutar petlje igre i slanjem tog vremena metodi `update`, odnosno tada bi imali poziv oblika `update( ElapsedTime )`. Time možemo unutar metode `update` projektil pomicati ovisno o proteklom vremenu i tako će brzina pomicanja projektila na ekranu biti konzistentna jer će se na brzom računalu projektil kretati puno manjim koracima nego na sporom računalu.

Sada se nameće drugi problem, a to je nedeterminističnost. Naime, brzo računalo će mnogo puta proći kroz petlju igre, ažurirati stanje igre, simulaciju fizike i ostalo, mnogo puta će raditi razne izračune, vjerojatno koristeći kompleksne matematičke formulacije, i time će mnogo puta doći do zaokruživanja rezultata prilikom aritmetike s brojevima s pomičnom točkom (eng. *Floating-point arithmetic*). Stoga će brzo računalo akumulirati puno veću grešku zaokruživanja nego sporo računalo što može dovesti do bitno drugačijih rezultata, primjerice projektil koji se ispuca će završiti na drugačijoj poziciji.

Dakle, metoda s varijabilnim koracima ažuriranja nije idealno rješenje. Bilo bi dobro da iteracije kroz petlju možemo obavljati u pravilnim intervalima, ali kako bi se prilagodili većem broju računala, vrijeme potrebno jednom prolasku petlje bi vjerojatno trebalo biti veće no što bi htjeli. Ipak, možemo napraviti kompromis. Iscrtavanje igre na zaslon zahtjeva mnogo procesorskog vremena, ali nam uglavnom nije potrebno toliko često osvježavanje zaslona koliko nam je potrebno ažuriranje fizike radi održavanja stabilnosti same video igre. Sljedeća varijacija petlje igre uzima to u obzir.

```
double previous = getCurrentTime();
double lag = 0.0;
while (true)
{
    double current = getCurrentTime();
    double elapsed = current - previous;
    previous = current;
    lag += elapsed;

    processInput();

    while (lag >= MS_PER_UPDATE)
    {
        update();
        lag -= MS_PER_UPDATE;
    }

    render(lag / MS_PER_UPDATE);
}
```

Isječak koda 2.3: Petlja igre (preuzeto iz [29])

Na početku petlje dohvatimo trenutno vrijeme i izračunamo koliko igra zaostaje u odnosu na stvarno vrijeme, odnosno koliko je vremena prošlo od prethodne iteracije petlje. Rezultat pohranimo u varijablu `lag`. Nakon što obradimo unose korisnika, obavljamo potrebna ažuriranja onoliko puta koliko vremenskih intervala koji su potrebni za jednu iteraciju ažuriranja (ta informacija je pohranjena u varijabli `MS_PER_UPDATE`) stane u vrijeme spremljeno u varijabli `lag`. Tek nakon što je igra „sustigla“ stvarno vrijeme (ne sasvim jer od `lag` oduzimamo višekratnik od `MS_PER_UPDATE` pa možemo imati mali ostatak koji se prenosi u sljedeću iteraciju), iscrtavamo igru na ekran. Vrijednost koju šaljemo metodi render predstavlja normalizirano vrijeme koje nismo još nadoknadili, odnosno vrijeme koje se tek treba nadoknaditi u sljedećoj iteraciji petlje igre. To je potrebno kako bi se objekti na ekranu mogli iscrtati na točnijoj poziciji, poziciji koja odgovara stvarnom vremenu. Dio koda zadužen za iscrtavanje zna brzine kretanja svih objekata u igri pa je sposoban realizirati traženi pomak.

Ovo je još uvijek poprilično pojednostavljena implementacija petlje igre. Srećom, prilikom razvoja video igara u *Unreal Engine*-u ne moramo brinuti o kompleksnoj implementaciji petlje igre jer je ona implementirana u samom upravitelju.

2.3 *Update method*

Update method, odnosno metoda ažuriranja, je oblikovni obrazac koji ide ruku pod ruku s *game loop* obrascem. Generalno, od video igara očekujemo da se na ekranu nešto aktivno događa, primjerice da se likovi kreću ili da projektili letu zrakom. Potrebno je dakle stalno ažurirati pozicije i stanja objekata. Recimo da imamo objekt koji se treba vječno kretati u nekom smjeru. Ne možemo samo staviti kod za promjenu njegove pozicije u beskonačnu petlju jer time, ako koristimo samo jednu programsku nit, blokiramo ostatak programa, drugi objekti se ne mogu ažurirati, korisnik ne može upravljati svojim likom itd. Objekti se zapravo trebaju ažurirati u koracima tako da svaki objekt može napraviti dio potrebnog ažuriranja sa svakim novim kadrom. Ažuriranje je dakle potrebno obavljati unutar petlje igre, kao što je bilo i prikazano u prethodnom poglavlju.

Recimo da imamo 10 objekata i da svaki radi nešto drugo, odnosno imamo drugačiji kod koji se koristi za ažuriranje pojedinog objekta, i sav taj kod strpamo u petlju igre. Recimo sada da imamo 20 takvih objekata, 50, 100. Nastao bi kaos unutar petlje igre, kod bi bio nepregledan i neodrživ. Rješenje je da se svaki objekt brine o vlastitom ažuriranju i da se svi objekti koji zahtijevaju ažuriranje nalaze u zajedničkoj kolekciji. Svaki objekt implementira metodu `update` u kojoj ažurira vlastito stanje za jedan korak. Također, potrebno je ili imati baznu klasu za objekte koji se mogu ažurirati i u njoj apstraktnu metodu `update` ili koristiti komponente, koje su tema sljedećeg oblikovnog obrasca. U svakoj iteraciji petlja igre prolazi kroz kolekciju objekata koji zahtijevaju ažuriranje i poziva nad njima metodu `update`.

```
1  for( int i = 0; i < sizeof(objectCollection); ++i)
2  {
3          objectCollection[i]->update();
4  }
```

Isječak koda 2.4: Ažuriranje objekata

Ažuriranje objekata u koracima sa svakim otkucajem zahtjeva korištenje dodatnih varijabli i memorije kako bi se pohranilo stanje svakog objekta da bi se moglo nastaviti s ažuriranjem u sljedećoj iteraciji petlje igre. Naglasimo da je ponekad važno kojim redom objekte ažuriramo s obzirom na to da se svi objekti ne ažuriraju istovremeno već jedan za drugim. Objekti često mogu unutar svoje `update` metode dohvaćati informacije i o drugim objektima unutar svijeta. Stoga ako se objekt A nalazi prije objekta B u kolekciji objekata, ono će unutar vlastite `update` metode moći dohvatiti jedino staro stanje objekta B, dok će objekt B u svojoj `update` metodi moći dohvatiti jedino novo stanje objekta A. Sekvencijalno ažuriranje objekata je generalno dobra stvar jer se tako izbjegavaju neugodne situacije koje mogu nastati zbog paralelizma.

Unutar `update` metoda često se dodaju novi objekti koji mogu zahtijevati ažuriranje. Ako ih samo dodamo na kraj kolekcije, oni će se ažurirati u istoj iteraciji petlje u kojoj su i dodani. To obično nije željeno ponašanje, a možemo ga vrlo jednostavno promijeniti. Potrebno je samo na početku iteracije petlje prebrojati koliko ima objekata koje je potrebno ažurirati i zaustaviti petlju koja prolazi kroz kolekciju objekata kada se dostigne taj broj. Na ovaj način će novo dodani objekti biti ažurirani tek u sljedećoj iteraciji. Do malo većeg problema dolazi kada objekt maknemo iz kolekcije tijekom ažuriranja jer tada možemo slučajno preskočiti neki drugi objekt. Ovaj problem se može riješiti popravljanjem pozicije iteratora ili odgađanjem uklanjanja objekta iz kolekcije dok se ne završi s ažuriranjem svih objekata u trenutnoj iteraciji.

2.3.1 *Update method* i *Unreal Engine*

Ticking ili otkucavanje predstavlja izvršavanje nekog koda jednom po kadru ili u intervalima. Sukladno tome, objekti u *Unreal Engine*-u ažuriraju se pomoću metode `Tick()`. `Tick()` je dakle metoda `update()` iz *Update method* oblikovnog obrasca. Varijabla `DeltaTime` predstavlja vrijeme proteklo od prethodnog iscrtavanja na ekran, odnosno od prethodne iteracije petlje, i ona se koristi prilikom ažuriranja.

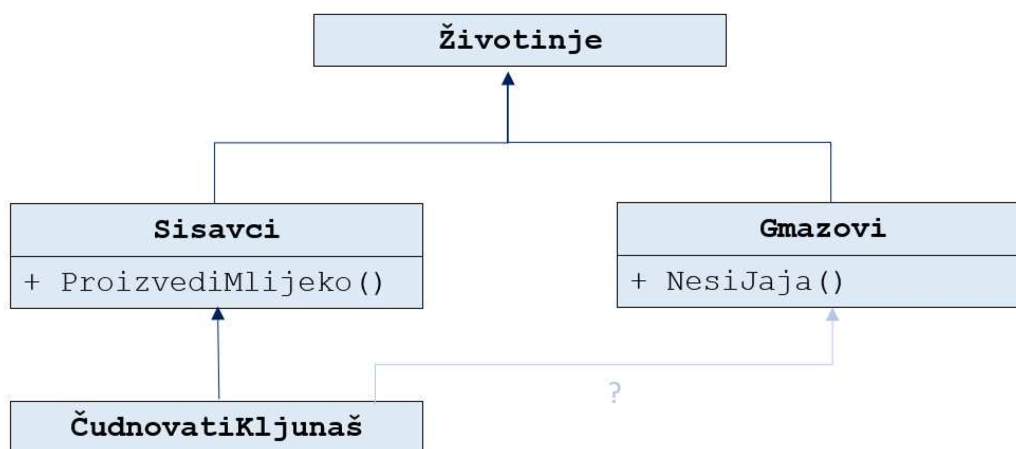
Aktere i komponente možemo dodijeliti nekoj grupi ažuriranja kako bi zadali kada se tijekom kadra treba obaviti njihovo ažuriranje s obzirom na ostale procese, prvenstveno s obzirom na upravitelja fizikom (eng. *physics engine*). Grupe ažuriranja su: `TG_PrePhysics` (ažurira se na početku kadra), `TG_DuringPhysics` (ažurira se dok se vrši simulacija fizike), `TG_PostPhysics` (ažurira se nakon simulacije fizike), `TG_PostUpdateWork` (ažurira se nakon što se obave razna druga ažuriranja). Nadalje, može se pojedinom akteru ili komponenti dodati zavisnost o ažuriranju nekog drugog aktera ili komponente što koristimo najčešće ako se oboje nalaze u istoj grupi. Time će akter ili komponenta koja ovisi o drugom akteru ili komponenti čekati da se prvo obavi njeno ažuriranje. Također, naslijeđene klase se uvijek ažuriraju nakon svojih roditelja.

2.4 *Component*

Komponenta (eng. *Component*) predstavlja oblikovni obrazac koji omogućuje alternativni način dijeljenja funkcionalnosti među entitetima u odnosu na nasljeđivanje. Važan problem u programiranju, pogotovo u velikim softverima, je problem međusobne ovisnosti klasa (eng. *coupling*). Korištenjem komponenti ne stvara se čvrsta veza među klasama kakva nastaje prilikom nasljeđivanja već klase koriste funkcionalnosti definirane u nekoj drugoj, zasebnoj klasi tako da sadrže instancu klase koja predstavlja željenu komponentu. Na ovaj način različiti dijelovi funkcionalnosti stoje zasebno i ne utječu jedan na dru-

goga. Također se „razbija“ kod na manje dijelove koje je lakše održavati i organizirati. Koncept korištenja komponenti u objektno orijentiranom programiranju naziva se kompozicija. Iako se nasljeđivanje dugo smatralo vodećom metodom u objektno orijentiranom programiranju, u novije vrijeme u programerskim krugovima često se čuje rečenica „Composition over inheritance“, tj. kompozicija je u većini slučajeva primjerenija metoda od nasljeđivanja. Nasljeđivanje se svakako još uvijek upotrebljava i često je jako korisno, ali se preporuča da, kada god je to moguće, se odabere kompozicija radije nego nasljeđivanje.

Različite komponente su međusobno razdvojene i najčešće ne znaju jedna za drugu i mogu se samostalno razvijati. Ponekad komponente ipak moraju međusobno komunicirati i dijeliti podatke. Komponente mogu međusobno komunicirati tako da modificiraju stanje objekta koji ih sadrži, što zahtjeva da se neke informacije koje bi se trebale nalaziti u komponenti prebace u klasu koja je vlasnik komponente. Na ovaj način pristup tim informacijama mogu imati i ostale komponente, što nije idealno. Drugi način je da komponente komuniciraju direktno jedna s drugom tako što čuvaju reference jedna na drugu. Ovakav pristup je jednostavan i direktan, ali stvara jaku ovisnost među komponentama. Treći način je da se implementira sustav za slanje poruka kojeg komponente mogu koristiti.



Slika 2.2: Nasljeđivanje

Pogledajmo primjer sa slike 2.2. Baznu klasu nam predstavlja klasa `Životinje` koju onda nasljeđuju klase `Sisavci` i `Gmazovi`. `Sisavci` imaju sposobnost proizvoditi mlijeko pomoću metode `ProizvediMlijeko()`, a `Gmazovi` mogu nesti jaja pomoću metode `NesiJaja()`. Dalje bi mogli imati primjerice klase `Psi` ili `Mačke` koje nasljeđuju klasu `Sisavci` te klasu `Kornjače` koja nasljeđuje klasu `Gmazovi`. Htjeli bi ovom stablu nasljeđivanja dodati klasu `ČudnovatiKljunaš` koji se svrstava pod sisavce, ali ima

sposobnost nesti jaja pa bi htjeli da ima i tu funkcionalnost. Možemo kopirati i dodati kod za nesti jaja direktno u klasu `ČudnovatiKljunaš`, ali generalno u programiranju želimo što manje dupliciranog koda. Stoga nam ovdje može pomoći korištenje komponenti `KomponentaZaProizvodnjuMlijeka` i `KomponentaZaNešenjeJaja` čime bi klase `Sisavci` i `Gmazovi` imali po jednu pripadnu komponentu dok bi klasa ČudnovatiKljunaš imala obje komponente.

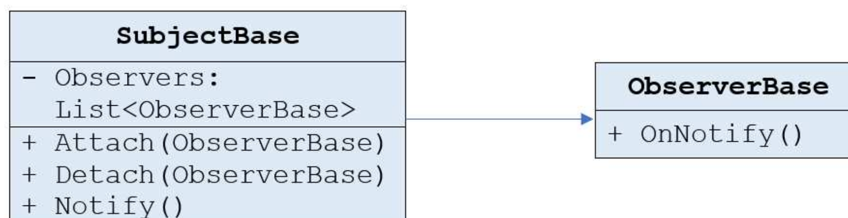
Komponente nam omogućavaju kreiranje kompleksnih entiteta, koji se često protežu kroz više različitih domena, jednostavnim dodavanjem potrebnih komponenti koje sadrže funkcionalnosti koje želimo dodati novoj klasi. Svaka klasa kojoj zatreba funkcionalnost implementirana u nekoj komponenti treba samo dodati instancu te komponente svojim svojstvima. Pojedine komponente tada predstavljaju jednu samostalnu domenu. Korištenje komponenti povećava fleksibilnost i mogućnost ponovne upotrebe dijelova koda što pospješuje efikasnost razvoja video igara. Ipak, korištenje komponenti dodaje svojevrsnu kompleksnost projektu jer tada objekti postaju skupovi manjih objekata koje je potrebno pravilno inicijalizirati i međusobno povezati.

2.5 *Observer Pattern*

Implementirajući video igru u nekom trenutku primijetimo da imamo mnoštvo objekata koji ovise o nekim drugima, koji onda ovise o trećima ili opet o onim prvima čime tvore kružnu ovisnost. Vrlo brzo postane teško uopće popratiti sve ovisnosti u kodu, a uglavnom nije moguće jednostavno ukloniti ovisnosti jer ti objekti uistinu i moraju ovisiti jedan o drugome, što nameće i priroda video igara u kojima su objekti unutar nje u međusobnoj interakciji. Stoga je te ovisnosti potrebno ostvariti na što jednostavniji i bezbolniji način koji neće uzrokovati probleme u daljnjem proširivanju koda. Standardni primjer ovoga imamo kod igračeva korisničkog sučelja. Kada se dogodi nešto što treba rezultirati promjenom na korisničkom sučelju imamo opciju da objekti koji trebaju primijetiti promjene stalno iznova pitaju da li je došlo do promjene. Ako objekt sa svakim novim kadrom dohvaća informacije i provjerava je li došlo do promjene, a promjena se desi samo jednom ili dvaput tijekom cijele igre, tada imamo mnogo suvišnih operacija. Problem je što objekt ne može znati kada je točno došlo do promjene pa mora stalno provjeravati, ali ovo možemo izbjeći korištenjem takozvanog *Observer Pattern*-a.

Cilj *Observer Pattern*-a je da objekti sami obavijeste sve objekte koji o njima ovise o relevantnim promjenama. Objekt koji inicira promjenu naziva se subjekt (eng. *Subject*), a objekti koji ovise o njegovoj promjeni nazivaju se promatrači (eng. *Observers*). Promatrači se zapravo pretplate na obavijesti o promjenama kod subjekta.

Promotrimo sliku 2.3 na kojoj su prikazane bazne klase za subjekta i promatrača koje možemo jednostavno naslijediti i dodati im nove funkcionalnosti i specifičnije implementa-



Slika 2.3: *Observer Pattern*

cije metoda. Bazna klasa za subjekta sadrži metodu `Notify()` koja prolazi listom promatrača i na svakom poziva njegovu metodu `OnNotify()`. Subjekt također ima javne metode `Attach()` i `Detach()` kojima se promatrači mogu dodati ili izbrisati iz liste promatrača nekog subjekta. Bazna klasa za promatrača ima metodu `OnNotify()`. Dobro je da metoda `OnNotify()` prima argumente s bitnim informacijama o događaju o kojem subjekt obavještava promatrača kako bi mogao reagirati na više različitih događaja od jednog ili više subjekata.

Unreal Engine implementira takozvani dispečer događaja (eng. *event dispatcher*) koji možemo shvatiti kao realizaciju *Observer pattern*-a. U uredniku *Unreal Engine*-a možemo kreirati dispečer za koji onda možemo vezati jedan ili više događaja koji se pozovu jednom kada se pozove dispečer.

2.6 *Factory Method Pattern* i *Abstract Factory*

Video igre, pogotovo one veće i kompleksnije, i razni drugi softveri skloni su promjenama. Kako se igra razvija, tako se mijenjaju zahtjevi, donose se drugačije odluke, implementiraju se nove stvari i stoga moramo osigurati da uvođenje noviteta u naš kod nema drastične posljedice na postojeću implementaciju. Fleksibilnost je ključna pa je *hardcoding* (eksplicitno ugrađivanje vrijednosti u kod) standardno nepoželjan, kako varijabli, tako i konkretnih tipova podataka, odnosno klasa u slučaju objektno orijentiranog programiranja. Direktno korištenje konkretnih klasa koje predstavljaju vrste neke općenitije klase i koje su sklone promjenama i dodavanju novih vrsta općenito nije održivo. Tu nam u pomoć priskače takozvani *Factory Method Pattern* koji spada u skupinu obrazaca stvaranja jer služi za apstrahiranje procesa kreacije objekata.

Objasnit ćemo ovaj koncept na pojednostavljenom primjeru klasa korištenim u poglavlju 4. Recimo da imamo klasu koja predstavlja čarobnjački štap - klasa `Staff` koja u različitim trenucima treba stvoriti magične projektile različitih tipova - instance klasa `ProjectileA` i `ProjectileB`. Različiti tipovi znače različitu vizualnu reprezentaciju (veličinu, boju, animaciju), različitu štetu (eng. *damage*), drugačiji inicijalni smjer i u

nekim slučajevima, različit broj ispaljenih projektila u jednom trenutku. Pretpostavimo da imamo enumerator u kojem su navedene sve različite klase projektila, te je vrsta projektila kojeg sljedeće treba stvoriti spremljena unutar varijable `NextProjectile`. Tada bi prvotno mogli napisati kod sljedećeg oblika:

```
1  void Staff::Cast()
2  {
3      switch(NextProjecile)
4          case ProjectileA:
5                  // stvori, inicijaliziraj i usmjeri projektil
6                  // tipa A koji će putovati zrakom i nanositi
7                  // štetu neprijateljima na koje naiđe
8                      break;
9          case ProjectileB:
10                 // stvori, inicijaliziraj i usmjeri nekoliko projektila
11                 // tipa B od kojih svaki ima drugačiji smjer te koji će
12                 // putovati zrakom i nanositi štetu neprijateljima
13                 // na koje naiđu
14                 break;
15 }
```

Isječak koda 2.5: Klasa za stvaranje projektila

Očito je da ako u budućnosti budemo dodavali još vrsta projektila da će kod postajati sve glomazniji i nepregledniji, a čarobnjački štap mora uvijek znati koje su mu točno vrste projektila dostupne, te ovisno o tipu projektila izvršiti određeni dio koda unutar svoje metode. Ovakvo programiranje nikako nije održivo, srećom možemo ovaj pristup unaprijediti koristeći *Factory Method Pattern*. Možemo za svaki tip projektila napraviti klasu koja će imati ulogu tvornice (eng. *Factory*) projektila određenog tipa. Te klase će stvarati i inicijalizirati projektile određenog tipa. U našem slučaju, pošto se radi o magičnim projektilima, ulogu tvornice ima klasa pod nazivom `Spell` (čarolija). Instance te klase inicijaliziramo željenim vrijednostima svojstava projektila koji će biti proizvedeni pomoću nje, a projektile stvaramo pozivajući metodu `CastSpell()`. Sada bi kod mogao izgledati slično sljedećem:

```
1  void Staff::Cast()
2  {
3          switch(NextProjecile)
4          case ProjectileA:
5                  SpellA->CastSpell()
6                      break;
7          case ProjectileB:
8                  SpellB->CastSpell()
```

```
                break;
}
```

Isječak koda 2.6: Klasa za stvaranje projektila preko tvornica

Kod već izgleda značajno preglednije, ali još uvijek sadrži `switch-case` naredbu, tj. u klasi `Staff` još uvijek zasebni slučajevi ovise o konkretnom tipu projektila, samo što sada u svakom slučaju koristimo drugu konkretnu implementaciju *Factory* klase (`SpellA`, `SpellB`), što bismo voljeli izbjeći. To možemo ako koristimo takozvani *Abstract Factory* oblikovni obrazac. Kako bi u `Staff` klasi podržali mnoge različite tipove *Factory* klase, i njen kod potpuno odvojili od stvaranih projektila, definiramo *Abstract Factory* klasu `SpellBase`. `SpellBase` nam omogućava da potklase odlučuju o tome koji će se projektil stvarati, odnosno da klasa `Staff` više ne ovisi o različitim `Projectile` klasama, niti o različitim `Spell` klasama. Čarobnjačkom štapu je sada dovoljno da zna za klasu `SpellBase` i kod može izgledati slično sljedećem:

```
class Staff
{
public:
    SpellBase* Spell;
    void Cast();
};

void Staff::Cast()
{
        Spell->CastSpell();
}

class SpellBase
{
public:
    virtual void CastSpell();
};

class SpellA : public SpellBase
{
public:
    void CastSpell();
    // vlastita implementacija ove metode
};
```

Isječak koda 2.7: Stvaranje projektila pomoću apstraktne tvornice

Kada bi htjeli unutar klase `Staff` spremiti stvoreni projektil, ali zadržati ovu razinu apstrakcije, to bi mogli postići time da dodamo apstrakciju i za klase projektila, klasu `ProjectileBase`, koju onda konkretni tipovi projektila nasljeđuju. Također u tom slučaju metode `CastSpell()` *Factory* klasa trebaju vraćati pokazivač na tip `ProjectileBase`. Tada bi klasa `Staff` i njena metoda `Cast()` klase `Staff` izgledale ovako:

```
1  class Staff
2  {
3  public:
4      SpellBase* Spell;
5      void Cast();
6  private:
7      ProjectileBase* Projectile;
8  };
9
10 void Staff::Cast()
11 {
12     Projectile = Spell->CastSpell();
13 }
```

Isječak koda 2.8: Spremanje projektila korištenjem bazne klase

Glavna prednosti korištenja ovog oblikovnog obrasca jest što kada želimo ubaciti novu klasu projektila, potrebno je samo dodati novu klasu koja nasljeđuje klasu `SpellBase` i klasa `Staff` može bez ikakvih promjena koristiti tu novu klasu.

2.7 *Prototype*

Ovaj oblikovni obrazac spada pod obrasce stvaranja i blizak je *Factory Method* obrascu. Svrha Prototip (eng. *Prototype*) oblikovnog obrasca je stvaranje novih objekta iz nekog postojećeg objekta kojeg nazivamo prototip, tj. njegovim kloniranjem. Omogućava instanciranje bez znanja o točnom tipu objekta.

Recimo da imamo baznu klasu `EnemyBase` za neprijatelje u video igri te nekoliko klasa neprijatelja koji nasljeđuju tu klasu: `EnemyA`, `EnemyB` itd. Želimo moći stvarati instance tih klasa pa definiramo klasu `SpawnerBase` koja sadrži virtualnu metodu `spawnEnemy()` koju implementiraju klase koje naslijede `SpawnerBase` klasu. Neka su to klase `SpawnerA` i `SpawnerB` koje u vlastitoj implementaciji metode `spawnEnemy()` stvaraju instance klase `EnemyA`, odnosno `EnemyB`, i vraćaju novo stvorenu instancu.

```
class SpawnerA: public SpawnerBase
{
public:
        virtual EnemyBase* spawnEnemy()
        {
                return new EnemyA();
        }
};
```

Isječak koda 2.9: Klasa za stvaranje neprijatelja određenog tipa

Ovakvim pristupom sa svakim novim tipom neprijatelja trebamo dodati i novi tip klase za stvaranje neprijatelja i imamo puno dupliciranog koda. *Prototype* oblikovni obrazac nalaže da klasi `EnemyBase` dodamo virtualnu metodu `clone()`. Metoda `clone()` klonira objekt nad kojim je pozvana i vraća taj objekt. Ovakvim pristupom imamo samo jednu klasu za stvaranje neprijatelja koja sadrži pokazivač na bazni tip neprijatelja. Taj pokazivač pokazuje na instancu klase `EnemyBase` ili na instancu bilo koje od njenih potklasa. Klasa `Spawner` u metodi `spawnEnemy()` poziva metodu `clone()` na toj instanci.

```
class Spawner
{
public:
    EnemyBase * spawnEnemy()
    {
        return Prototype->clone();
    }
    EnemyBase * Prototype;
};
```

Isječak koda 2.10: Klasa za stvaranje neprijatelja kloniranjem

Sada samo promjenom varijable Prototype možemo stvarati različite tipove neprijatelja. Koristeći *Prototype* oblikovni obrazac objekte je dovoljno kreirati samo jednom, a za daljnje instanciranje kloniraju se postojeći objekti. Ipak još uvijek imamo dosta ponavljajućeg koda jer svaki tip neprijatelja implementira vlastitu metodu `clone()`, koja ne mora nužno biti trivijalna.

Unreal Engine koristi svojevrsne prototipe pod nazivom *Class Default Object*, skraćeno *CDO*, koji se koriste prilikom kreiranja novih objekta. Više o ovome nalazi se u potpoglavlju 3.5.2.

Poglavlje 3

Unreal Engine

3.1 Uvod

Unreal Engine se prvi put pojavio 1998. godine kada je tvrtka *Epic Games, Inc.* izdala svoju video igru *Unreal*. Iako je razvijen primarno za 3D igre u prvom licu, može se koristiti za razvoj igara u gotovo svim žanrovima, a u novije vrijeme proširio se i na filmsku industriju. *Unreal Engine* je iznimno moćan upravitelj igara poznat po opsežnom skupu alata i urednika (eng. *editors*) koji pomažu u razvoju video igara. Kao i većina ostalih upravitelja igara, napisan je u jeziku *C++*.

3.2 *Unreal Engine 5*

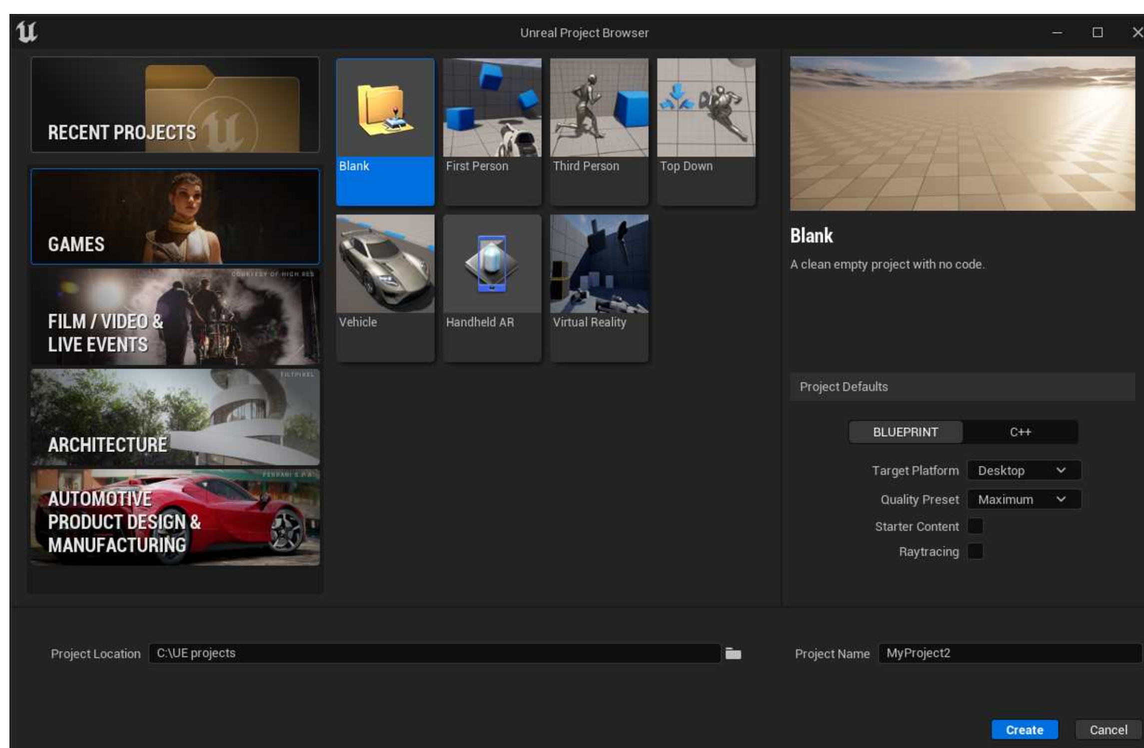
Najnovija verzija, *Unreal Engine 5*, izdana je u travnju 2022. i nju koristimo u ovom radu. *Unreal Engine 5* uvodi brojne novitete od kojih su najznačajniji *Lumen* i *Nanite*.

Epic Games opisuje *Lumen* kao potpuno dinamičan sustav za globalno osvjetljenje i refleksiju koji instantno reagira na promjene scene i svjetla. *Lumen* imitira ponašanje svjetla u stvarnom životu - svjetlo osim što obasjava sve površine vidljive iz njegova izvora, ono se i odbija od površine. Globalnim osvjetljenjem nazivamo odbijanje svjetlosti od grubu površinu u svim smjerovima, a refleksijom nazivamo odbijanje svjetlosti od glatku površinu. *Lumen* s ovakvim pristupom omogućava stvaranje dinamičnijih i iznimno realnih scena.

Nanite je sustav virtualizirane mikropoligonalne geometrije koji podržava automatsku razinu detalja. Prikazuje samo detalje koji su trenutno vidljivi te prikazuje više detalja na objektima koji su bliži kameri, a manje na onima koji su udaljeni. *Nanite* stoga omogućuje prikaz iznimno velike razine detalja bez značajnih gubitaka performansi.

3.2.1 Kreiranje projekta u *Unreal Engine 5*

Prilikom kreiranja novog projekta, *Unreal Engine* nam nudi nekolicinu predložaka koji sadrže neke bazne funkcionalnosti tipične za pojedini žanr igrice. Tako osim što možemo odabrati potpuno prazan projekt, na izbor imamo primjerice predložak za igre u prvom licu koji sadrži lik igrača reprezentiranog rukama vidljivima iz perspektive prvog lica te u kojem su implementirane osnovne kretnje i akcije lika. Osim predložaka koji služe za lakši početak razvoja video igre u pojedinom žanru, tu su i predlošci za razne druge namjene poput filma, arhitekture, dizajna proizvoda i slično.



Slika 3.1: Kreiranje projekta i predlošci u *Unreal Engine 5*

Kada kreiramo novi projekt generira se istoimeni direktorij u kojem se nalazi sam *Unreal Engine* projekt u obliku *.uproject* datoteke zajedno s raznim drugim bitnim direktorijima i datotekama (npr. *Binaries*, *Config*, *Content*, *Source*).

Također, automatski se kreira i osnovni nivo (eng. level) čiji sadržaj varira ovisno o odabranom predlošku. Nivo, ponekad nazivan i mapa, je dio virtualnog svijeta igre i sadrži sve što igrač može vidjeti i s čime može međudjelovati. Video igre su obično podijeljene u više nivoa te se tijekom igre prelazi iz jednog nivoa u drugi. Kolekcija nivoa spremljena je unutar objekta pod nazivom *Svijet* (eng. *World*). On je zadužen za prikazivanje nivoa,

za dinamičko stvaranje aktera i još mnogo toga. U svakom trenutku je u memoriji učitan samo jedan, trenutni nivo.

Nakon što je projekt stvoren, otvori se urednik nivoa i možemo započeti s razvojem video igre.

3.2.2 Razvojno okruženje *Unreal Engine*-a

Na slici 3.2 prikazan je urednik nivoa u *Unreal Engine 5*. U središtu se nalazi prozor za prikaz trenutnog sadržaja nivoa (eng. *viewport*). To je „prozor u virtualni svijet" video igre koju razvijamo. Pomoću njega se možemo kretati kroz nivo te manipulirati objektima unutar nivoa. Na vrhu se nalazi izbornik, a ispod njega osnovna alatna traka koja sadrži prečace za neke osnovne alate te naredbe za pokretanje igre, pauziranje i slično. S desne strane urednika nalazi se takozvani *outliner* u kojem je hijerarhijski poredan sav sadržaj trenutno učitanog nivoa. Na donjem dijelu urednika nalazi se ladica sa sadržajem (eng. *content drawer*) u kojoj se nalaze svi dostupni resursi - sav *C++* kod, svi dostupni materijali, teksture, animacije, zvukovi itd.

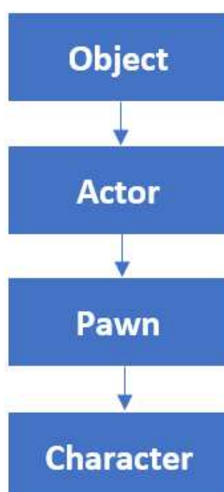
Slika 3.2: Urednik nivoa u *Unreal Engine 5*

S desne strane, ispod *outliner*-a, nalazi se kartica s detaljima trenutno označenog objekta u kojoj možemo uređivati njegova svojstva. Njen sadržaj ovisi o trenutno označenom

objektu i bit će drugačiji za različite vrste objekata. Ako je primjenjivo za trenutni objekt, u kartici detalja možemo mijenjati njegovu lokaciju, rotaciju i veličinu, dodijeliti mu geometrijsku mrežu i materijale, omogućiti mu simulaciju fizike, određivati mu ponašanje prilikom kolizije s ostalim objektima i još mnogo toga.

3.2.3 Klase u *Unreal Engine*-u

Klase u *Unreal Engine*-u organizirane su hijerarhijski, odnosno svaka klasa nasljeđuje neku drugu. U *Unreal Engine* dostupan nam je niz klasa koje možemo naslijediti prilikom kreiranja nove klase te ćemo ovdje navesti one najčešće korištene.



Slika 3.3: Hijerarhija klasa u *Unreal Engine*-u

Klasa *Object* je bazna klasa koju nasljeđuje većina ostalih klasa unutar *Unreal Engine*-a. Sadrži neke osnovne funkcionalnosti potrebne *Unreal Engine*-u, o čemu će više biti rečeno kasnije u ovom poglavlju. Klasa *Object* ne sadrži funkcionalnosti potrebne da bi se instance klase mogle postaviti unutar nivoa. Za to je pak korisna druga klasa, klasa *Actor*, odnosno akter, koja nasljeđuje i proširuje klasu *Object*. Klasa *Actor* dolazi s raznim metodama i varijablama koje su potrebne svim objektima koji se postavljaju unutar svijeta kao što je mijenjanje lokacije aktera te njegovo stvaranje i uništavanje. Klasa *Pawn*, odnosno pijun, nasljeđuje klasu *Actor* te ima dodatnu sposobnost da objekte ove klase mogu kontrolirati igrači ili umjetna inteligencija, odnosno kažemo da ima sposobnost biti „zaposjednuta“ (eng. *possessed*). Klasa *Character*, odnosno lik, nasljeđuje klasu *Pawn* i namijenjena je da ju „zaposjedne“ igrač. Prvenstveno je namijenjena za vertikalno orijentirane likove koji imaju sposobnost hodanja, trčanja, skakanja i slično. Nadalje, u većini projekata standardno se koristi i klasa *PlayerController* koja je odgovorna za kontroliranje

lika kojim upravlja igrač te klasa *Game Mode* koja definira pravila igre, bodovanje u igri, uvjete za pobjedu i poraz i slično. Svaki nivo može imati samo jednu *GameMode* klasu.

3.2.4 Komponente u *Unreal Engine*-u

Osnovni tipovi komponenti koji su nam na raspolaganju u *Unreal Engine*-u su *Actor Component*, *Scene Component* i *Primitive Component*. *Actor Component* je bazna klasa za ostale komponente. Koristimo je uglavnom za neke apstraktne funkcionalnosti poput kretanja i ostalih funkcionalnosti koje nemaju fizičku reprezentaciju. *Scene Component* nasljeđuje *Actor Component* te dodatno ima fizičku lokaciju i rotaciju u svijetu. Također, ovu komponentu možemo pridruživati drugim komponentama ovog tipa, što s *Actor Component* ne možemo. Nadalje, Primitive Component nasljeđuje *Scene Component*, a razlikuje ih to što *Primitive Component* ima geometrijsku reprezentaciju, a *Scene Component* nema.

Jedna od komponenti koja je obavezno prisutna u svakoj video igri je *Static Mesh Component*, naslijeđena iz *PrimitiveComponent*, koja predstavlja standardnu poligonalnu mrežu koju dodajemo akterima kako bi im dali vizualnu reprezentaciju u virtualnom svijetu. Ovu komponentu najčešće koristimo za zidove, podove, dekoracije i slično, dok za predstavljanje humanoidnih likova koristimo *Skeletal Mesh Component* koja predstavlja poligonalnu mrežu kojoj je pridružen kostur te je njena osnovna značajka što se može koristiti u animacijama. Klasa *Character*, spomenuta u prethodnom potpoglavlju, originalno sadrži *Skeletal Mesh Component*. Također, klasa *Character* dolazi i s *Capsule Component*-om koja predstavlja volumen u obliku kapsule koji služi za jednostavniju detekciju kolizija. Naime, kada bi za detekciju kolizija koristili samu geometrijsku mrežu koju koristimo za izgled lika, računanje kolizija bi bilo suviše sporo jer su te mreže često sačinjene od jako velikog broja trokuta. Upravo zbog toga često koristimo pojednostavljene oblike za detektiranje kolizija kao što su kapsula, sfera, kocka i slično. Klasa *Character* ima i *Character Movement Component*-u koja sadrži neke dodatne funkcionalnosti kretanja koje ne mogu imati ostali tipovi aktera. Postoji još mnogo preddefiniranih komponenti koje su nam na raspolaganju, a možemo i kreirati vlastite.

3.3 *Blueprint* i *C++*

Za razvoj video igre unutar *Unreal Engine*-a imamo mogućnost koristiti programski jezik *C++*, u kojem je i sam *Unreal Engine* napisan, te vizualni skriptni jezik pod nazivom *Blueprint*. *Blueprint* je jezik napravljen posebno za *Unreal Engine*, a baziran je na čvorovima i njihovom međusobnom povezivanju. Postavlja se pitanje što odabrati za razvoj igre, *Blueprint* ili *C++*, i trebamo li uopće birati. U nastavku slijede neke prednosti korištenja *Blueprint*-a, a zatim prednosti korištenja jezika *C++*.

Najveća prednost *Blueprint*-a je upravo to što je napravljen baš za *Unreal Engine* pa ga možemo koristiti direktno iz urednika te se promjene koje napravimo gotovo instantno primjene na igru. S druge strane, za *C++* moramo koristiti neko drugo razvojno okruženje te ga sa svakom promjenom moramo ručno kompajlirati (prevesti u strojni jezik) kako bi promjene postale vidljive u uredniku. Iz ovog razloga je u *Blueprint*-u jednostavnije namještati i isprobavati različite vrijednosti svojstava pojedinih objekata. *Blueprint* je jednostavan za početnike, a isto tako ga jednostavno mogu koristiti i dizajneri koji rade na razvoju video igre što omogućava efikasniju raspodjelu posla među članovima razvojnog tima. Nadalje, u *Blueprint*-u možemo jako brzo doći do nekog funkcionalnog stadija igre stoga je on odličan za prototipiranje i za brzo iteriranje. Kako je *Blueprint* vizualni jezik, lakše je predočiti i pratiti tok igre nego u *C++* izvornom kodu, pogotovo ako koristimo asinkrone funkcijske pozive.

Prva i glavna prednost *C++*-a je to što *C++* ima bolje performanse. To je zbog toga što prilikom kompajliranja *C++* koda u strojni jezik dobijemo generalno manje instrukcija koje se moraju izvršiti na procesoru nego prilikom kompajliranja *Blueprint*-a zato što se *Blueprint* ne prevodi direktno u strojni jezik. Znamo da video igre koriste dosta matematike, a složene matematičke operacije mogu biti komplicirane i spore za izvođenje u *Blueprint*-u, stoga ih je bolje pisati u *C++*-u. Također je bitna prednost to što s *C++*-om imamo pristup svim funkcionalnostima *Unreal Engine*-a, dok *Blueprint* ima ograničeni pristup, stoga s *C++*-om imamo veću fleksibilnost u razvoju video igre i možemo ju bolje prilagoditi vlastitim potrebama. Nadalje, s *C++*-om možemo imati bolje organiziran kod pa je on bolji za korištenje u velikim projektima gdje *Blueprint* može vrlo brzo postati nepregledan jer bi imali ogromne grafove s hrpom čvorova i bilo bi teško pronaći pojedine dijelove implementacije. Također, *C++* je industrijski standardiziran jezik pa ga velik broj ljudi već dobro poznaje.

Iako u većini slučajeva možemo sve potrebno implementirati i u *Blueprint*-u i u *C++*-u bez značajnih razlika u performansama, pitanje zapravo nije koristiti *C++* ili *Blueprint*, nego gdje koristiti *C++*, a gdje *Blueprint*, kako bi imali najbolje od oba svijeta. *C++* je generalno bolji za programiranje sustava niže razine, za izgradnju temelja koji čine igru, dok je *Blueprint* dobro koristiti za dogradnju tih sustava na višoj razini i za kozmetičke detalje. Ovo je standardna podjela, ali to nikako nije strogo pravilo i *Unreal Engine* nam ne postavlja granice po pitanju toga za što trebamo koristiti *C++*, a za što *Blueprint*. Stoga svatko ima slobodu odlučiti kako želi pristupiti razvoju video igre ovisno o svojim mogućnostima i iskustvu.

3.4 Korištenje *Blueprint*-a u *Unreal Engine*-u

3.4.1 Tipovi *Blueprint*-a

Postoji pet tipova *Blueprint*-a koje možemo koristiti u Unreal Engine-u. *Blueprint* nivoa (eng. *Level Blueprint*) se koristi za upravljanje globalnim događajima unutar nivoa i postoji samo jedan za svaki nivo. *Blueprint* klase (eng. *Blueprint Class*), kojeg često nazivamo samo *Blueprint*, koristimo za dodavanje funkcionalnosti akterima koje postavljamo unutar nivoa. *Blueprint* klasa može naslijediti drugu *Blueprint* klasu ili pak *C++* klasu. Zatim imamo podatkovni *Blueprint* (eng. *Data-Only Blueprint*). To je *Blueprint* klasa koja sadrži sva svojstva i komponente klase koju nasljeđuje s time da se svojstva mogu modificirati, ali se ne mogu dodavati novi elementi. Potom imamo *Blueprint* sučelje (eng. *Blueprint Interface*), kolekciju imena jedne ili više funkcija bez implementacije. Sučelje se može dodati drugim klasama koje onda moraju implementirati funkcije navedene u njemu. I zadnje imamo *Blueprint* makro (eng. *Blueprint Macro*) kojeg kreiramo kao zaseban graf kojem možemo definirati ulazne i izlazne vrijednosti. Zatim možemo stvoreni *Blueprint* makro koristiti u drugim *Blueprint*-ima pomoću čvora koji predstavlja taj makro. Kada dođe red na izvršavanje tog čvora izvršit će se čitav graf definiran unutar pripadnog makro-a.

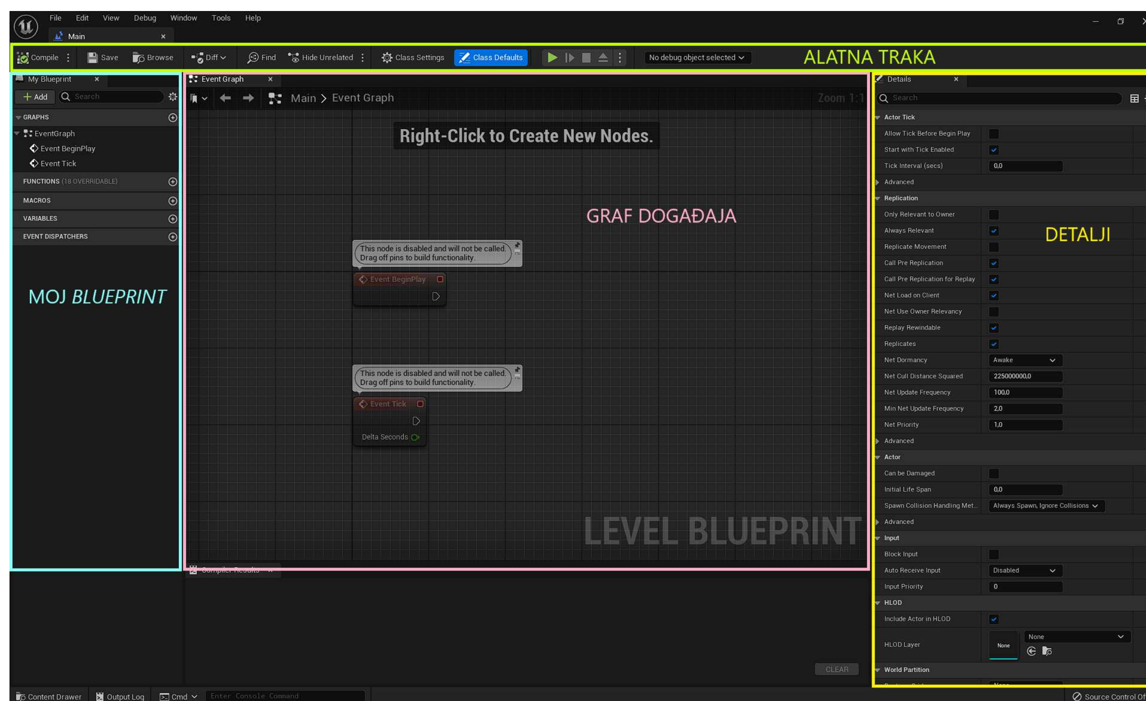
3.4.2 Urednik za *Blueprint*

Na slici 3.4 se nalazi urednik za *Blueprint* nivoa. Može se vidjeti da na vrhu postoji alatna traka s raznim korisnim alatima. S lijeve strane nalazi se kartica pod nazivom *Moj Blueprint* (eng. *My Blueprint*) preko koje se upravlja s grafovima, funkcijama, makroima, varijablama i analogonima C++-ovih delegata - dispečerima (eng. *dispatchers*). S desne strane nalazi se kartica detalja preko koje možemo uređivati svojstva trenutno selektiranog sadržaja, slično kao i na kartici detalja sa slike 3.2, a u sredini se nalazi graf događaja (eng. *Event Graph*) koji služi za skriptiranje. Desnim klikom na prazno područje unutar okvira za graf događaja otvara nam se kontekstualni izbornik u kojem su izlistane sve funkcije koje trenutno možemo koristiti.

3.4.3 Graf događaja

Graf događaja se sastoji od čvorova (eng. *nodes*). Čvorovi su vizualne reprezentacije događaja, funkcija i varijabli, a različiti tipovi čvorova predstavljeni su različitim bojama.

Čvorovi imaju priključke preko kojih ih povezujemo s drugim čvorovima. Postoje dvije vrste priključaka - priključci za izvršavanje i podatkovni priključci. Priključci za izvršavanje (eng. *execution pin*) označeni su bijelim trokutom pri vrhu čvora. Ulazni priključak za izvršavanje nalazi se s lijeve strane i on nam govori kada se čvor treba izvršiti,



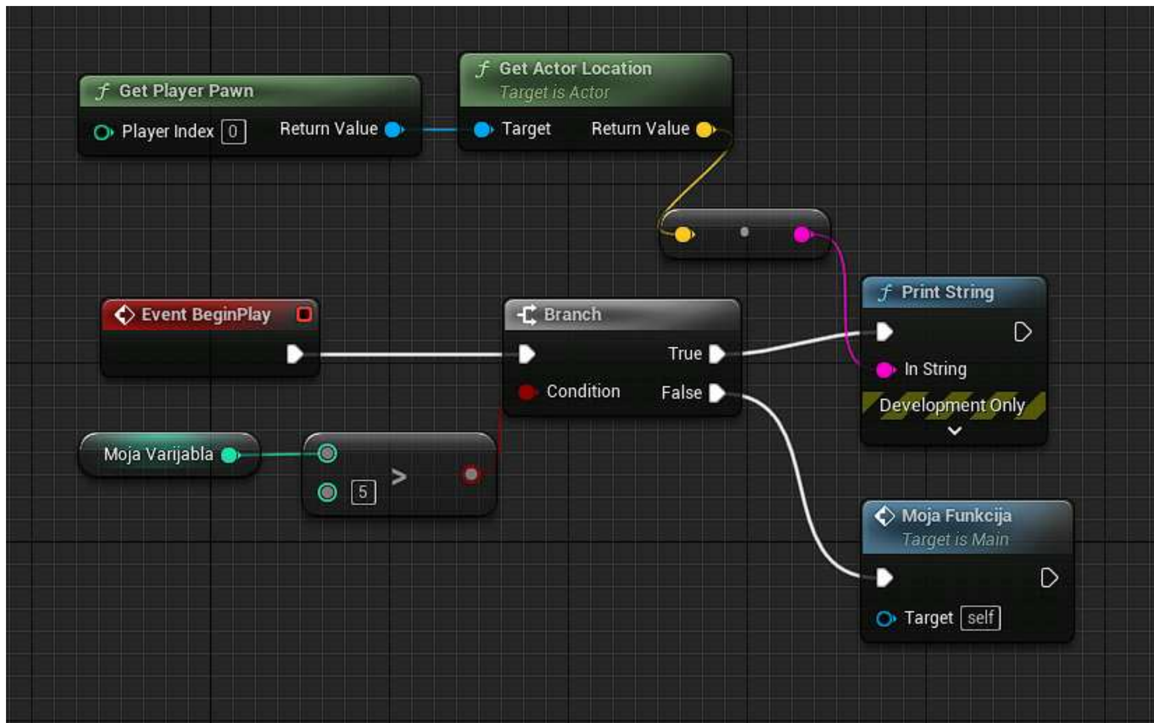
Slika 3.4: Urednik za *Blueprint* nivoa

a izlazni priključak se nalazi s desne strane i on nam govori što treba napraviti nakon što čvor obavi svoj posao. Ako želimo da se čvor B izvrši nakon što se izvrši čvor A, onda trebamo izlazni priključak za izvršenje čvora A povezati s ulaznim priključkom za izvršenje čvora B.

Čvorovi mogu imati i podatkovne priključke koji mogu biti ulazni i izlazni. Ulazni podatkovni priključci nalaze se s lijeve strane čvora i služe kao parametri odnosno za dohvaćanje podataka, a izlazni se nalaze s desne strane čvora i služe za vraćanje podataka, tj. predstavljaju povratnu vrijednost.

Takozvane čiste funkcije (eng. *pure functions*) nemaju priključke za izvršenje. One ne mijenjaju stanje objekata i izvode se po potrebi, odnosno kad god nam zatreba izlazni podatak. Na slici 3.5 vidimo da su takve funkcije primjerice *Get Player Pawn*, *Get Actor Location* i operator usporede „veće od". Čiste funkcije uglavnom koristimo za dohvaćanje objekata ili svojstava, kao što smo u primjeru na slici dohvatili lokaciju igračevog pijuna, te za odgovaranje na pitanja poput da li je varijabla veća od pet.

Na slici 3.5 također vidimo da u *Blueprint*-u možemo dodavati vlastite varijable i funkcije.



Slika 3.5: Primjer *Blueprint*-a

3.4.4 Događaji

U *Blueprint*-u postoje posebne funkcije koje se zovu događaji (eng. *events*). Događaji i reakcije na događaje velik su dio većine video igara i po tome što je programiranje u *Blueprint*-u bazirano na događajima je graf događaja i dobio naziv. Postoje razni standardni događaji koji se koriste u *Unreal Engine*-u od kojih ćemo mi ovdje navesti dva najvažnija: *BeginPlay* i *Tick*.

BeginPlay je prvi događaj koji se pozove kada se stvori instanca Blueprint klase i koristimo ga za inicijalizaciju.

Tick je događaj koji se poziva sa svakim kadrom. *Tick* se poziva s parametrom *DeltaTime* koji sadrži vrijeme koje je prošlo od iscrtavanja prethodnog kadra i omogućava implementaciju igre neovisne o broju kadrova po sekundi.

U *Blueprint*-u također možemo dodavati vlastite događaje te koristiti već spomenute dispečere događaja za koje možemo vezati jedan ili više događaja koji se pozovu jednom kada se pozove dispečer. Dispečer se može pozvati i iz *Blueprint*-a nivoa čime ostvarujemo komunikaciju između *Blueprint*-a klase i *Blueprint*-a nivoa.

3.5 Korištenje *C++*-a u *Unreal Engine*-u

Svaka *C++* klasa u *Unreal Engine*-u sastoji se od datoteke zaglavlja (*.h*) u kojoj navodimo deklaraciju klase, njezinih svojstava i metoda te od datoteke izvornog koda (*.cpp*) u kojoj se nalaze implementacije metoda navedenih unutar datoteke zaglavlja. Nakon što dodamo projektu novu C++ klasu iz nje možemo generirati *Blueprint* klasu koju zatim možemo proširivati unutar urednika.

Tipovi unutar *C++* koda koji pripadaju *Unreal Engine*-u lako se primjećuju jer standardno započinju prefiksom u obliku jednog velikog tiskanog slova. Tako primjerice prefix U imaju klase koje nasljeđuju `UObject`, dok prefiks A označava aktera i njega imaju sve klase koje nasljeđuju klasu `AActor`, dakle klase koje predstavljaju objekte koji se mogu postaviti unutar nivoa. Predlošci klasa (eng. *template classes*) imaju prefiks T, sučelja (eng. *interface*) imaju prefiks I, enumeratori prefiks E i slično.

Programiranje s *C++*-om u *Unreal Engine*-u je slično standardnom programiranju u *C++*-u, ali sučelje za programiranje aplikacija (*application programming interface, API*) koje pruža *Unreal Engine* znatno pojednostavljuje kodiranje. *Unreal Engine C++ API* optimiziran je za razvoj video igara i mnoštvo stvari koje nam mogu zatrebati i koje se često pojavljuju u razvoju video igara, *Epic Games* je već implementirao u svom API-ju. Ipak, postoje bitne razlike između programiranja s *C++*-om u *Unreal Engine*-u i standardnog programiranja s *C++*-om.

3.5.1 *Unreal Engine* i *C++* standardna biblioteka

Ono što se na prvu može činiti pomalo neobično jest to što *Unreal Engine* uglavnom ne koristi *C++* standardnu biblioteku (eng. *standard library, stdlib*). Razlog je prvenstveno povijesni jer u doba kada je *Unreal Engine* nastajao, *C++* standardna biblioteka je bila spora i nestabilna pa nije mogla udovoljiti zahtjevima koje su postavljale video igre. Stoga je *Epic Games* izbjegavao korištenje standardnih *C++* biblioteka i koristio vlastite implementacije tipova i metoda kako bi imao veću kontrolu nad memorijom, performansama i mogućnostima. *Unreal Engine* koristi vlastite tipove spremnika (`TArray`, `TMap`, itd.) i vlastiti tip za nizove znakova (`FString`) umjesto standardnih *C++* tipova. Većina tipova je analogna onima iz *C++* standardne biblioteke i koriste se na sličan način. *Unreal Engine* također koristi i vlastite biblioteke za razne matematičke operacije, operacije nad vektorima, nad stringovima itd. Situacija se u novije vrijeme polako mijenja jer je danas standardna biblioteka jako stabilna i optimizirana te *Unreal Engine* počinje sve više koristiti neke njezine elemente. Tako se primjerice preporučuje korištenje atomskih tipova iz standardne biblioteke iz zaglavlja `<atomic>`, radije nego *Unreal Engine*-ova implementacija `TAtomic`. Također se na primjer mogu koristiti sve metode nad brojevima s pomičnom točkom iz zaglavlja `<cmath>`, ali još uvijek je preporučeno izbjegavanje sprem-

nika i string-ova iz standardne biblioteke. Više o aktualnim standardima kodiranja s *C++*-om u *Unreal Engine*-u može se pročitati u službenoj dokumentaciji *Unreal Engine*-a ([6]).

3.5.2 *UObject* i refleksijski sustav *Unreal Engine*-a

`UObject` je bazna klasa za sve objekte unutar *Unreal Engine*-a. Kada kreiramo novu klasu koja nasljeđuje neku postojeću *Unreal Engine* klasu kao što je na primjer klasa *Actor*, *ActorComponent*, *Pawn* itd., ona na svom hijerarhijskom putu nasljeđuju klasu `UObject`. To je važno zbog toga što je `UObject` dio refleksijskog sustava *Unreal Engine*-a, čiji opis slijedi u nastavku.

U programiranju, izraz refleksija ili refleksijsko programiranje označava introspekcijsku sposobnost programa, odnosno sposobnost programa da pregleda i modificira sam sebe tijekom izvođenja. *C++* nema ugrađeni refleksijski sustav, stoga *Unreal Engine* koristi vlastiti. Refleksijski sustav *Unreal Engine*-a (*Unreal Engine Reflection System*) je sustav za sakupljanje i manipuliranje informacijama izvučenim iz *C++* koda. Refleksijski sustav je iznimno značajan dio *Unreal Engine*-a zaslužan za komunikaciju između *Blueprinta* i *C++*-a, sakupljanje smeća, serijalizaciju i mrežno repliciranje. Kako bi primjerice klase, svojstva ili metode iz *C++* koda bili dostupni refleksijskom sustavu, potrebno ih je u datoteci zaglavlja označiti posebnim makro-ima kao što je `UCLASS()` za klase, `UPROPERTY()` za svojstva, `UFUNCTION()` za funkcije itd.

Unutar zagrada navedenih makro-a možemo dodavati razne specifikatore, odnosno ključne riječi kojima definiramo razinu izlaganja unutar urednika ili *Blueprint*-a te način na koji će se ponašati kroz različite aspekte *Unreal Engine*-a. Ovih specifikatora ima mnogo, a neki najčešće korišteni su: `EditAnywhere` kojim označavamo da se svojstvu možemo mijenjati vrijednost unutar urednika (na panelu detalja), `VisibleAnywhere` kojim označavamo da je svojstvo vidljivo unutar urednika, ali se ne može tamo i mijenjati, `BlueprintReadWrite` kojim označavamo da se svojstvo može čitati i uređivati unutar *Blueprint*-a, `BlueprintReadOnly` kojim označavamo da se svojstvo može čitati, ali ne i uređivati unutar *Blueprint*-a, `BlueprintCallable` koji označava da metodu možemo pozivati unutar *Blueprint*-a, `BlueprintPure` koji označava da metodu možemo pozivati unutar *Blueprint*-a i da ona ne mijenja objekt koji ju posjeduje.

Refleksijski sustav omogućuje korištenje raznih značajki za manipuliranje objektima. Jedna korisna značajka jest to što kada izbrišemo ili dodamo novo svojstvo koje je označeno s `UPROPERTY()` makro-om, učitavanje postojećeg sadržaja odvija se bez problema tako što se novim svojstvima dodjele standardne vrijednosti, a uklonjena svojstva se jednostavno ignoriraju. Također, implementirana je automatska inicijalizacija što znači da se sva svojstva klase koja nasljeđuje `UObject`, bili oni predani refleksijskom sustavu preko `UPROPERTY()` makro-a ili ne, nuliraju prije poziva samog konstruktora u kojem se potom mogu dodjeljivati vlastite početne vrijednosti svojstava.

Primjer datoteke zaglavlja jedne klase prikazan je u isječku koda

```
#pragma once

#include "GameFramework/Pawn.h"
#include "EnemyBase.generated.h"

UCLASS()
class SURVIVALWIZARDRY_API AEnemyBase : public APawn
{
    GENERATED_BODY()

public:
    AEnemyBase();

public:
    virtual void Tick(float DeltaTime) override;

protected:
    virtual void BeginPlay() override;

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    float Damage = 30.f;
};
```

Isječak koda 3.1: Primjer datoteke zaglavlja

Datoteka zaglavlja klase koja nasljeđuje `UObject` mora sadržavati određene elemente kako bi mogla pravilno funkcionirati s *Unreal Engine*-om. Ti elementi vide se na primjeru iz isječka koda 3.1, a to su: uključivanje datoteke čije se ime sastoji od imena klase i ekstenzije *.generated.h* (u primjeru je to datoteka *EnemyBase.generated.h*), `UCLASS()` makro neposredno prije definiranja klase, te `GENERATED_BODY()` makro. Svaka nova klasa koju kreiramo mora sadržavati navedene elemente ako želimo da koristi privilegije koje pruža refleksijski sustav *Unreal Engine*-a.

Takozvani *Unreal Header Tool (UHT)* prilikom kompilacije *C++* koda, točnije neposredno prije standardnog procesa kompilacije, skenira datoteke zaglavlja i pronalazi sve elemente vezane uz *Unreal Engine*. Zatim generira *C++* kod na temelju pronađenih makro-a i specifikatora te ga smješta u *.generated.h* datoteku, koju on sam i generira. Linija `#include 'EnemyBase.generated.h'` nužno mora biti posljednja u nizu naredbi `#include`.

Nadalje, `UCLASS()` makro omogućuje da klasa `AEnemyBase` bude vidljiva *Unreal Engine*-u, a `GENERATED_BODY()` makro postavlja kod iz *EnemyBase.generated.h* unutar klase kako bi ona mogla pravilno funkcionirati s *Unreal Engine*-om. U primjeru također vidimo

da je varijabla `Damage` predana refleksijskom sustavu pomoću makro-a `UPROPERTY()` te da se može dohvaćati i mijenjati u uredniku i u *Blueprint*-u.

Osim `UCLASS()` makro-a, postoji i kasa s istim imenom. `UClass` je klasa koja se koristi za refleksiju te ona također nasljeđuje `UObject` klasu. Kada klasu označimo makro-om `UCLASS()`, generira se instanca klase `UClass` za taj `UObject` koja sadrži informacije o klasi i reprezentira ju tijekom izvođenja. `UClass` objekt možemo dohvatiti pozivajući metodu `GetClass()` na instanci neke klase ili tako da pozovemo statičku metodu `StaticClass()` ako nemamo instancu. To može biti korisno primjerice kod dinamičkog kreiranja aktera u *Unreal Engine*-u gdje koristimo `StaticClass()` metodu kako bi dali do znanja od koje klase želimo stvoriti instancu.

`UClass` sadrži poseban objekt kojeg zovemo *Class Default Object (CDO)* i on predstavlja predložak objekta neke klase sa standardnim vrijednostima. Kreira se korištenjem konstruktora svaki put kada se pokrene *Unreal Engine* i preporuka je da se nakon toga više ne mijenja. Služi za kreiranje novih objekata jer se dinamički stvoreni objekti dobivaju kopiranjem *Class Default Object*-a. *Class Default Object* postoji dokle god je *Unreal Engine* pokrenut, stoga uvijek možemo po potrebi dohvaćati standardne vrijednosti neke klase. Kada se *Class Default Object* neke klase promjeni, *Unreal Engine* će prilikom učitavanja pokušati primijeniti te promjene na sve instance te klase tako da ako neko svojstvo objekta ima vrijednost jednaku staroj vrijednosti tog svojstva u *Class Default Object*-u, ažurirat će se na novu vrijednost. Ako svojstvo pak sadrži neku drugu vrijednost, pretpostavlja se da je to napravljeno namjerno i ta vrijednost se neće mijenjati.

3.5.3 Sakupljač smeća

Sakupljanje smeća (eng. *Garbage Collection, GC*) je oblik upravljanja memorijom u kojem se nastoji osloboditi memorija koju su alocirali objekti koji se više ne upotrebljavaju. *Unreal Engine* vodi bilješke o trenutnim referencama na objekte u obliku grafa i implementira sakupljača smeća koji u pravilnim intervalima pregledava taj graf i briše sve objekte koji se ne nalaze u njemu, dakle objekte bez referenci. Objekti se također mogu eksplicitno predati na uništenje pozivom metode `MarkAsGarbage()` i time će sakupljač smeća tijekom sljedeće iteracije počistiti taj objekt i nulirati sve njegove reference. Akteri se uglavnom uništavaju korištenjem metode `Destroy()` koja ukloni aktera iz liste aktera trenutnog nivoa, obavi razne druge poslove radi očuvanja stabilnosti i na kraju pozove metodu `MarkAsGarbage()`.

Postoji nekoliko vrsta pokazivača koje možemo koristiti tijekom kodiranja video igre u *Unreal Engine*-u. Prvo imamo standardni pokazivač na `UObject` označen `UPROPERTY()` makro-om, dakle koji je vidljiv refleksijskom sustavu. On sprječava da sakupljač smeća pokupi objekt na koji referencira. Objekt će se uništiti tek kada više nema takvih referenci na njega, osim ako ga eksplicitno ne predamo na uništenje i u tom slučaju sve reference na

njega koje su vidljive refleksijskom sustavu automatski se nuliraju. Dakle, nije moguće da pokazivač pokazuje na obrisani dio memorije nakon što se objekt uništi stoga je standardna provjera je li pokazivač jedak `nullptr` dovoljna za njegovo sigurno korištenje.

S druge strane, pokazivač na `UObject` koji nije vidljiv refleksijskom sustavu nije vidljiv niti sakupljaču smeća, stoga se neće automatski nulirati niti će spriječiti sakupljača smeća da uništi objekt na koji pokazuje. U ovom slučaju nećemo imati način za utvrditi valjanost pokazivača, odnosno pokazuje li on na valjanu adresu ili na adresu uništenog objekta. Standardna provjera je li pokazivač jednak `nullptr` nije dovoljna jer čak i ako pokazivač nije jednak `nullptr`, to nam ne garantira njegovu valjanost. U slučaju uništenja objekta na koji pokazuje, pokazivač će pokazivati na memoriju gdje se nekoć nalazio taj objekt, no on se tamo više ne nalazi i stoga korištenje tog pokazivača nije sigurno.

Ako ipak želimo imati pokazivač na `UObject` koji ne sprječava sakupljača smeća da uništi objekt na koji pokazuje, bolje je koristiti slabe pokazivače, koji u *Unreal Engine*-u postoje pod nazivom `TWeakObjectPtr`. Ovakav pokazivač će se automatski nulirati kada se objekt na koji pokazuje uništi. `TWeakObjectPtr` nam omogućava jednostavnu provjeru valjanost pokazivača pomoću metode `Get()` koja vraća `nullptr` ako je objekt uništen ili je pokazivač eksplicitno postavljen na null, inače vraća valjani pokazivač na `UObject`. Postoji i metoda `IsValid()` kojoj je povratna vrijednost *boolean* varijabla čija je vrijednost jednaka `true` ukoliko bi metoda `Get()` vratila valjan pokazivač, a `false` inače.

Unreal Engine ima i vlastite inačice pametnih pokazivača iz *C++11*: `TUniquePtr`, `TSharedPtr`, `TweakPtr` te `TsharedRef`, implementiranih unutar *Unreal Smart Pointer* biblioteke. Ipak, ove pokazivače ne možemo koristiti s `UObject` klasom jer ih sakupljač smeća nije svjestan.

Poglavlje 4

Survival Wizardry

4.1 Uvod

Ovo poglavlje opisuje video igru napravljenu u *Unreal Engine*-u u sklopu praktičnog dijela ovog rada. Usput će biti objašnjeni još neki pojmovi i koncepti iz *Unreal Engine*-a koji su korišteni prilikom razvoja ove video igre. U izradi video igre korištena je verzija *Unreal Engine* 5.0.3. te programski jezik *C++* u kombinaciji s *Blueprint* skriptnim jezikom. Izvorni kod može se pronaći u *Github* repozitoriju na sljedećoj poveznici: `https://github.com/MatejaT4/SurvivalWizardry.git`.

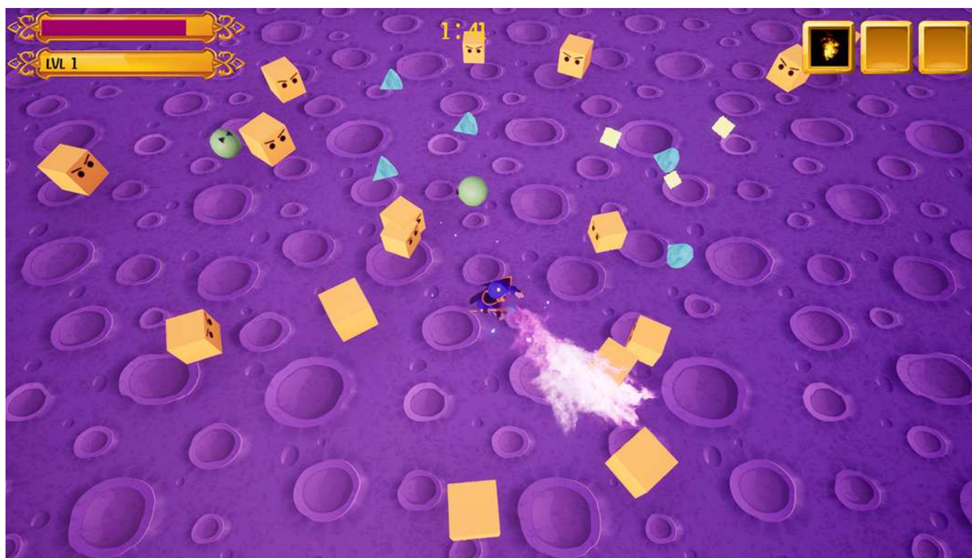
Survival Wizardry je video igra *Shoot 'em up* žanra igara u kojima se igrač bori s velikim brojem neprijatelja izbjegavajući njihove udarce. Inspiracije za ovu video igru su nezavisne video igre *Vampire Survivors* i *Brotato*.

U video igri *Survival Wizardry* igrač upravlja čarobnjakom koji pokušava preživjeti hordu neprijatelja koji se kreću prema njemu. Čarobnjak automatski baca čarolije kojima pokušava ubiti neprijatelje te skuplja *experience* pomoću kojeg diže nivo i tako dobiva nove čarolije ili poboljšava postojeće. Potrebno je paziti da neprijatelj ne dotakne čarobnjaka jer mu tako nanosi štetu. Snimke zaslona igre prikazane su na slikama 4.1 i 4.2

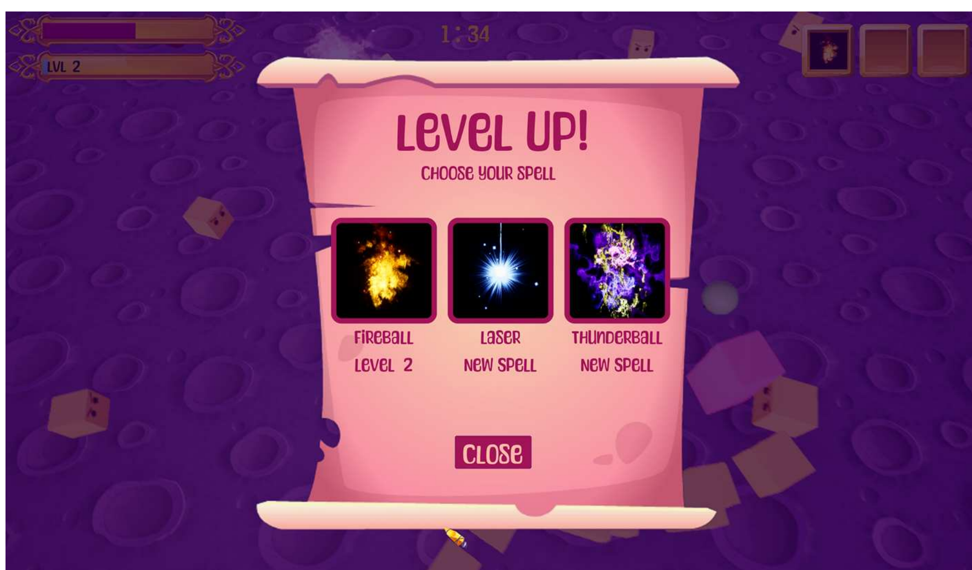
Slijedi opis pojedinih elemenata od kojih se sastoji ova video igra, njihovih *C++* klasa i pripadnih *Blueprint* klasa.

4.2 *Game Mode*

U ovom projektu, *Game Mode* je polazište implementacije logike igre, pa ćemo opisivanje klasa započeti s njim. *Game Mode* klasa za ovaj projekt implementirana je većinom u *C++* klasi pod nazivom `ASurvivalWizardryGameModeBase` koja nasljeđuje *Unreal Engine*-ovu `AGameModeBase` klasu. Potom je kreirana i *Blueprint* verzija klase bazirana na ovoj *C++*



Slika 4.1: Snimka zaslona igre *Survival Wizardry*



Slika 4.2: Snimka zaslona igre u trenutku podizanja nivoa

klasi pod nazivom `BP_SurvivalWizardryGameModeBase`, koja je u *Unreal Engine* uredniku postavljena kao *Game Mode* klasa za osnovni nivo igre, odnosno mapu, naziva *Main*. Također imamo i `MainMenuGameMode` koji će biti opisan odjeljku 4.13.

`BlueprintImplementableEvent` je specifikator za funkcije s `UFUNCTION()` makroom kojim označavamo metodu koju ćemo implementirati u *Blueprint*-u, ali ju s ovakvom

deklaracijom možemo pozivati unutar *C++* koda. Postoji i `BlueprintNativeEvent` specifikator koji dopušta da se funkcija implementira i u *C++*-u i u *Blueprint*-u. Ako je povratni tip funkcije s nekim od ovih specifikatora jednak `void`, onda ona u *Blueprint*-u zapravo predstavlja događaj koji se aktivira kada unutar *C++* koda pozovemo tu funkciju. *C++* klasa `ASurvivalWizardryGameModeBase` koristi nekoliko funkcija sa specifikatorom `BlueprintImplementableEvent` kako bi obavijestila *Blueprint* verziju klase o pojedinim događajima gdje ih ona onda koristi kako bi mijenjala korisničko sučelje. Primjer deklaracije ovakve funkcije nalazi se u isječku koda 4.1, dok se primjer poziva ove funkcije, odnosno aktiviranje događaja, može vidjeti u isječku koda 4.3 u liniji 5.

```
1 UFUNCTION(BlueprintImplementableEvent, Category="CppEvents")
2 void StartGameCountdownEvent();
```

Isječak koda 4.1: BlueprintImplementableEvent

Klasa `ASurvivalWizardryGameModeBase` čuva pokazivače na igračevog lika i kontroler (klase `AWizard` i `AMyPlayerController`) koje dohvaća u metodi `BeginPlay()` u kojoj također onesposobljava korisnikove unose dok igra ne započne pozivom metode `SetPlayerEnabledState()` na igračevom kontroleru. Nakon što se obavi potrebna inicijalizacija, na kraju `BeginPlay()` metode poziva se metoda `StartGameCountdown()` koja započinje odbrojavanje do početka igre nakon čega se poziva metoda `StartGame()` u kojoj se osposobljavaju korisnikovi unosi te emitira događaj o početku partije. Na kraju se pokrene odbrojavanje partije, odnosno sat čije se minute i sekunde postavljaju u metodi `Clock()`.

Za deklaraciju pokazivača, primjerice na igračevog lika, odnosno na instancu `AWizard` klase, dobro je koristiti takozvanu deklaraciju unaprijed (eng. *forward declaration*). Kada želimo u nekoj klasi koristiti neki drugi tip, moramo kompajleru dati do znanja gdje se on nalazi. To radimo korištenjem `#include` direktive, odnosno uključivanjem datoteke zaglavlja u kojoj je deklariran taj tip. Time naš kod postaje veći jer pretprocesor kod iz zaglavlja jednostavno prepiše tamo gdje smo ga uključili. To ne bi bio idealan rezultat unutar datoteke zaglavlja jer općenito želimo da datoteke zaglavlja budu što kraće upravo zato što ih često uključujemo u druge datoteke i time stalno širimo svoj kod i povećavamo vrijeme kompilacije. Uključivanje u datoteku zaglavlja nam nije potrebno kada samo deklariramo pokazivač, ali nam treba ako želimo konstruirati objekt ili pristupati varijablama i metodama. Dakle, u slučaju kada trebamo deklarirati samo pokazivač na neki tip možemo izbjeći uključivanje njegove datoteke zaglavlja tako što ćemo ispred deklaracije dodati ključnu riječ `class` koja govori kompajleru da je to takozvani nepotpuni tip i neće nam javljati grešku. Ipak, datoteku zaglavlja moramo uključiti u *.cpp* datoteci u kojoj ćemo koristiti taj pokazivač.

```
UPROPERTY()
class AWizard* Wizard;
```

Isječak koda 4.2: Deklaracija unaprijed

`Tick` metoda u `ASurvivalWizardryGameModeBase` klasi je onesposobljena jer su nam u ovom slučaju dovoljna dva brojača (eng. *timer*). Brojači izvršavaju zadanu *callback* funkciju s odgodom ili u određenim intervalima. Brojačima upravlja menadžer brojača koji je globalna instanca klase `FTimerManager`. Možemo ga dohvatiti pomoću metode `GetWorldTimerManager()` nakon čega pomoću metode `SetTimer()` možemo postaviti brojač. Kako bi postavili brojač potrebna nam je struktura `FTimerHandle` *Unreal Engine*-a, stoga klasi `ASurvivalWizardryGameModeBase` dodamo svojstvo tipa `FTimerHandle` koje služi kao identifikator za brojač i pomoću njega možemo brojač pauzirati, ponovno pokrenuti i slično. `FTimerHandle` je prvi argument za metodu `SetTimer()`, drugi je pokazivač na `UserClass`, odnosno objekt nad kojim pokrećemo brojač i tu možemo proslijediti `this` pokazivač. Zatim slijedi referenca na metodu koju će pokretati brojač, tj. na *callback* metodu. Sljedeći argument predstavlja interval brojača u sekundama, a zatim slijedi *boolean* argument koji ako je jednak `false`, *callback* metoda će se izvršiti samo jednom, a ako je jednak `true` izvršavat će se u intervalima definiranim prethodnim argumentom. Posljednjim argumentom možemo zadati koliko želimo da traje početna odgoda brojača, odnosno odgoda do prvog poziva *callback* metode. U klasi `ASurvivalWizardryGameModeBase` brojače koristimo za odbrojavanje do početka partije i za odbrojavanje trajanja partije.

```
void ASurvivalWizardryGameModeBase::StartGameCountdown()
{
    GetWorldTimerManager().SetTimer(CountdownTimerHandle, this,
    &ASurvivalWizardryGameModeBase::StartGame, GameCountdownLength, false);
    StartGameCountdownEvent();
}
```

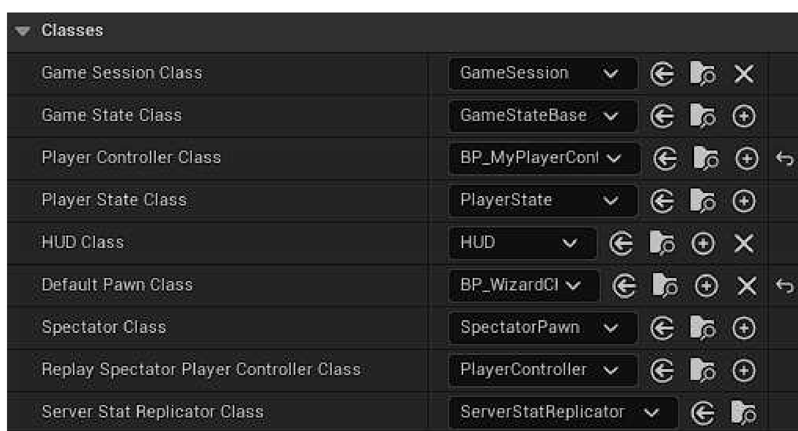
Isječak koda 4.3: StartGameCountdown metoda i postavljanje brojača

Nadalje, `ASurvivalWizardryGameModeBase` zadužen je za čuvanje informacija o trenutnom *experience*-u i nivou igrača. Ažuriranje trenutnog *experience*-a i nivoa igrača obavlja se pomoću metode `AddExperience()` koja se poziva svaki put kada skupimo *experience*, što se provjerava u klasi `AExpPickuble`.

`ASurvivalWizardryGameModeBase` također ažurira stanje igre kada ubijemo neprijatelja ili kada umre čarobnjak unutar metode `ActorDied()`. U slučaju kada umre čarobnjak, partija završava pa se poziva metoda `EndGame()` s argumentom `false`, a ako neprijatelj

umre na njegovom mjestu se stvara dragulj koji donosi *experience*. Metoda `EndGame()` prima jedan *boolean* argument – vrijednost `true` označava da je igra završila pobjedom, a vrijednost `false` označava da je završila porazom.

Game Mode klasa definira koje će se druge važne klase koristiti u igri, npr. može se postaviti klasa za igračev kontroler, što je u ovom slučaju klasa `BP_MyPlayerController`, te klasa za igračevog lika, u ovom slučaju `BP_WizardClass`. Koje klase će se koristiti možemo postavljati unutar *Blueprint*-a *Game Mode* klase u kartici detalja (slika 4.3). Kako bi *Unreal Engine* znao koju *Game Mode* klasu treba koristiti potrebno ju je postaviti u opcijama projekta unutar urednika ili u opcijama svijeta ako želimo da različite mape koriste različite *Game Mode* klase.



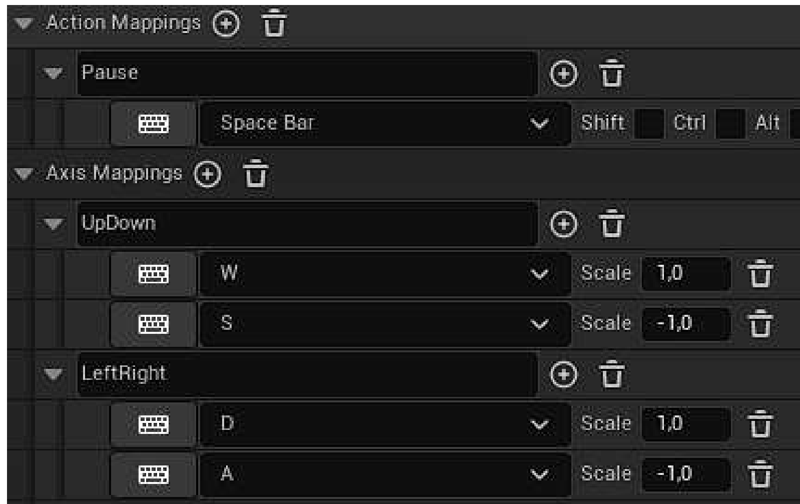
Slika 4.3: Game Mode klase

4.3 *Player Controller*

Kao što je rečeno, klasa koja predstavlja igračev kontroler je `BP_MyPlayerController`. To je *Blueprint* klasa bazirana na *C++* klasi `AMyPlayerController` koja nasljeđuje *Unreal Engine*-ovu klasu `APlayerController`. Glavni zadatak ove klase je obraditi korisnikov unos. Kako bi reagirali na korisnikove unose potrebno je vezati unos za *callback* funkciju što možemo napraviti unutar metode `SetupInputComponent()`. Prije toga moramo u opcijama projekta u uredniku mapirati unose na posebna imena koja kasnije možemo koristiti prilikom povezivanja s *callback* funkcijom čime dobivamo dodatan sloj apstrakcije pomoću kojeg možemo jednostavno mijenjati kontrole. Postoje dvije vrste mapiranja unosa: mapiranje akcija (eng. *action mappings*) i mapiranje osi (eng. *axis mappings*).

Mapiranje akcija odnosi se na mapiranje diskretnog pritiska tipki na tipkovnici ili mišu. Ovakvi unosi direktno aktiviraju određeno ponašanje.

Mapiranje osi pak mapira kontinuirane unose i funkcija za koju vežemo unos mora primati jedan argument tipa `float` koji predstavlja vrijednost osi. *Unreal Engine* svaki kadar provjerava da li je pritisnuta neka tipka te će izračunati vrijednost osi ovisno o pritisnutim tipkama i svaki kadar će se pozivati pridružena *callback* metoda s izračunatom vrijednosti osi.



Slika 4.4: Mapiranje unosa

Na slici 4.4 vidimo da su primjerice tipke **W** i **S** mapirane na *UpDown* ime s tim da je tipki **W** pridružen faktor 1.0, a tipki **S** faktor -1.0. Ako ne pritišćemo ni **W** ni **S** vrijednost osi je 0.0, ako pritisnemo **W** šalje se 1.0, a za **S** se šalje -1.0. Ako su obje tipke pritisnute istovremeno njihove vrijednosti se zbrajaju pa se šalje vrijednost 0.0.

Klasa kontroler sadrži komponentu `UInputComponent` koja omogućuje povezivanje unosa i *C++* metoda pomoću metoda `BindAxis()` i `BindAction()`. Sva povezivanja obavljamo unutar metode `SetupInputComponent()` kao što je prikazano u isječku koda 4.4.

```
1  void AMyPlayerController::SetupInputComponent()
2  {
3      Super::SetupInputComponent();
4      InputComponent->BindAxis(TEXT("UpDown"), this,
5          &AMyPlayerController::MoveUpDown);
6      InputComponent->BindAxis(TEXT("LeftRight"), this,
7          &AMyPlayerController::MoveLeftRight);
8  }
9
10 void AMyPlayerController::MoveUpDown(float Value)
```

```
11 {
12         FRotator Rotation = GetPawn()->GetActorRotation();
13         Rotation.Pitch = 0.f;
14         Rotation.Yaw = 0.f;
15         FVector WorldDirection = UKismetMathLibrary::
16                                     GetForwardVector(Rotation);
17         GetPawn()->AddMovementInput(WorldDirection, Value);
18 }
```

Isječak koda 4.4: Povezivanje unosa i implementacija kretanja lika

Ime *UpDown* koje zapravo predstavlja tipke **W** i **S** je povezano s metodom `MoveUpDown()` u kojoj je definirano kretanje lika u smjeru gore-dole. Analogno tipke **A** i **D** uzrokuju kretanje lika u smjeru lijevo-desno. Također, imamo nekoliko mapiranja akcija, primjerice tipka **SpaceBar** mapirana je na ime *Pause*.

4.4 *Wizard*

`BP_WizardClass` predstavlja čarobnjaka, odnosno klasu igračevog lika i bazirana je na *C++* klasi `AWizard` koja nasljeđuje *Unreal Engine*-ovu klasu `ACharacter`. U konstruktoru kreiramo razne potrebne komponente pomoću metode `CreateDefaultSubobject` kojoj kao parametar šaljemo tip komponente koju želimo kreirati, a kao argument ime komponente koje će biti prikazano u *Blueprint*-u. Primjer stvaranja komponente iz *C++* koda prikazan je u isječku koda 4.5.

```
1 Camera = CreateDefaultSubobject<UCameraComponent>(TEXT("Camera"));
```

Isječak koda 4.5: Stvaranje komponente

Klasa `AWizard` sadrži svojstvo `Staff` koje je deklarirano na sljedeći način

```
1 UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Staff",
2             meta = (AllowPrivateAccess = "true"))
3 TSubclassOf<class AStaffBase> Staff;
```

Isječak koda 4.6: Deklaracija tipa klase

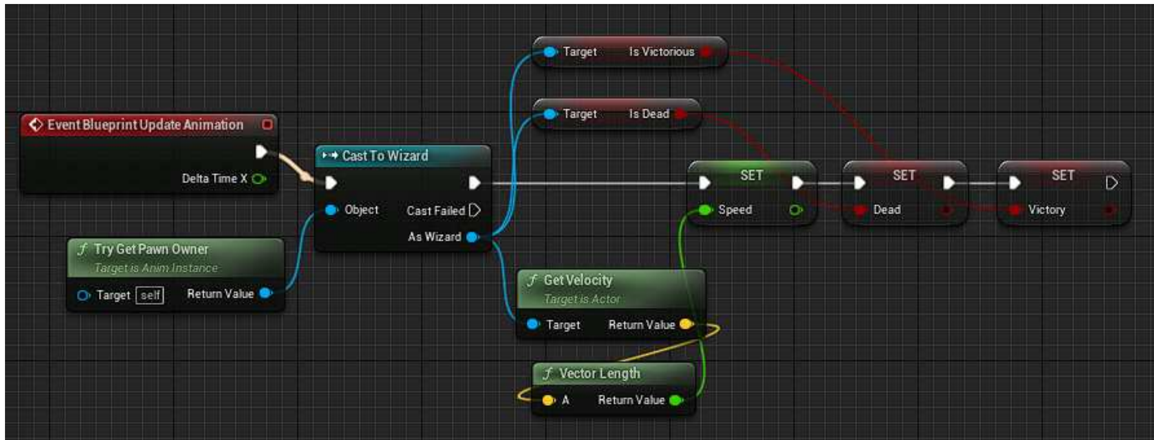
`TSubclassOf<>` nam omogućava pohranjivanje varijable koja predstavlja klasu koju smo predali kao tip unutar šiljastih zagrada ili bilo koju klasu izvedenu iz nje, tj. bilo koju njenu potklasu ili *Blueprint* klasu baziranu na njoj. Ako tome pridodamo `EditAnywhere`

specifikator, dobivamo mogućnost postavljanja konkretne klase iz urednika ili iz neke druge klase. Dakle, možemo imati baznu klasu nekog objekta te u *Blueprint*-u napraviti nekoliko klasa baziranih na toj klasi i na ovaj način možemo iz urednika određivati koja će se točno klasa koristiti, što može biti dobra praksa za razdvajanje dizajnerskog od programerskog dijela razvoja video igre. Alternativa ovome je korištenje običnog `UClass` pokazivača, ali nam ovaj pristup ne osigurava da je dodijeljena klasa uistinu izvedena iz željene klase. Što se tiče varijable `Staff`, nju postavljamo unutar urednika jer tako možemo u *Blueprint*-u dodati vizualnu reprezentaciju, tj. poligonalnu mrežu, te varijablu `Staff` postaviti na odgovarajuću *Blueprint* klasu.

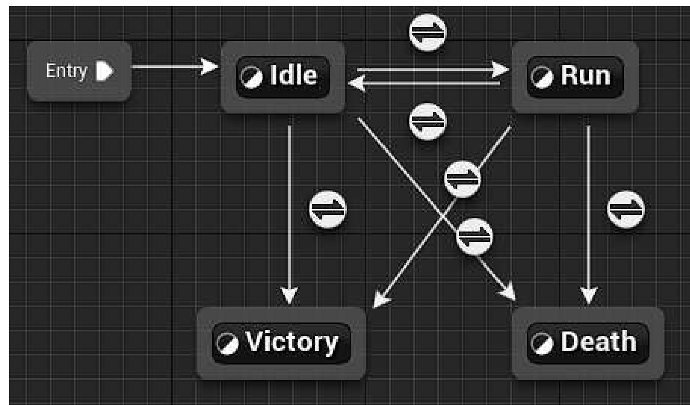
U metodi `BeginPlay()` stvara se instanca one klase koja je zadana preko varijable `Staff`, sprema se u varijablu `StaffInstance` i veže se na za nju predviđeno mjesto na *Skeletal Mesh* komponenti. Klasa `AWizard` čuva razna svojstva o igračevom liku poput njegove brzine kretanja i maksimalnog zdravlja (eng. *health*) kojima možemo postavljati vrijednosti unutar urednika te sadrži metode za dohvaćanje pojedinih informacija. Čarobnjak, tj. instanca klase `AWizard`, prilikom primanja štete kratak period ne može primati štetu kako bi se izbjeglo primanje štete prilikom svakog otkucaja u kojem je u kontaktu s protivnikom. To nazivamo stanjem nepobjedivosti i ono se postavlja i miče pomoću metoda `BecomeInvincible()` i `StopBeingInvincible()`. Kada igra započne, klasa `AWizard` pozove nad objektom `StaffInstance` metodu `StartCasting()`, a metodu `HandleDestruction()` pozove nakon što igra završi.

Lik čarobnjaka preuzet je iz *Epic Games*-ove trgovine sadržaja te uz skeletnu mrežu (eng. *skeletal mesh*), materijale i teksture sadrži i razne animacije koje se mogu upotrijebiti u video igri. Te animacije potrebno je ukomponirati u logiku igre odnosno postaviti kada će se koja animacija pokrenuti. To možemo napraviti koristeći *Animation Blueprint*. U *Animation Blueprint*-u postoje tri vrste grafa: graf događaja, graf animacija i graf stanja (eng. *state machine*). U grafu događaja kreiramo logiku i ažuriramo varijable koje ćemo koristiti u ostalim grafovima. Varijable koje koristimo u *Animation Blueprint*-u za igračevog lika, `WizardAnimation_BP`, su: `speed`, `dead` i `victory`. U grafu događaja dohvaćamo odgovarajuće vrijednosti varijabli iz klase igračevog lika i spremamo ih u varijable u *Animation Blueprint*-u (slika 4.5).

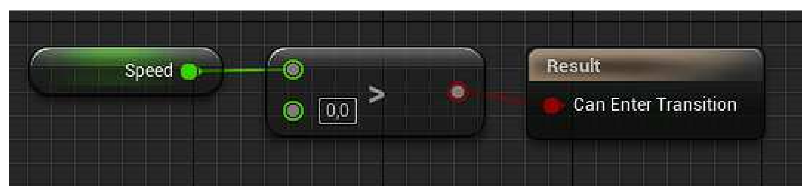
U animacijskom grafu dodajemo čvor koji predstavlja novi graf stanja i spajamo ga s čvorom koji predstavlja konačnu pozu za trenutni kadar. Graf stanja prikazan je na slici 4.6. Imamo četiri stanja za igračevog lika: *Idle* (stanje mirovanja), *Run* (stanje trčanja), *Victory* (stanje pobjede) i *Death* (stanje smrti, odnosno poraza). Svakom stanju pridružena je neka animacija. Uz stanja potrebno je definirati i prelaske iz jednog stanja u drugo. *Idle* stanje je prvotno i osnovno stanje, a ono prelazi u stanje *Run* kada varijabla `speed` postane strogo veća od nula. Iz stanja Run se prelazi u stanje *Idle* kada je varijabla `speed` jednaka nula. U stanje *Victory* se prelazi kada varijabla `victory` postane istinita i analogno za stanje *Death*.



Slika 4.5: Graf događaja u *Animation Blueprint*-u



Slika 4.6: Graf stanja



Slika 4.7: Prijelaz iz stanja Idle u stanje Run

4.5 *Health*

I igračev lik i neprijatelji imaju zdravlje, odnosno *health*, stoga je logika koja se bavi time odvojena u zasebnu komponentu, `UHealthComponent`, koja nasljeđuje *Unreal Engine* klasu `UActorComponent` i koju onda jednostavno dodamo klasama koje trebaju funk-

cionalnost koju pruža ova komponenta. Ona sadrži svojstvo `MaxHealth` koje predstavlja maksimalnu vrijednost zdravlja te svojstvo `Health` koje predstavlja trenutnu vrijednost zdravlja. Metoda zadužena za primanje štete, odnosno smanjenje zdravlja igračevom liku ili neprijateljima, je metoda `DamageTaken()`. Ta metoda veže se za *multi-cast* delegat `OnTakeAnyDamage` sa sljedećom linijom koda

```
GetOwner()->OnTakeAnyDamage.AddDynamic(this,&UHealthComponent::DamageTaken);
```

Isječak koda 4.7: Povezivanje metode na *multi-cast* delegat *OnTakeAnyDamage*

Multi-cast delegati u *C++*-u ili dispečeri događaja u *Blueprint*-u predstavljaju realizaciju *Observer* oblikovnog obrasca. Funkcije iz raznih klasa možemo vezati za *multi-cast* delegat ili dispečer te će se one izvršiti nakon što se izvrši delegat odnosno nakon poziva metode `Broadcast()` nad njim u *C++* slučaju ili izvršavanjem *Call Dispacher* čvora u *Blueprint*-u.

`OnTakeAnyDamage` delegat poziva metodu `Broadcast()` nakon što pozovemo metodu `UGameplayStatics::ApplyDamage()`. `ApplyDamage()` metodu pozivamo iz klase `AEnemyBase`. `DamageTaken()` iz `UHealthComponent` klase prikazana je u sljedećem isječku koda. Bitno je da signatura metode bude točno kako je prikazano u isječku koda kako bi se metoda mogla povezati s delegatom.

```
void UHealthComponent::DamageTaken(
    AActor* DamagedActor,
    float Damage,
    const UDamageType* DamageType,
    class AController* Instigator,
    AActor* DamageCauser)
{
    if(Damage <= 0.f) return;
    Health -= Damage;
    if(Health <= 0.f && SurvivalWizardryGameModeBase)
    {
        SurvivalWizardryGameModeBase->ActorDied(DamagedActor);
    }
}
```

Isječak koda 4.8: Metoda za primanje štete

`UHealthComponent` također sadrži i metodu `Heal()` kojom se povećava vrijednost varijable `Health` za određeni iznos.

4.6 *Staff*

Čarobnjački štap predstavljen je klasom `AStaffBase` i generalno služi kao spremnik za čarolije, odnosno instance klase `ASpellBase`. Sadrži statičku mrežu koju dodjeljujemo u *Blueprint* proširenju klase. Klasa `AStaffBase` sadrži varijablu `numSlots` koja označava koliko čarolija čarobnjački štap može sadržavati te sadrži polje pokazivača na instance klase `ASpellBase`. Prije je spomenuto kako klasa `AWizard` kada započne partija poziva metodu `StartCasting()` na instanci klase `AStaffBase`, a klasa `AStaffBase` tada pak poziva metodu `StartCasting()` iz klase `ASpellBase` nad instancama čarolija koje su pohranjene u polju `spells`. Također, metoda `HandleDestruction()` prolazi poljem `spells` i poziva nad njima metodu `HandleDestruction()`. Kada igrač skupi dovoljno *experience*-a i podigne nivo može odabrati jednu od ponuđenih opcija kako bi dobio novu čaroliju ili poboljšao neku postojeću što se realizira pomoću metode `AddSpell()`.

4.7 *Spells*

Informacije o čarolijama pohranjene su u tablici *DataTableOfSpells* baziranoj na strukturi podataka `FSpellStructure` čija su svojstva: `SpellName` (ime čarolije), `DisplayTexture` (slika koja predstavlja čaroliju), `ProjectileClass` (klasa projektila koju koristi čarolija), `Damage` (koliku štetu nanosi čarolija), `Cooldown` (koliko vremena treba proći između dva bacanja čarolije) i `Size` (veličina projektila kojeg čarolija stvara) .

Klasa `ASpellBase` uz dio svojstava koji su jednaki kao u strukturi `FSpellStructure` sadrži i nivo čarolije. Čaroliji možemo povećati nivo čime ona nanosi veću štetu i ima kraću frekvenciju bacanja (eng. *cooldown*). Metoda `CastProjectile()` služi za bacanje čarolija, odnosno ta metoda stvara instancu klase spremljene u varijabli `ProjectileClass`. Metoda `CastProjectile()` vezana je na brojač koji ju poziva svaki put kada istekne njoj pridruženo vrijeme između dva bacanja. Postoji nekoliko potklasa klase `ASpellBase` koje predstavljaju različite tipove čarolija i imaju ponešto drugačije funkcionalnosti. Tako primjerice klasa `ARadialProjectileSpell` ne stvara samo jedan projektil ispred čarobnjaka u smjeru njegova kretanja kako je implementirano u baznoj klasi, već stvara šest projektila s različitim rotacijama koji se onda kreću radijalno od čarobnjakove pozicije u trenutku stvaranja.

4.8 *Projectiles*

Klasa `AProjectileBase` sadrži `UProjectileMovementComponent` što je *Unreal Engine*-ova komponenta zadužena za pomicanje projektila. `AProjectileBase` klasa sadrži me-

todu `Initialize()` pomoću koje postavljamo sva svojstva koja su potrebna instanci ove klase poput brzine projektila, koliku štetu nanosi neprijateljima i slično. Ovu metodu pozivamo nakon što stvorimo instancu projektila i dalje se ne moramo brinuti o njegovom kretanju jer *Unreal Engine* to obavlja za nas. U `BeginPlay()` metodi vežemo metodu `OnOverlap()` na *multi-cast* delegat `OnComponentBeginOverlap` koji se aktivira kada se neki objekt nađe u koliziji s projektilom. Kolizija uzrokuje nanošenje štete objektu koji se našao u koliziji s projektilom. U uredniku možemo postaviti koji parovi tipova objekata mogu biti u koliziju, tj. možemo definirati kada će se kolizija aktivirati, a kada ignorirati.

```
1   void AProjectileBase::OnOverlap(
2           UPrimitiveComponent* OverlappedComponent,
3           AActor* OtherActor,
4           UPrimitiveComponent* OtherComponent,
5           int32 OtherBodyIndex,
6           bool bFromSweep,
7           const FHitResult& SweepResult
8   )
9   {
10      auto DamageTypeClass = UDamageType::StaticClass();
11      if(OtherActor && OtherActor != this)
12      {
13          if(Cast<AEnemyBase>(OtherActor))
14          {
15              UGameplayStatics::ApplyDamage(OtherActor, Damage,
16                              nullptr, this, DamageTypeClass);
17              --NumberOfPierces;
18              if(NumberOfPierces < 0)
19              {
20                  Destroy();
21              }
22          }
23          else
24          {
25              Destroy();
26          }
27      }
28  }
```

Isječak koda 4.9: Metoda za nanošenje štete

4.9 *Enemies*

Osnovni dio implementacije neprijatelja nalazi se u klasi `AEnemyBase`, a potom se ta klasa proširuje unutar *Blueprint*-a kako bi dobili različite tipove neprijatelja s različitim vizualnim reprezentacijama i s različitim vrijednostima svojstava. Kao i klasa `AWizard`, klasa `AEnemyBase` sadrži instancu klase `UHealthComponent`. Metoda `OnOverlap()` koja je vezana na *multi-cast* delegat `OnComponentBeginOverlap`, provjerava da li je neprijatelj u koliziji s čarobnjakom, odnosno igračevim likom, te ako je, provjerava da li je čarobnjak u stanju nepobjedivosti. Ako čarobnjak nije nepobjediv nanosi mu se šteta u vrijednosti varijable `Damage` pozivom metode `UGameplayStatics::ApplyDamage()`. Neprijatelji trebaju pratiti čarobnjaka stoga implementiramo njihovo kretanje u metodi `Move()` koju onda pozivamo u metodi `Tick()`, dakle svaki kadar ažuriramo poziciju svakog neprijatelja. Metoda `Tick()` prosljeđuje metodi `Move()` varijablu `DeltaTime` kako bi mogli implementirati kretanje koje je neovisno o broju kadrova u sekundi. Implementacija metode `Move()` prikazana je u sljedećem isječku koda.

```
1  void AEnemyBase::Move(float DeltaTime)
2  {
3          FVector enemyLocation = GetActorLocation();
4          if(wizard)
5          {
6                  FVector wizardLocation = wizard->GetActorLocation();
7                  FRotator current = GetActorRotation();
8                  FVector direction = wizardLocation - enemyLocation;
9                  direction.Normalize();
10                 FRotator target = direction.Rotation();
11                 FRotator interpolation = FMath::RInterpTo(current, target,
12                                                    DeltaTime, TurnRate);
13                 interpolation.Pitch=0.f;
14                 interpolation.Roll=0.f;
15                 SetActorRotation(interpolation);
16                 enemyLocation.X+=interpolation.Vector().X*DeltaTime*Speed;
17                 enemyLocation.Y+=interpolation.Vector().Y*DeltaTime*Speed;
18                 SetActorLocation(enemyLocation);
19         }
20 }
```

Isječak koda 4.10: Metoda za kretanje neprijatelja

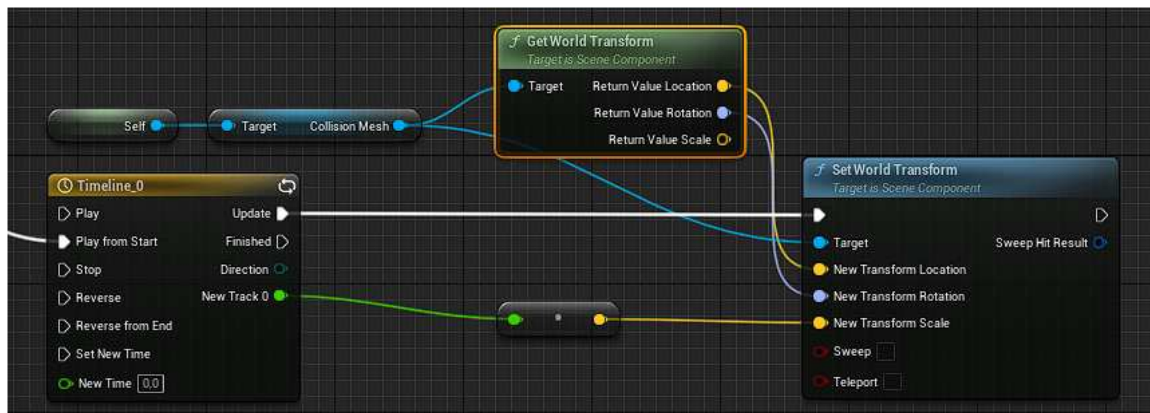
Kada neprijatelj umre pozove se metoda `HandleDestruction()` unutar koje se poziva metoda `Destroy()` koja uništava instancu aktera i oslobađa njegovu memoriju. Također, pokreću se prikladni zvučni i vizualni efekti na način prikazan u isječku koda 4.11. Varijable `DeathSound` (tipa pokazivač na klasu `USoundBase`) i `DeathParticles` (tipa poka-

zivač na `UParticleSystem`) postavljamo na konkretan zvuk i sustav čestica u *Blueprint* verzijama klase što omogućava jednostavno mijenjanje i isprobavanje različitih zvukova i vizualnih efekata.

```
1  void AEnemyBase::HandleDestruction()
2  {
3      Destroy();
4      if(DeathSound)
5      {
6          UGameplayStatics::PlaySoundAtLocation(this, DeathSound,
7                                                GetActorLocation());
8      }
9      if (DeathParticles)
10     {
11         UGameplayStatics::SpawnEmitterAtLocation(this,
12                                                  DeathParticles,
13                                                  GetActorLocation(),
14                                                  GetActorRotation());
15     }
16 }
```

Isječak koda 4.11: Metoda za uništenje neprijatelja

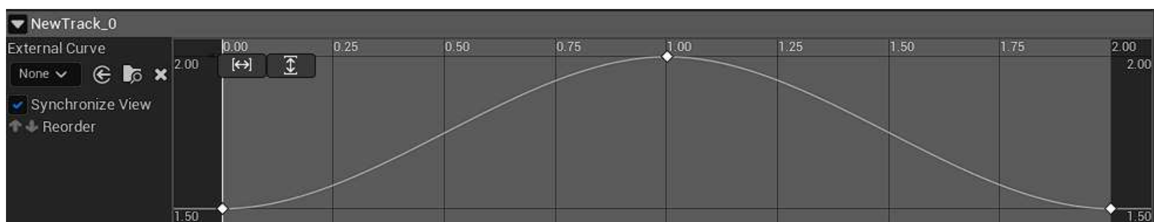
Pojedine vrste neprijatelja, odnosno pojedine *Blueprint* klase bazirane na *C++* klasi `AEnemyBase` , imaju animaciju tijekom kretanja implementiranu unutar *Unreal Engine* urednika pomoću takozvanog *timeline*-a. Za konkretan primjer uzet ćemo *Blueprint* klasu `BP_Cubie` . Dio grafa događaja ove klase prikazan je na slici 4.8.



Slika 4.8: *Timeline* čvor

Timeline čvor omogućava implementiranje jednostavnih animacija u ovisnosti o vremenu. Duplim klikom na *Timeline* čvor otvara nam se poseban *Timeline* urednik u kojem

možemo dodati novu traku i definirati izlazne vrijednosti *Timeline* čvora u određenim trenucima u vremenu. Točke na grafu predstavljaju uređeni par (vrijeme, vrijednost). Postoji nekoliko vrsta traka. Ona koju u ovom primjeru koristimo je *float* traka koja služi za animiranje skalarnih vrijednosti. Uz nju imamo još vektorsku traku kojom mijenjamo vrijednost vektora pa pomoću nje možemo animirati primjerice rotaciju objekta. Postoji i traka događaja koja omogućava postavljanje diskretnih trenutaka u periodu vremena kada će se aktivirati događaj (ovaj tip koristimo kod stvaranja neprijatelja). Posljednji tip trake je traka boja koju možemo koristiti za animiranje boja. Primjer trake koju koristimo u klasi `BP_Cubie` prikazan je na slici 4.9. Ovaj konkretan *timeline* mijenja veličinu instance klase `BP_Cubie` kroz vrijeme prateći krivulju definiranu u traci u *timeline*-u.



Slika 4.9: *Timeline* traka

4.10 *Experience Gem*

Klasa `AExpPickuble` predstavlja *experience* koje se stvori na mjestu neprijatelja nakon što on umre. Kada igračev lik dođe u koliziju s instancom ove klase pozove se njena metoda `OnOverlap()` koja je vezana na *multi-cast* delegat `OnComponentBeginOverlap`. Ova metoda pozove metodu `AddExperience()` nad `ASurvivalWizardryGameModeBase` klasom koja ažurira igračev ukupan *experience*. Nakon toga se instanca klase `AExpPickuble` uništi. Također, u trenutku kolizije s igračem ova klasa pokrene zvučni efekt sakupljanja.

4.11 *Health Pack*

Klasa `AHealthPack` predstavlja objekte koji se stvaraju na nasumičnim lokacijama i kada ih igrač skupi napuni mu se zdravlje za određeni iznos. Ova klasa, jednako kao i klasa `AExpPickuble`, sadrži metodu `OnOverlap()` vezanu na *multi-cast* delegat. Unutar te metode poziva se metoda `Heal()` nad igračevim likom, odnosno instancom klase `AWizard`.

4.12 *Spawner*

C++ klasa `AASpawner` i na njoj bazirana *Blueprint* klasa `BP_Spawner` zadužene su za dinamičko stvaranje objekta. U klasi `AASpawner` implementirano je nekoliko metoda za stvaranje instance zadane klase. Metoda `ASpawnOnCircleAroundWizard()` prikazana je u isječku koda 4.12 i nju koristimo za stvaranje neprijatelja na nasumičnoj poziciji na kružnici radijusa zadanim kroz argument metode i u čijem se središtu nalazi čarobnjak. Za stvaranje *health pack*-ova želimo još veću nasumičnost stoga njih stvaramo bilo gdje unutar arene pomoću metode `ASpawnInArena()`.

```
void ASpawner::SpawnOnCircleAroundWizard(
    UClass* SpawningType,
    float SpawnHeight,
    float Radius)
{
        if(SpawningType == nullptr){
                return;
        }
        if(Wizard == nullptr){
                return;
        }
        if(World)
        {
                FVector PlayerLocation = Wizard->GetActorLocation();
                FRotator Rotation = FRotator::ZeroRotator;
                FVector Location = PlayerLocation;
                float angle = FMath::RandRange(0.f, 2*PI);
                Location.X += FMath::Sin(angle)*Radius;
                Location.Y += FMath::Cos(angle)*Radius;
                Location.Z = SpawnHeight;
                FActorSpawnParameters params;
                params.SpawnCollisionHandlingOverride=
                    ESpawnActorCollisionHandlingMethod::AlwaysSpawn;
                World->SpawnActor(SpawningType,&Location,&Rotation,params);
        }
}
```

Isječak koda 4.12: Metoda za stvaranje instanci zadane klase na kružnici oko čarobnjaka

U klasi `BP_Spawner` koristimo navedene metode u kombinaciji s *Timeline*-om koji sadrži traku događaja za svaku vrstu neprijatelja. Svaka točka na grafu pojedine trake predstavlja da će se u tom trenutku stvoriti instanca klase kojoj je pridružena konkretna traka.

4.13 Korisničko sučelje

Svaka video igra zahtjeva nekakav oblik korisničkog sučelja (eng. *user interface*). *Unreal Engine* koristi *Unreal Motion Graphics UI Designer* (*UMG*) pomoću kojeg možemo kreirati elemente korisničkog sučelja, menije, *HUD*-ove (*Heads-up Displays*) i slično. Osnovna stavka *UMG*-a su takozvani *Widgets* koji predstavljaju postojeće dijelove funkcionalnosti koje možemo koristiti prilikom kreiranja vlastitog sučelja. To su primjerice gumb, klizač, tekst, slika i još mnogo njih, a možemo ih koristiti unutar *Widget Blueprint*-a koji reprezentira neki dio korisničkog sučelja. *Widget Blueprint* sastoji se od dva dijela: dizajnerski dio u kojem slažemo vizualne elemente sučelja i dio s grafom događaja u kojem implementiramo funkcionalnosti koje podupiru odabrane vizualne elemente. Instancu *Widget Blueprint*-a kreiramo koristeći *Create Widget* čvor čiju izlaznu vrijednost koja predstavlja referencu na kreirani objekt možemo proslijediti čvoru *Add to Viewport* koji ga dodaje prozoru za prikaz.

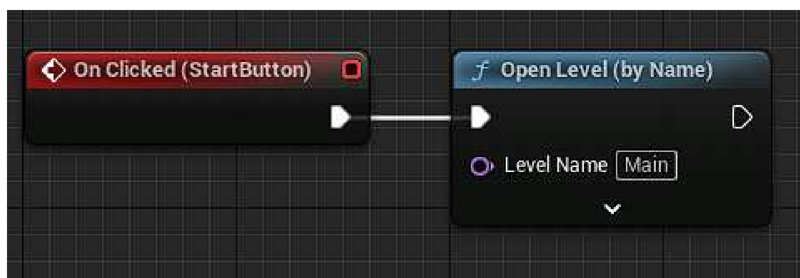
Glavni meni ove video igre implementiran je u zasebnom nivou, *Main_Menu_Level*, a *Widget Blueprint* koji predstavlja sučelje glavnog menija naziva se *Main_Menu_Widget*. Njega stvaramo i dodajemo prozoru za prikaz u *Blueprint*-u nivoa nakon što se aktivira *BeginPlay* događaj (slika 4.10).

Slika 4.10: Prikazivanje korisničkog sučelja

Main_Menu_Widget sastoji se od nekoliko najosnovnijih elemenata: platno, slika, gumb i tekst. Platno, odnosno *Canvas Panel* je spremnik za ostale *widget*-e i omogućava proizvoljno postavljanje elemenata unutar njega. Nadalje, imamo pozadinsku sliku predstavljenu *Image widget*-om te naslov igre predstavljen s *Text widget*-om. Na posljetku imamo tri gumba te je svakom od njih pridružen prikladni tekst. Gumbu možemo jednostavno pridružiti funkcionalnost dodavanjem *On Clicked* događaja te potom možemo u grafu događaja ovog *Widget Blueprint*-a koristiti taj događaj. Primjerice, klik na gumb *Start* emitira njemu pridružen *On Clicked* događaj nakon čega se učitava nivo u kojem je implementirana sama igra.

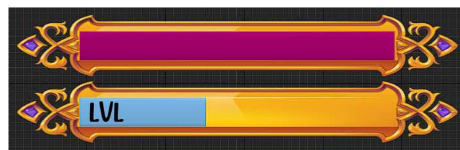
Informacije o trenutnom stanju zdravlja i *experience*-a dobivamo preko dvije trake (eng. *progress bar*) prikazane na slici 4.13. Implementacija se nalazi u *Widget Blueprint*-u pod

Slika 4.11: Glavni meni



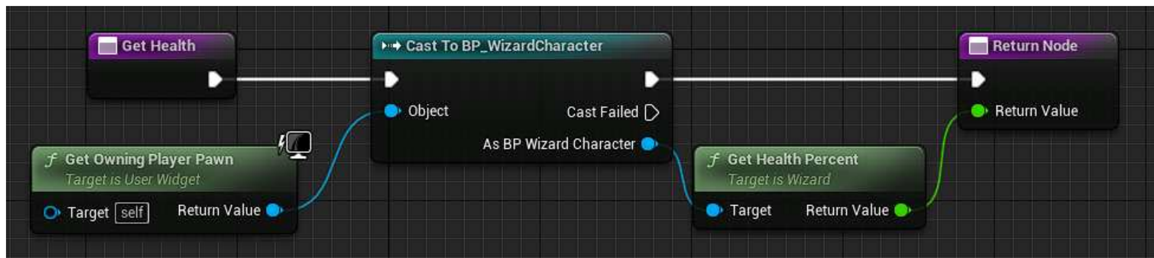
Slika 4.12: Funkcionalnost *Start* gumba

nazivom *ProgressWidget*. Traku možemo povezati s funkcijom implementiranom u *Blueprint*-u. Tako je primjerice traka zdravlja povezana s funkcijom *Get Health* prikazanoj na slici 4.14. Ova funkcija poziva metodu `GetHealthPercent()` iz *C++* klase `AWizard` i vraća njen rezultat kao izlaznu vrijednost na temelju koje se ispunjava traka zdravlja.



Slika 4.13

Uz dva prethodno opisana *Widget Blueprint*-a ovaj projekt ih sadrži još nekoliko, primjerice *widget* za odbrojavanje do početka partije, *widget* za odbrojavanje partije, *widget*



Slika 4.14

za zaslon na kraju partije i *widget* za kursor miša koji možemo postaviti u opcijama projekta u *Unreal Engine* uredniku. Nadalje, imamo *widget* za iscrtavanje čarolija koje su trenutno dostupne igraču. To radimo tako da pretražujemo tablicu čarolija u potrazi za pripadajućim slikama s obzirom na sadržaj polja `spells` na objektu koji predstavlja čarobnjački štap pridružen čarobnjaku. Važan nam je i *LevelUpWidget* koji se iscrtava kada igrač podigne nivo i ispisuju mu se opcije za dodavanje nove čarolije ili unaprjeđenje postojeće (slika 4.2). Opcije se odabiru nasumično iz tablice čarolija te ako odabrana čarolija već postoji u polju `spells`, za tu čaroliju se nudi podizanje njezinog nivoa, a ako ne postoji, igrač može odabrati novu čaroliju.

Zaključak

Nakon što je napravljen pregled arhitekture standardnog upravitelja video igara, vidi se koliko su to kompleksni sustavi i da je u većini slučajeva bolje koristiti neki postojeći upravitelj igara, nego implementirati vlastiti iz nule.

Unreal Engine pruža širok aspekt alata iznimno korisnih za razvoj video igara. Iako možda kompleksan za početnike zbog svoje opsežnosti, malo napredniji korisnici lako mogu vidjeti prednosti njegova korištenja. Korištenje *Blueprint*-a pokazalo se iznimno jednostavnim za obavljanje nekih manjih funkcionalnosti, dok se kompleksnije stvari lakše implementiraju u *C++*-u zbog manje preglednosti u velikim grafovima u *Blueprint*-u. Postavljanje vizualnih elemenata korisničkog sučelja unutar *Unreal Engine*-a pokazalo se dosta jednostavnim i nudi se velik izbor predefiniranih elemenata koji se lako povezuju s ostatkom igre. Paralelno korištenje *C++*-a i *Blueprint*-a početnicima može stvarati zbunjenost i uzrokovati pretjeranu raštrkanost dijelova implementacije. Nakon ponešto navikavanja na refleksijski sustav *Unreal Engine*-a, odnosno na vezu *C++*-a i *Blueprinta*-a postaje lakše balansirati korištenje jednog i drugog.

Nažalost, u službenoj dokumentaciji *Unreal Engine*-a, koja predstavlja glavni izvor materijala za učenje, manjka primjera korištenja metoda, pogotovo s *C++* strane, stoga je ponekad otežan pronalazak potrebnih metoda i načina njihova korištenja.

Bibliografija

[1] *Epic Games Developer Community*, https://dev.epicgames.com/community/, (siječanj 2023.).

[2] *Game development terms*, https://unity.com/how-to/beginner/game-development-terms#general-game-development-terms, (listopad 2022.).

[3] *How does the game loop actually work?*, https://forums.unrealengine.com/t/how-does-the-game-loop-actually-work/90508/2, (siječanj 2023.).

[4] *Is this the correct practice with UObjects and smart pointers?*, https://forums.unrealengine.com/t/is-this-the-correct-practice-with-uobjects-and-smart-pointers/279408/4, (siječanj 2023.).

[5] *Mixing blueprints and C++*, https://dev.epicgames.com/community/learning/tutorials/YDpo/mixing-blueprints-and-c, (siječanj 2023.).

[6] *Unreal Engine 5 Documentation*, https://docs.unrealengine.com/5.0/, (siječanj 2023.).

[7] *Unreal Engine Community Wiki*, https://unrealcommunity.wiki/, (siječanj 2023.).

[8] *What is CDO ? Part 2*, https://forums.unrealengine.com/t/what-is-cdo-part-2/349582/3, (siječanj 2023.).

[9] *What's the difference between using TWeakObjectPtr or using UObject**, https://forums.unrealengine.com/t/whats-the-difference-between-using-tweakobjectptr-or-using-uobject/284354, (siječanj 2023.).

[10] *Why doesn't UE utilize STL containers?*, https://forums.unrealengine.com/t/why-doesnt-ue-utilize-stl-containers/34551/38, (siječanj 2023.).

[11] *Wikipedia, Game Engine*, https://en.wikipedia.org/wiki/Game_engine, (studeni 2022.).

[12] *Wikipedia, Observer pattern*, https://en.wikipedia.org/wiki/Observer_pattern, (studeni 2022.).

[13] *Wikipedia, Prototype pattern*, https://en.wikipedia.org/wiki/Prototype_pattern, (studeni 2022.).

[14] *Wikipedia, Unreal Engine*, https://en.wikipedia.org/wiki/Unreal_Engine, (studeni 2022.).

[15] *Wikipedia, Video Game*, https://en.wikipedia.org/wiki/Video_game, (studeni 2022.).

[16] The Cherno, *Game engine series, YouTube*, https://youtu.be/vtWdgtMo1T4, (listopad 2022.).

[17] A. Cookson, R. DowlingSoka i C. Crumpler, *Unreal Engine 4 Game Development in 24 Hours*, Sams Publishing., 2016.

[18] Rachel Cordone, *Unreal Engine 4 Game Development Quick Start Guide*, Packt Publishing, 2019.

[19] Michael Dawson, *Beginning C++ Through Game Programming*, Cengage Learning PTR, 2015.

[20] John P. Doran i Matt Casanova, *Game Development Patterns and Best Practices*, Packt Publishing, 2017.

[21] James D. Foley, Andries van Dam, Steven K. Feiner, John Hughes, Morgan McGuire, David F. Sklar i Kurt Akeley, *Computer Graphics: Principles and Practice*, Addison-Wesley, 2013.

[22] Alex Forsythe, *Blueprints vs. C++: How They Fit Together and Why You Should Use Both, YouTube*, https://youtu.be/VMZftEVDuCE, (listopad 2022.).

[23] Hammad Fozi, Goncalo Marques, David Pereira i Devin Sherry, *Game Development Projects with Unreal Engine*, Packt Publishing, 2020.

[24] T. Fullerton, *Game Design Workshop: A Playcentric Approach to Creating Innovative Games*, Taylor & Francis Group, 2019.

[25] J. Gregory, *Game Engine Architecture*, Taylor & Francis Group, 2019.

[26] Leonardo Jelenković, *Operacijski sustavi, predavanja*.

[27] Mladen Jurak, *Objektno Programiranje (C++) predavanja*, https://web.math.pmf.unizg.hr/nastava/opepp/old/Slides/Predavanja/html-noslides/slides-0.html.

[28] R. Koster, *A Theory of Fun for Game Design*, O'Reilly, 2013.

[29] Robert Nystrom, *Game Programming Patterns*, 2014.

[30] S. Rogers, *Level Up! The Guide to Great Video Game Design*, John Wiley and Sons, 2014.

[31] Crystal Clear Game Studios, *Unreal Engine Magic System RPG Series 31 - Setting up SpellBook Data Table*, https://youtu.be/I-pur1zjv_s, (siječanj 2023.).

Sažetak

Ovaj rad započinje uvođenjem pojma video igara i upravitelja video igara te s njihovim upoznavanjem kroz povijest. Opisuje se arhitektura standardnog upravitelja igara, a kako se ona sastoji od mnogo različitih dijelova, svaki je dio opisan u kratkim crtama. Nakon toga navode se osnovni pojmovi objektno orijentiranog programiranja kako bi se lakše mogao pratiti ostatak rada. Nakon toga slijedi opis nekolicine oblikovnih obrazaca koji se često koriste u razvoju video igara. Najvažniji oblikovni obrasci za video igre su: *Game loop*, *Update Method* i *Components*. Potom slijedi upoznavanje s *Unreal Engine*-om, jednim od najpopularnijih komercijalnih upravitelja igara današnjice. Navodimo njegove osnovne elemente te se upoznajemo s vizualnim skriptnim jezikom pod nazivom *Blueprint* te s *Unreal Engine*-ovim aplikacijskim programskim sučeljem za rad s *C*++-om. Uvodimo pojam sustava za refleksiju *Unreal Engine*-a i objašnjavamo njegovu važnost. Na kraju rada opisujemo video igru pod nazivom *Survival Wizardry*, napravljenu u sklopu praktičnog dijela ovog rada, te kroz njezinu implementaciju upoznajemo još razne koncepte iz *Unreal Engine*-a.

Summary

This master thesis begins by introducing the concept of video games and game engines and introduces them through history. The architecture of the standard game engine is described, and since it consists of many different parts, each part is described in short lines. After that, the basic terms of object-oriented programming are stated so that the rest of this paper can be followed more easily. This is followed by a description of several design patterns that are often used in video game development. The most important design patterns for video games are: Game loop, Update Method and Components. Then, we are introduced to *Unreal Engine*, one of the most popular commercial game engines available at this time. We list its basic elements and get acquainted with the visual scripting language called *Blueprint* and with *Unreal Engine*'s application programming interface for working with *C++*. We introduce the concept of the *Unreal Engine* reflection system and explain its importance. At the end, we describe the video game named *Survival Wizardry*, made as a practical part of this master thesis, and through its implementation we get to know various other concepts from *Unreal Engine*.

Životopis

Rođena sam 1.1.1998. u Šibeniku, gdje sam pohađala Osnovnu školu Fausta Vrančića. Srednjoškolsko obrazovanje sam stekla na prirodoslovno-matematičkom smjeru Gimnazije Antuna Vrančića u Šibeniku. 2016. godine upisujem preddiplomski studij Matematika na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta u Zagrebu. Preddiplomski studij završavam 2019. godine kada upisujem diplomski sveučilišni studij Računarstvo i matematika na istom fakultetu. U travnju i svibnju 2020. godine obavljam studentsku praksu u firmi *Xylon D.o.o.*, a od travnja do srpnja 2022. godine obavljam praksu i studentski rad u firmi *Ekobit*.