Primjena Unity Engine-a za izradu 3D računalne igre

Čatalinac, Dino

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
ELEKTROTEHNIČKI FAKULTET

Sveučilišni studij

PRIMJENA UNITY ENGINE-A ZA IZRADU 3D RAČUNALNE IGRE

Diplomski rad

Dino Čatalinac

Osijek, 2016.

SADRŽAJ

1 UVOD

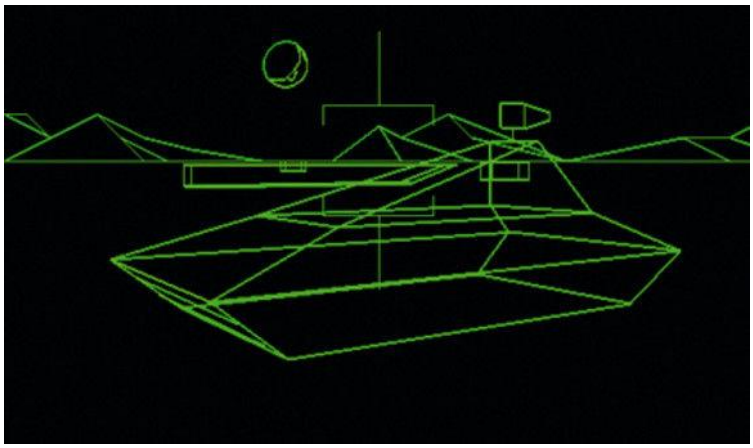
1.1 Povijest 3D računalnih igara

Prve računalne igre pojavljuju se već 1950-tih godina. Te su igre uglavnom bile razvijane u sveučilištima, velikim korporacijama ili vladinim organizacijama u svrhu istraživanja i provedbe raznih simulacija, te su bile nedostupne široj upotrebi. To su najčešće bile razne logičke igre (poput šaha), kartaške igre, vojne simulacije i sl. Mnoga računala u to vrijeme nisu imala mogućnost spremanja programa, pa su rane implementacije računalnih igara trajno izgubljene, stoga je nemoguće reći koja je prva video igra u povijesti računarstva. Prva računalna igra službeno prezentirana javnosti je Bertie the Brain (Slika 1.1.), koju je razvio Josef Kates 1950. u Torontu.

Slika 1.1 Zabavljač Danny Kaye slavi pobjedu nad Bertie the Brain-om na kanadskoj nacionalnoj izložbi

Računalne igre nisu stekle masovnu popularnost sve do 1970-tih pojavom arkada i igračih konzola, koje su postale dostupne široj javnosti. Prije pojave 3D video igara, računalnu grafiku su činili grafički znakovi (eng. *sprite*), odnosno dvodimenzionaln slike (eng. *bitmap*), te su objekti u virtualnim scenama bili ograničeni predefiniranim animacijama. Za razliku od sprite-ova, 3D modeli imaju vlastiti "endoskeleton", te mogućnost fizičke interakcije sa okolinom. Npr. ukoliko krpena lutka padne na stepenicu, neće se samo statično zaustaviti sa nekom unaprijed definiranom animacijom, nego će obaviti svoje noge ili ruke oko stepenica.

Prvi veliki 3D uspjeh doživio je *Battlezone* (slika 1.2.), igra razvijena 1980. koja je koristila vektorsku grafiku, slično kao *Asteroids* (koja je bila 2D). Iako vrlo jednostavna za današnje standarde, tada je to bila izuzetno složena igra, u kojoj je postojala mogućnost kretanja u bilo kojem smjeru unutar virtualnog svijeta.



Slika 1.2. Battlezone je bio smatran toliko realističnim da ga je koristila čak i američka vojska za uvježbavanje svojih vojnika

Prva prava Freescape igra bio je *Driller*, razvijen 1987. Za razliku od *Battlezone*-a, 3D svijet u *Driller*-u je imao, iako jednobojno, teksturirani svijet. Prvi veći hit 3D je doživio 1992. distribucijom *Wolfenstein 3D*-a, pucačinom koja je imala teksturiranu unutrašnjost, međutim vrlo limitiranu interakciju sa svijetom. Iste godine izašao je *Ultima Underworld*, igra koja je imala površine s nagibom, napredne efekte osvjetljenja, dijalog, logičke zagonetke, magiju, fiziku, 3D objekte, priču, mogućnost gledanja gore-dolje i još mnogo toga. No sve ove igre su po današnjim standardima još uvijek smatrane polu-3D ili nekad zvane 2.5D, jer su imale ravne plohe i nije npr. postojala mogućnost da jedna soba bude iznad ili ispod druge. Kasnije su neke igre uspjele riješiti te probleme, međutim modeli objekata i likova su još uvijek bili renderirani sprite-ovima, uglavnom zbog performansi.

Prva potpuna 3D igra bio je Quake, izdan 1996., međutim po cijeni lošije vizualnosti. Kao i kod svake grafičke tehnologije, tijekom narednih godina vizualnost se poboljšala, ali je postajalo sve jasnije da veća procesorska snaga nije dovoljna da održi prihvatljive perfomanse. Kako su se standardi vizualnosti povećavali, tako se povećavala i potreba za zasebnim grafičkim karticama (procesorima) koje se pojavljuju 1997. godine. Sljedeći veliki iskorak 3D grafika doživljava

pojavom tzv. *shader*-a, kodiranih skripti koje manipuliraju načinom renderiranja pojedinih piksela na zaslonu, no o tome će se detaljnije pričati kasnije kroz ovaj rad. Od 1980-tih do danas, računalne igre postale su jedan od najpopularnijih načina zabave i dio moderne kulture u većini dijelova svijeta.

1.2 Opis rada

U ovom radu će se ukratko opisati osnove 3D računalne grafike, te načine prikazivanja modela u 3D virtualnom svijetu. Nakon toga će biti upoznavanja sa Unity engine-om, platformom za razvoj 3D računalnih igara, kako se u njoj izrađuju 3D modeli, te kako se realizira igrivost (eng. *gameplay*), odnosno interakcija igrača sa virtualnim svijetom uz korištenje C# programskog jezika. Kao primjer, u ovom radu će se realizirati 3D igra poznata pod žanrom "Tower Defense", gdje igrač mora graditi kule i topove za obranu od invazivnih neprijatelja prije nego uspiju doći do određene točke. U ovoj igri poseban će fokus biti okrenut prema implementaciji umjetne inteligencije i već navedenom gameplay-u, a manje na modeliranju samog virtualnog svijeta, koji za ovakav tip igre nije ni bitan. Iako će biti govora o izradi modela u Unity-u, koristit će se već gotovi asseti (resursi) da bi se igra brže i jednostavnije realizirala, jer sama izrada 3D modela nije (jedina) tema ovoga rada, nego je cilj razviti zanimljivu, ali i vizualno interesantnu računalnu igru bez previše gubljenja vremena na manje bitne detalje.

2 OSNOVE 3D RAČUNALNE GRAFIKE

3D računalna grafika je slika grafički obrađena korištenjem digitalnih računala i specijaliziranih 3D software-a (alata). Ona se također može definirati i kao proces stvaranja takvih slika ili kao grana znanosti koja proučava 3D računalnu tehnologiju i njene srodne tehnologije.

3D računalna grafika se razlikuje od 2D računalne grafike u tome što se trodimenzionalni opis nekog geometrijskog podatka sprema u računalo u svrhu izvođenja izračuna i renderiranja dvodimenzionalnih slika. Općenito, vještina 3D modeliranja, koja priprema geometrijske podatke za 3D računalnu grafiku je slična skulpturiranju odnosno oblikovanju, dok je vještina 2D grafike analogna slikanju odnosno crtanju. Međutim, 3D računalna grafika se oslanja na mnoge iste algoritme generiranja kao i 2D računalna grafika.

Ponekad se 2D slika dobivena izračunima trodimenzionalnih geometrijskih podataka unaprijed renderira (eng. *pre-render*), međutim vrlo je često potrebno da se ona generira u stvarnom vremenu (eng. *real-time*). Najpoznatiji API-ji za renderiranje slika u stvarnom vremenu su OpenGL i Direct3D. Mnoge današnje grafičke kartice pružaju mogućnost hardware-ubrzanja (eng. *hardware acceleration*) baziranog na tim API-ima, pa često omogućuju prikaz kompleksne 3D računalne grafike u stvarnom vremenu.

2.1 Stvaranje 3D računalne grafike

Proces stvaranja 3D računalne grafike se može podijeliti u 3 osnovne faze:

- Modeliranje
- Priprema rasporeda scene
- Renderiranje

Modeliranje

Faza modeliranja se može opisati kao proces oblikovanja pojedinih objekata koji će se kasnije koristiti u sceni. Postoji nekoliko tehnika modeliranja, neke od njih su:

- Konstruktivna kruta geometrija (eng. *constructive solid geometry*)
- NURBS modeliranje
- Poligonsko modeliranje (eng. *polygon modeling*)
- Površinsko odsijecanje (eng. *subdivision surfaces*)
- Implicitne površine (eng. *implicit surfaces*)

Proces modeliranja podrazumijeva i uređivanje površina objekata, promjene gradivnih svojstava (boje, osvjetljenja, odsjaja, sjene, prozirnosti, indeksa refrakcije i dr.), dodavanja tekstura, mapiranja ispupčenja i drugih svojstava.

Priprema 3D modela za animacije je također dio modeliranja. U objekte se ugrađuju *kosturi* (eng. *skeleton*), tj. središnje okosnice koje imaju sposobnost promjene oblika objekta ili mu omogućuju kretanje. Kod složenijih modela proces definiranja animacija spada pod posebnu fazu koja se naziva montaža (eng. *rigging*), pri čemu su modelu dodijeljenje posebne kontrole koje mu omogućuju lakšu i intuitivniju animaciju, poput izraza lica ili složenih pokreta usta pri govoru.

Priprema rasporeda scene

Priprema scene podrazumijeva slaganje virtualnih objekata, svjetala, kamera i drugih entiteta unutar scene koja će se kasnije koristiti za stvaranje nepokretne slike ili animacije. Ukoliko se koristi za animaciju, ova faza često koristi tehniku uokvirivanja (eng. *keyframing*) koja olakšava stvaranje kompleksnih pokreta unutar scene. Pomoću keyframing-a, umjesto korištenja fiksnih pozicija, rotacija ili skala (eng. *scaling*) za svaki okvir (eng. *frame*) unutar neke animacije, potrebno je samo definirati nekoliko ključnih okvira između kojih su stanja u svakom okviru interpolirana.

Osvjetljenje (eng. *lighting*) je bitan aspekt pripreme scene. Kao i u stvarnom svijetu prilikom rasporeda npr. neke filmske scene, osvjetljenje je bitan faktor koji doprinosi estetici i vizualnoj kvaliteti krajnjeg produkta. Kao takvo, ono je umjetnost koju je teško savladati. Efekti osvjetljenja mogu značajno doprinijeti općenitom ugođaju i emocionalnom odgovoru kojeg je potakla scena, što je dobro poznato fotografima i tehničarima za osvjetljenje na pozornicama.

Teselacija i mreže

Proces transformiranja opisa objekta, poput središnje koordinate sfere i njenog radijusa u poligonski prikaz sfere se naziva teselacija (eng. *tessellation*). Ova faza se koristi u poligonskom renderiranju, pri čemu se objekti pojednostavljuju iz apstraktnih opisa (eng. *primitives*) poput sfera, stožaca i dr. u tzv. “mreže“ (eng. *mesh*) koje čine međusobno spojeni trokuti. Mreže trokuta (za razliku od npr. kvadrata) se često koriste jer se lakše renderiraju pomoću tzv. *scanline* renderiranja.

Poligonski opisi objekata se ne koriste u svim tehnikama renderiranja, te se u tim slučajevima faza teselacije preskače prilikom prijelaza iz apstraktnog opisa u renderiranu scenu.

Renderiranje

Renderiranje je proces stvaranja 2D slike ili animacije iz pripremljene scene. Ovo se može usporediti sa fotografiranjem ili snimanjem scene u stvarnom životu nakon što je ona prethodno pripremljena.

Renderiranje interaktivnih medija, poput video igara i simulacija, se izvodi u stvarnom vremenu pri otprilike 20 to 120 slika u sekundi (eng. *frames per second, fps*). Animacije za neinteraktivne medije, poput video zapisa i filmova, se renderiraju mnogo sporije. Renderiranje koje nije u stvarnom vremenu omogućava podjelu ograničene snage procesora unutar većeg vremenskog intervala da bi se dobila slika veće kvalitete. Vrijeme renderiranja pojedinog okvira (slike, eng. *frame*) može biti od jedne sekunde do jednog ili više sati, ovisno o složenosti scene. Renderirani okviri se spremaju na tvrdi disk, te se poslije prema potrebi prebacuju na neki prijenosni medij, poput optičkog diska. Ti se okviri nakon toga prikazuju sekvencijalno pri visokim brzinama, obično 24, 25 ili 30 fps, pri čemu se dobije iluzija pokreta.

Postoje dva načina na koji se to dobije: praćenje zraka svjetlosti (eng. *ray tracing*, slika 2.1.) i poligonsko renderiranje bazirano na grafičkoj kartici. Ciljevi su različiti:

Slika 2.1. Ray tracing može renderirati vrlo realistične slike

U *ray tracing*-u, cilj je foto-realističnost. Vremena renderiranja su reda nekoliko sekundi do ponekad i nekoliko dana za samo jednu sliku/okvir. Ovo je osnovna tehnika renderiranja korištena u filmovima, digitalnoj mediji, umjetničkim radovima i dr.

Kod renderiranja u stvarnom vremenu, cilj je prikaz što je više moguće informacija koje ljudsko oko može procesirati unutar jedne tridesetine sekunde. Ovdje je glavni prioritet brzina, a ne foto-realističnost. Ovdje je zapravo cilj dobiti rezultat na način kako ljudsko oko percipira svijet, tako da krajnja dobivena slika nije i slika stvarne prirode, nego ona slika s kojom se oko može poistovjetiti. Ovo je osnovna tehnika korištena kod video igara i interaktivnih simulacija.

Foto-realistična kvaliteta je često željeni rezultat, te su zbog toga razvijene mnoge specijalizirane metode renderiranja. One mogu biti od jednostavnijih metoda poput *wireframe* renderiranja pomoću poligona, do složenijih tehnika poput: scanline renderiranja, ray tracing-a ili izračenja (eng. *radiosity*).

Software-i za renderiranje mogu simulirati takve vizualne efekte kao svjetlosne leće (eng. *lens flares*), dubine vidokruga (eng. *depth of field*) ili zamagljenje uzrokovano kretanjem (eng. *motion blur*). To su pokušaji simuliranja vizualnih fenomena koje su rezultat optičkih karakteristika kamera i ljudskog oka. Ovi efekti daju sceni dodatni element foto-realističnosti.

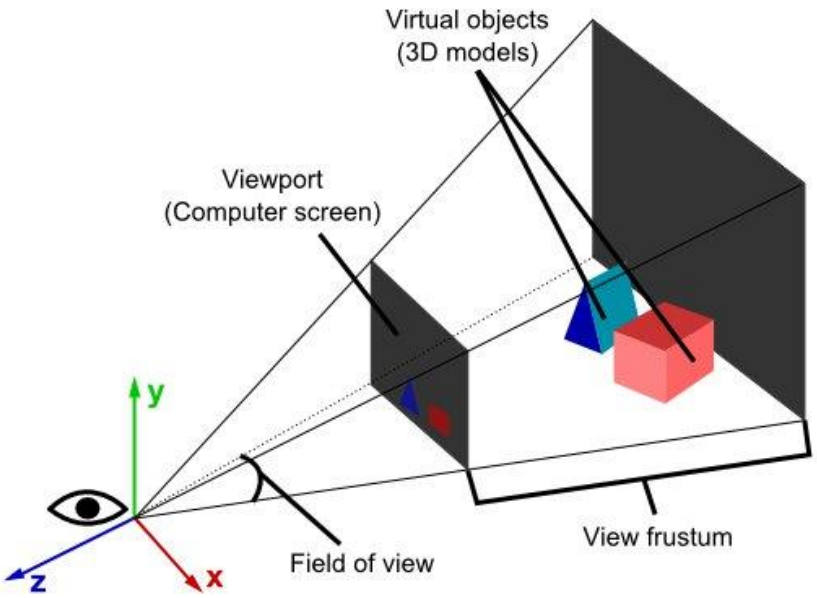
Razvijene su tehnike za simulaciju i drugih prirodno nastajućih fenomena, poput interakcije svjetlosti sa različitim oblicima materije. Primjer takvih tehnika su sustavi čestica (eng. *particle systems*), koji mogu simulirati kišu, dim ili vatru, volumetrijsko uzorkovanje (eng. *volumetric sampling*) za simulaciju magle, prašine i drugih atmosferskih efekata, kaustiku (za simulaciju refrakcije svjetla od različitih površina) i površinsko raspršivanje (eng. *subsurface scattering*), za efekt reflektiranja svjetlosti unutar zatvorenih objekata.

Proces renderiranja je skup u smislu potrebnog vremena izvršenja kalkulacija, s obzirom na kompleksnu raznolikost fizikalnih procesa koje je potrebno simulirati. Procesorka snaga računala je posljednjih godina rapidno porasla, omogućujući postepeno viši stupanj realističnog renderiranja. Filmski studiji koji produciraju računalno-generiranu animaciju obično koriste čitave "farme" za renderiranje (eng. *render farm*) za stvaranje slika unutar razumnog vremena. Međutim, kako cijena hardware-a konstantno opada, postaje sve više moguće kreirati manju količinu 3D animacija na osobnim računalima.

Često je software za renderiranje (eng. *renderers*) uključen s 3D software-om, no ponekad dolaze i kao zasebni plugin-ovi za razne 3D aplikacije. Krajnji rezultat dobiven renderom je često

samo jedan djelić čitave animirane (eng. *motion-picture*) scene, odnosno mnogo se slojeva materijala zasebno renderiraju, te se integriraju u konačan snimak koristeći software za kompoziranje.

Projekcija

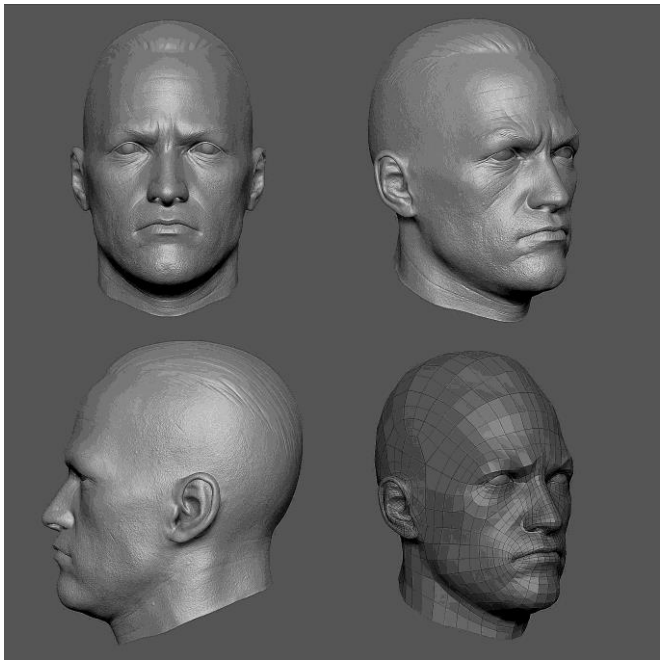


Slika 2.2. Perspektivna projekcija

S obzirom da ljudsko oko vidi u tri dimenzije, matematički model opisan unutar računala se mora transformirati nazad u sliku s kojom se ljudsko oko može poistovjetiti sa stvarnom slikom u prirodi. No, kako uređaj za prikaz (prvenstveno monitor) može prikazivati samo dvodimenzionalno, ovaj matematički model je potrebno pretvoriti u dvodimenzionalnu sliku. Često se to dobiva pomoću projekcije, uglavnom korištenjem perspektivne projekcije (Slika 2.2.). Glavni smisao perspektivne projekcije (kako i inače funkcionira ljudsko oko) je u tome da su udaljeniji objekti manji u usporedbi s onima koji su bliže oku. Stoga da bi se treća dimenzija razbila na zaslon, koristi se odgovarajuća matematička operacija, u ovom slučaju operacija dijeljenja.

Ortogonalna projekcija se uglavnom koristi u CAD ili CAM aplikacijama, gdje znanstveni modeli zahtjevaju precizne mjere i očuvanje treće dimenzije.

2.2 3D model



Slika 2.3. 3D model iz različitih perspektiva

3D model je trodimenzionalni poligonski opis objekta, često prikazan preko računala ili nekog drugog video uređaja. Objekt može varirati od entiteta iz stvarnog svijeta do fikcije, od minijaturnog do ogromnog. Bilo što iz stvarnog svijeta se može opisati nekim 3D modelom.

3D modeli se najčešće kreiraju posebnim softwareima, no ne nužno. S obzirom da je 3D model samo skup podataka (točaka i drugih informacija), on se može kreirati ručno ili algoritamski. Iako oni najčešće postoje virtualno (spremljeni u memoriji), čak i samo opis modela na papiru se može smatrati 3D modelom.

3D modeli se koriste svugdje gdje se koristi 3D računalna grafika. Oni su se zapravo pojavili prije nego se 3D računalna grafika počela koristiti na osobnim računalima. Mnoge su računalne igre koristile unaprijed renderirane slike 3D modela kao *sprite*-ove prije nego su ih računala mogla renderirati u stvarnom vremenu.

Danas 3D modeli imaju široku i raznoliku primjenu. U medicini se primjerice koriste za detaljan opis ljudskih organa. Filmska industrija ih koristi za likove i objekte za animiranu sliku. Industrija video igara ih koristi kao resurse (eng. *assets*) za računalne igre. Znanstveni sektor ih

koristi za demonstraciju kemijskih spojeva. U arhitekturi se koriste za prikaz predloženih zgrada i pejzaža. Tehnička industrija ih koristi za dizajn novih uređaja, vozila, građevina i dr.

3D model sam po sebi nije vizualan. On se može renderirati kao jednostavni *wireframe* različitih razina detalja, ili se zasjenčati na različite načine. Mnogi 3D modeli su međutim obloženi "pokrivačima" koji se nazivaju teksture. Proces poravnavanja tekstura sa koordinatama 3D modela se naziva teksturalno mapiranje (eng. *texture mapping*). Tekstura nije ništa drugo nego obična slika, no ona daje modelu više detalja i čini ga foto-realističnijim. 3D model osobe izgleda mnogo realističnije sa teksturom kože i odjeće, nego kao jednostavni monokromatski model (slika 2.3.) ili kao wireframe modela.

Osim tekstura, u 3D modele se mogu dodati i drugi efekti koji doprinose njegovoj foto-realističnosti. Primjerice, mogu se izmijeniti površinske normale koje određuju način na koji se površina osvjetljava. Neke površine mogu imati izbočine i mnogo drugih trikova se može izvesti.

3D modeli su često animirani za određenu primjenu, npr. u filmovima i računalnim igrama. Mogu biti animirani unutar 3D modelera ili u zasebnim alatima. Često postoji dodatna informacija s kojom je lakše animirati. Primjerice, 3D modeli ljudi i životinja imaju čitave kosture u sebi, tako da izgledaju realistično kada se kreću.

2.3 3D projekcija

3D projekcija je matematička transformacija koja se koristi za projekciju trodimenzionalnih točaka na dvodimenzionalnoj površini. Ona se koristi s ciljem simulacije odnosa kamere i subjekta. 3D projekcija je često prvi korak u procesu prikaza trodimenzionalnih oblika dvodimenzionalno u računalnoj grafici, odnosno proces renderiranja.

Ovdje će se opisati algoritam koji je bio standard kod prvih računalnih simulacija i video igara, no koji se i danas koristi, iako s većim modifikacijama za pojedini slučaj primjene.

Podaci potrebni za projekciju

Podaci o objektu koji je potrebno renderirati su obično spremljeni u obliku skupine točaka, međusobno povezanih u trokute. Svaka točka je vektor kojeg čine 3 broja koji opisuju njene X,Y,Z koordinate od ishodišne točke objekta kojem pripada. Svaki trokut je također vektor kojeg čine 3 točke ili indeksi točaka. Dodatno, objekt ima i 3 koordinate X,Y,Z i neku vrstu rotacije, npr. 3 kuta

alpha, *beta* i *gamma*, koji opisuju njegovu poziciju i orjentaciju relativno na ishodišnu točku scene (eng. *world reference frame*).

Zadnje što dolazi je promatrač, često se koristi i pojam kamera. Kamera također ima niz X,Y,Z koordinata i *alpha*, *beta*, *gamma* kuteve, koji opisuju poziciju promatrača i smjer gledanja.

Svi ovi podaci se obično spremaju kao decimalne vrijednosti, čak i kad ih mnogi programi pretvaraju u cjelobrojne u svojim algoritmima s ciljem ubrzanja kalkulacija.

Prvi korak: transformacija scene

Prvi korak je transformacija točaka uzimajući u obzir poziciju i orjentaciju objekta kojem pripadaju. To se radi korištenjem seta od četiri matrice:

$$\begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

-translacija

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

-rotacija oko x-osi

$$\begin{bmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

-rotacija oko y-osi

$$\begin{bmatrix} \cos\gamma & -\sin\gamma & 0 & 0 \\ \sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

-rotacija oko z-osi

Ove četiri matrice se zajedno množe i kao rezultat se dobije matrica transformacije scene (eng. *world transform matrix*). Ako se koordinate neke točke pomnože sa ovom matricom, kao rezultat se dobije točka čije su koordinate izražene unutar referentnog sustava scene.

Potrebno je napomenuti da za razliku od množenja običnih brojeva, redoslijed množenja matrica je bitan: promjenom redoslijeda mijenja se i rezultat. Kada su u pitanju rotacijske matrice, dobro je imati fiksan redoslijed množenja. Objekt bi se trebao rotirati prije nego je translatiran, u suprotnom pozicija objekta bi se rotirala s obzirom na ishodište svijeta, gdje god se ono nalazilo.

$$Transformacija\ scene = translacija \cdot rotacija \tag{2.1.}$$

Da bi se upotpunila transformacija na najopćenitij način, koristi se dodatna matrica koja se naziva matrica skale (eng. *scaling matrix*) za skaliranje modela uzduž osi. Ova matrica se množi sa prethodne četiri da bi se dobila potpuna transformacija scene. Oblik ove matrice je sljedeći:

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

-pri čemu su s_x, s_y i s_z faktori skale oko tri koordinatne osi

S obzirom da je pogodno skalirati model unutar njegovog zasebnog prostora modela odnosno koordinatnog sustava, skaliranje bi trebala biti prva transformacija. Konačna transformacija stoga glasi:

$$Transformacija\ scene = translacija \cdot rotacija \cdot skaliranje \tag{2.2.}$$

Drugi korak: transformacija kamere

Drugi korak je identičan prvom, samo što se koriste koordinate promatrača umjesto koordinata objekta, te se koriste inverzi matrica koji se množe obrnutim redoslijedom. Rezultirajuća matrica može transformirati koordinate iz scenskog u promatračev referentni sustav.

Kamera gleda u smjeru svoje z-osi, x-os obično predstavlja smjer lijevo, dok je y-os smjer gore.

$$\begin{bmatrix} 1 & 0 & 0 & -x \\ 0 & 1 & 0 & -y \\ 0 & 0 & 1 & -z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

-inverzna translacija objekta (inverz translacije je translacija u suprotnom smjeru)

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

-inverz rotacije oko x-osi (inverz rotacije je rotacija u suprotnom smjeru)

$$\begin{bmatrix} \cos\beta & 0 & -\sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

-inverz rotacije oko y-osi

$$\begin{bmatrix} \cos\gamma & -\sin\gamma & 0 & 0 \\ \sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

-inverz rotacije oko z-osi

Matrice dobivene iz prethodna dva koraka se mogu pomnožiti čime se dobiva matrica kojom se mogu transformirati koordinate točke iz referentnog sustava objekta u referentni sustav promatrača.

$$Transformacija\ kamere = inverzna\ rotacija \cdot inverzna\ translacija \quad (2.3.)$$

$$Trenutna\ transf. = transformacija\ kamere \cdot transformacija\ scene \quad (2.4.)$$

Treći korak: transformacija perspektive

Rezultirajuće koordinate bi već bile dobre za izometrijsku projekciju ili nešto slično, ali za realistično renderiranje je potreban dodatni korak da bi se ispravno simulirala distorzija perspektive.

Doista, ova simulirana perspektiva je glavni način predodžbe udaljenosti prema promatraču u simuliranom prikazu. Distorzija perspektive se dobiva korištenjem sljedeće 4x4 matrice:

$$\begin{bmatrix} {}^{1}/_{\tan\mu} & 0 & 0 & 0 \\ 0 & {}^{1}/_{\tan\nu} & 0 & 0 \\ 0 & 0 & \frac{B+F}{B-F} & \frac{-2BF}{B-F} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

gdje je μ kut između linije koja izlazi iz kamere duž njene z-osi i ravnine koja prolazi kroz kameru i desni rub zaslona, dok je ν kut između iste linije i ravnine koja prolazi kroz kameru i gornji rub zaslona. Ova projekcija bi trebala izgledati pravilno ukoliko se gleda sa jednim okom, pri čemu je oko promatrača u pravcu linije kroz središte zaslona, te se μ i ν mogu fizički mjeriti uz pretpostavku da je oko kamera. Na uobičajenim zaslonima računala $\tan\mu$ iznosi otprilike ${}^{1}/_{3}\tan\nu$, dok $\tan\nu$ može biti od 1 do 5, ovisno koliko je daleko promatrač od zaslona.

F je pozitivan broj koji predstavlja udaljenost promatrača od prednje odsječne ravnine (eng. *clipping plane*), koja određuje najmanju moguću udaljenost objekta od kamere. B je pozitivan broj koji predstavlja udaljenost do stražnje odsječne ravnine, najdalje koliko neki objekt može biti. Ukoliko objekt može biti neograničenu udaljenost od kamere, B može biti beskonačan, pri čemu je $(B+F)/(B+F) = 1$ i $-2BF/(B-F) = -2F$.

Ukoliko se ne koristi Z-buffer i svi su objekti ispred kamere, može se jednostavno koristiti 0 umjesto $(B+F)/(B+F)$ i $-2BF/(B-F)$.

Sve se izračunate matrice mogu zajedno pomnožiti da bi se dobila konačna matrica transfomacije. Svaka točka (opisana vektorom triju koordinata) se može pomnožiti ovom matricom, te se izravno dobivaju koordinate zaslona na kojima je potrebno renderirati točku.

Vektor je potrebno proširiti na četiri dimenzije korištenjem homogenih koordinata:

$$\begin{bmatrix} x' \\ y' \\ z' \\ \omega' \end{bmatrix} = [Transf.perspektive] \cdot [Transf.kamere] \cdot [Transf.scene] \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \qquad (2.5)$$

U bibliotekama za računalnu grafiku, poput OpenGL-a, potrebno je dodavati matrice u obrnutom redoslijedu kako se primjenjuju, odnosno prvo transformaciju perspektive, pa transformaciju kamere, te nakon toga transfomaciju objekta, jer biblioteka procesuira transformacije u obrnutom redoslijedu kako su davane. Ovo je korisno, jer se transformacija scene obično češće mijenja od transformacija kamere, dok se transformacija kamere češće mijenja od transformacije perspektive. Tako se može npr. dohvatiti transformacija scene sa stoga transformacija, te se pomnožiti sa novom transformacijom scene, bez da se išta promijeni na transformacijama kamere i perspektive.

Potrebno je zapamtiti da su $\{x'/\omega', y'/\omega'\}$ konačne koordinate, pri čemu $\{-1,-1\}$ obično predstavlja donji lijevi kut zaslona, $\{1,1\}$ donji desni, $\{1,-1\}$ donji desni i $\{-1,-1\}$ predstavlja gornji lijevi kut zaslona. Ukoliko rezultirajuća slika ispadne naopačke, dovoljno je zamijeniti vrh i dno.

Ako se koristi Z-buffer, $\{z'/\omega'\}$ koji je jednak -1 odgovara prednjem dijelu Z-buffera, dok 1 odgovara stražnjem dijelu Z-buffera. Ukoliko je prednja odsječna ravnina preblizu, Z-buffer sa konačnom vrijednošću će biti neprecizniji. Isto vrijedi i za stražnju odsječnu ravninu, ali u manjem obujmu. Z-buffer radi ispravno sa beskonačnom udaljenošću stražnje odsječne ravnine, ali ne i sa prednjom odsječnom ravninom na udaljenosti 0.

Objekti bi se trebali renderirati samo gdje vrijedi $-1 \leq z'/\omega' \leq 1$. Ukoliko je manje od -1, objekt je ispred prednje odsječne ravnine. Ukoliko je veće od 1, objekt je iza stražnje odsječne ravnine. Za crtanje običnog jednobojnog trokuta $\{x'/\omega', y'/\omega'\}$ za tri kuta sadržava dovoljno informacija. Za crtanje teksturiranog trokuta, pri čemu je jedan od kuteva trokuta iza kamere, sve su koordinate $\{x', y', z', \omega'\}$ potrebne za sve tri točke, u suprotnom tekstura nebi imala pravilnu perspektivu, te se točka iza kamere nebi pojavila na ispravnoj lokaciji. Projekcija trokuta čija je jedna točka iza kamere tehnički zapravo nije trokut, jer je njena površina beskonačna i suma dva kuta je veća od 180°, pri čemu je treći kut posljedično tome negativan. Također, ako je točka na ravnini koja prolazi normalom kamere prema smjeru gledanja, ω' je 0, te $\{x', y', z', \omega'\}$ postaje beznačajno.

2.4 Z-buffering

U 3D računalnoj grafici, z-buffering predstavlja rukovođenje dubinskim koordinatama slika, obično izvršeno na razini hardware-a, a ponekad u software-u. To je jedno od rješenja za problem vizualnosti, gdje je potrebno definirati koji su elementi vidljivi, a koji ne. Crtačev algoritam je još jedna uobičajena solucija, iako manje efikasna, za rukovođenje neprozirnim elementima scene.

Kada je objekt renderiran 3D grafičkom karticom, dubina (eng. *depth*) generiranog pixela (z-koordinata) je spremljena u tzv. z-buffer. Ovaj buffer je obično uređen kao dvodimenzionalni niz (x-y) sa jednim elementom za svaki pojedini pixel na zaslonu. Ukoliko je potrebno renderirati neki drugi objekt na istom pixelu, grafička kartica uspoređuje dvije dubine, te odabire onaj koji je "bliže" promatraču. Odabrana dubina je tada spremljena u z-buffer, zamjenjujući staru. Na kraju će z-buffer omogućiti grafičkoj kartici da ispravno reproducira dubinsku percepciju: bliži objekt skriva udaljeniji.

Zrnatost (eng. *granularity*) z-buffera ima visok utjecaj na kvalitetu scene: 16-bitni z-buffer može rezultirati sa tzv. artefaktima (eng. *artifacts*) kada su dva objekta jako blizu jedan drugome. 24-bitni ili 32-bitni z-buffer se ponaša mnogo bolje. 8-bitni z-buffer se gotovo nikada ne koristi jer ima jako malenu preciznost.

Također, preciznost udaljenosti z-buffera nije raspoređena jednako duž same udaljenosti. Bliže vrijednosti su mnogo preciznije (stoga se mnogo točnije prikazuju bliži objekti) od udaljenijih vrijednosti. Obično je ovo poželjno, ali ponekad će se udaljeniji objekti pojaviti kao artefakti. Varijacija z-bufferinga koja rezultira mnogo ujednačenijom distribucijom preciznosti se zove w-buffering.

Na početku nove scene, z-buffer se mora očistiti na definiranu vrijednost, obično 1.0, jer je ova vrijednost gornja granica (na skali od 0 do 1) za dubinu, što znači da niti jedan objekt nije prisutan u narednom frustumu gledanja (eng. *viewing frustum*).

Opseg vrijednosti dubina u prostoru kamere koje je potrebno renderirati se često definira između *near* i *far* vrijednosti z. Nakon transformacije perspektive, nova vrijednost z' je jednaka:

$$z' = \frac{far + near}{far - near} + \frac{1}{z}\left(\frac{-2 \cdot far \cdot near}{far - near}\right) \tag{2.6}$$

Rezultirajuće vrijednosti z' su normalizirane između vrijednosti -1 i 1, gdje je *near* ravnina na -1, dok je *far* ravnina na 1. Vrijednosti izvan ove domene odgovaraju točkama koje nisu unutar frustuma gledanja, te se stoga nebi smjele renderirati.

Da bi se z-buffer implementirao, vrijednosti z' su linearno interpolirane kroz prostor zaslona između točaka poligona i ove srednje vrijednosti su obično spremljene u z-buffer na fiksnoj točki. Vrijednosti z' su grupirane mnogo gušće blizu *near* ravnine, te mnogo raspršenije na većim udaljenostima, čime se dobiva bolja preciznost bliže kameri. Što je *near* ravnina bliže kameri, to je manja preciznost na većim udaljenostima. *Near* ravnina koja je postavljena preblizu je čest uzrok neželjenih artefakata kod udaljenijih objekata.

Da bi se implementirao w-buffer, stare vrijednosti z unutar prostora kamere, ili w, se spremaju u buffer, obično kao decimalne vrijednosti. Međutim, ove se vrijednosti ne mogu linearno interpolirati kroz prostor zaslona između točaka – obično se moraju obrnuti, interpolirati, pa opet obrnuti. Rezultirajuće vrijednosti w, za razliku od z', su podjednako raspoređeni između *near* i *far*.

Da li će z-buffer ili w-buffer rezultirati u boljoj slici ovisi o aplikaciji.

2.5 Shader

Shader je računalni program koji se koristi u 3D računalnoj grafici za određivanje konačnih površinskih svojstava objekta ili slike. To obično uključuje proizvoljno složene algoritme apsorpcije svjetlosti, difuzije, mapiranja tekstura, refleksije, refrakcije, osjenčavanja, premještanja površina te naknadno-procesuiranim efektima.

Po dizajnu, shaderi su idealni kandidati za paralelno izvršavanje od strane više grafičkih procesora, koji se nalaze na grafičkoj kartici, što omogućava skalabilno multiprocesiranje, te smanjuje opterećenje na CPU kada je u pitanju renderiranje scena. S obzirom na specifične ciljeve koje moraju izvršiti, shaderi se obično pišu vlastitim programskim jezikom, posebno dizajniranim za unutrašnje snage i slabosti različitih računskih modela. Iako su ograničeni u nekim dijelovima kada se uspoređuju sa tradicionalnim pristupom, paralelna arhitektura shadera je iskorištena za kombinaciju visoko-skalabilne procesorske snage sa fleksibilnošću programabilnih uređaja, što predstavlja blagodat kada su u pitanju rastući zahtjevi za grafičkom kvalitetom.

Rastuće perfomanse i programabilnost arhitektura zasnovanih na shaderima privukla je istraživače koji pokušavaju iskoristiti novi paralelni model za opću namjenu na grafičkim procesorima. Pokazalo se da se shaderi mogu koristiti za obradu različitih tipova informacija, a ne samo specifičnih zadataka renderiranja. Ovaj novi programski model, koji predstavlja *stream* procesiranje, omogućuje brze kalkulacije pri vrlo niskoj cijeni koji će raditi na širokoj bazi (npr. na kućnom PC-u).

Shaderi su jednostavni programi koji opisuju svojstva tjemena ili piksela. Tjemeni shaderi (eng. *vertex shader*) opisuju svojstva (poziciju, teksture, koordinate, boju i dr.) tjemena, dok piksel shaderi (eng. *pixel shader*) opisuju svojstva (boju, z-dubinu i prozirnost) piksela. Shaderi zamjenjuju dio grafičkog hardware-a koji se naziva Fiksni Funkcijski Cjevovod (eng. *Fixed Function Pipeline*, FFP), koji se tako naziva jer izvršava mapiranje osvjetljenja i tekstura u tzv. hard-code maniri. Shaderi pružaju programabilnu alternativu ovom hard-kodiranom pristupu.

Osnovni koraci pri grafičkoj protočnoj obradi ili cjevovodu (eng. *graphics pipeline*) su sljedeći:

- CPU šalje instrukcije (kompajlirani programi kodirani shading jezikom) i geometrijske podatke grafičkom procesoru na grafičkoj kartici
- Geometrija se transformira unutar tjemenog shadera
- Ukoliko je aktivan geometrijski shader unutar grafičkog procesora, dolazi do nekih izmjena u geometriji scene
- Ukoliko je aktivan teselacijski shader u grafičkom procesoru, geometrije u sceni se mogu podijeliti na manje dijelove
- Izračunata geometrija se triangulira (dijeli u trokute)
- Trokuti su podijeljeni u fragmentne četverokute
- Fragmentni četverokuti se oblikuju prema fragmentnom shaderu
- Izvršava se test Z-dubine, fragmenti koji prođu test se ispisuju na zaslon i mogu biti ubačeni u frame-buffer

Grafički cjevovod koristi ove korake da bi transformirao trodimenzionalne (ili dvodimenzionalne) podatke u korisne dvodimenzionalne podatke koji se mogu prikazati na zaslonu. Općenito, to je velika piksel matrica ili "frame buffer".

Piksel shaderi

Piksel shaderi (ponekad nazivani i fragmentni shaderi) izračunavaju boju i ostala svojstva svakog "fragmenta" (tehnički naziv za pojedini piksel). Piksel shaderi mogu biti od najjednostavnijih (oni koji kao izlaz daju samo jedan piksel kao vrijednost boje) do složenijih koji određuju i vrijednosti osvjetljenja, mapiraju ispupčenja, sjene, refleksiju, translucenciju i dr. Oni mogu izmijeniti Z-dubinu fragmenta (za Z-buffering) ili kao izlaz dati više boja ako je aktivno više

elemenata za renderiranje. U 3D grafici, piksel shader sam ne može generirati jako složene efekte, jer djeluje nad samo jednim fragmentom, bez znanja o geometriji scene. Međutim, piksel shaderi imaju znanje o koordinatama scene koje se renderiraju i mogu uzorkovati zaslon i susjedne piksele ukoliko je sadržaj čitavog zaslona poslan kao tekstura u shader. Ova tehnika omogućuje široki spektar naknadno obrađenih dvodimenzionalnih efekata, poput zamaglenja, ili detekcije rubova.

Tjemeni shaderi

Tjemeni shaderi se izvršavaju jednom za svako tjeme objekta koje je dano grafičkom procesoru. Svrha je transformacija 3D pozicije svakog tjemena unutar virtualnog prostora u 2D koordinatu na kojoj se prikazuje na zaslonu. Tjemeni shaderi mogu manipulirati svojstvima poput pozicije, boje i koordinata tekstura, ali ne mogu generirati nova tjemena. Izlaz tjemenog shadera se šalje u sljedeću fazu grafičkog cjevovoda, koja je ili geometrijski shader (ukoliko postoji) ili u tzv. raster. Tjemeni shaderi omogućuju snažnu kontrolu nad detaljima pozicije, kretnje, osvjetljenja i boje u bilo kojoj sceni koja sadržava 3D modele.

2.6 OpenGL i Direct3D

OpenGL (Open Graphics Library) je standardna specifikacija koja definira cross-platform API za pisanje aplikacija koje generiraju 3D računalnu grafiku (kao i 2D računalnu grafiku). Sučelje se sastoji od 250 različitih funkcijskih poziva koje se mogu koristiti za crtanje kompleksnih trodimenzionalnih scena od jednostavnih geometrijskih primitiva (eng. *primitive*) odnosno najjednostavnijih geometrijskih oblika kojima računalo može rukovoditi. OpenGL je razvijen od strane Silicon Graphics-a i popularan je u industriji računalnih igara gdje mu konkurenciju stvara Direct3D prilagođen za Microsoftove platforme. OpenGL se naširoko koristi u CAD-u, virtualnoj stvarnosti, znanstvenim vizualizacijama, informacijskim vizualizacijama, simulacijama letova i razvoju računalnih igara.

U svojoj suštini OpenGL je specifikacija, odnosno to je jednostavno dokument koji precizno opisuje set funkcija i što one moraju raditi. Iz ovih specifikacija proizvođači hardware-a kreiraju implementacije – biblioteke funkcija stvorenih da odgovoaraju opisu funkcija u OpenGL specifikaciji, iskorištavajući hardware ubrzavanje gdje god je to moguće. Proizvođači hardware-a moraju zadovoljiti određene testove da bi se njihove implementacije mogle kvalificirati kao

OpenGL implementacije. Efikasne OpenGL implementacije isporučene od strane proizvođača postoje za Mac OS, Windows, Linux, mnoge Unix platforme i PlayStation 3.

OpenGL ima dvije svrhe:

- Skrivanje kompleksnosti sučelja između različitih 3D akceleratora, tako što pruža programeru jedinstveni uniformni API
- Skrivanje različitih mogućnosti raznih platformi, tako što zahtjeva da sve implementacije podržavaju cijeli set OpenGL funkcija

Osnovna operacija OpenGL-a je primanje primitiva poput točaka, linija i poligona, te njihova pretvorba u piksele. To se radi u grafičkom cjevovodu poznatom kao OpenGL state machine[1]. Većina OpenGL naredbi šalje primitive u grafički cjevovod ili opisuju način kako će cjevovod procesirati te primitive. OpenGL je proceduralni API niže razine, koji zahtjeva od programera da diktira točne korake koji su potrebni da bi se renderirala scena. Ovo se razlikuje od opisnih API-ja, gdje programer samo treba opisati scenu, te ostavlja biblioteci zadaću obrade detalja za renderiranje. OpenGL-ov nisko-razinski dizajn zahtjeva od programera dobro poznavanje grafičkog cjevovoda, ali također pruža određenu razinu slobode pri implementaciji novih algoritama renderiranja.

Direct3D je dio Microsoft-ovog DirectX API-ja koji je dostupan samo u Windows operacijskim sustavima, te je osnova za grafičke API-je u Xbox i Xbox 360 konzolama. Direct3D se koristi za renderiranje 3D grafike u aplikacijama gdje su perfomanse vrlo važne, poput video igara. Direct3D koristi hardwaresko ubrzanje ukoliko je ono dostupno na grafičkoj kartici.

Direct3D je 3D API, tj. sadrži mnogo naredbi za 3D renderiranje, ali samo nekoliko za renderiranje 2D grafike[2]. Microsoft teži konstantnom ažuriranju Direct3D-a da bi podržavao zadnje tehnologije dostupne na 3D grafičkim karticama. Direct3D pruža potpunu emulaciju vertex-software-a, ali ne i emulaciju piksel software-a za svojstva koja nisu dostupna u hardware-u. Npr., ako program kodiran korištenjem Direct3D-a zahtjeva piksel shadere i grafička kartica na korisnikovom računalu ne podržava tu mogućnost, Direct3D ga neće emulirati.

[1] https://www.opengl.org/documentation/specs/version1.1/state.pdf
[2] Microsoft DirectX SDK Readme (December 2005)(http://msdn.microsoft.com/directx/sdk/read- 59 mepage/default.aspx

3 PRIMJENA UMJETNE INTELIGENCIJE

Inteligencija se uobičajeno smatra kao sposobnost prikupljanja znanja, te korištenja tog znanja za rješavanje složenih problema. Umjetna inteligencija je znanost koja proučava i razvija inteligentne strojeve i računalne programe koji mogu učiti, prosuđivati, prikupljati znanje, komunicirati, te upravljati i percipirati objekte. John McCarthy definirao je pojam 1956. kao granom računarstva zaduženoj za razvoj računala koja se ponašaju kao čovjek. To je proučavanje algoritama koji omogućuju percepciju razuma, te djelovanje.

Umjetna inteligencija se razlikuje od psihologije jer je orijentirana na računalne kalkulacije, dok se istovremeno razlikuje od računarstva po tome što je orijentirana na percepciju, razumijevanje i djelovanje. Ona čini strojeve pametnijima i korisnijima, a radi uz pomoć umjetnih neuronskih mreža i znanstvenih teorema (matematičke logike). Tehnologije umjetne inteligencije su se razvile do te mjere da pružaju stvarne koristi u različitim područjima primjene. Neke od većih područja primjene čine ekspertni sustavi, obrada prirodnih jezika, prepoznavanje govora, robotika i senzori, računalni vid i prepoznavanje scena, računalno-potpomognute inteligentne instrukcije, neuronsko računarstvo i dr.

Slika 3.1. Cilj AI-ja nije samo da oponaša čovjeka, već i da obavlja zadatke bolje od njega

Prednost umjetne inteligencije naspram one prirodne je u tome što je trajna, konzistentna, jeftinija, ima mogućnost dupliciranja i širenja, može se dokumentirati i može obaviti određene zadatke mnogo brže i bolje od čovjeka.

3.1 Područja umjetne inteligencije

A. **Prepoznavanje jezika:** Sposobnost razumijevanja i odgovaranja na prirodni jezik. Prevođenje izgovorenog jezika u pisani oblik, te prevođenje iz jednog prirodnog jezika u drugi.
    - Prepoznavanje govora
    - Obrada semantičkih informacija
    - Odgovaranje na pitanja
    - Vraćanje informacija
    - Prevođenje jezika

B. **Samoučeći i prilagodljivi sustavi:** Prilagodba ponašanja s obzirom na prošla iskustva, te razvoja općih pravila koji se tiču okoline.
    - Kibernetika
    - Formacija koncepata

C. **Rješavanje problema:** Formuliranje problema u prikladnom obliku, planiranje njegovog rješenja, te znati kada je potrebna nova informacija, te kako ju dobiti.
    - Zaključivanje
    - Interaktivno rješavanje problema
    - Automatsko pisanje programa
    - Heurističko (istraživačko) pretraživanje

D. **Vizualna percepcija:** Analiziranje opažene scene povezujući ju s internim modelom koji predstavlja "znanje o okolini" percipirajućeg organizma. Rezultat ove analize je strukturirani set odnosa između različitih entitea unutar scene.
    - Prepoznavanje uzorka
    - Analiza scene

E. **Modeliranje:** Sposobnost razvoja internog opisa i seta transformacijskih pravila koja se mogu koristiti za predviđanje ponašanja i odnosa između skupine stvarnih objekata ili entiteta.
    - Opisni problem za sustave rješavanja problema
    - Modeliranje prirodnih sustava (ekonomski, socijalni, ekološki, biološki itd.

F. **Robotika:** Kombinacija gornje navedenih sposobnosti uz sposobnost kretanja unutar okoline, te manipulacije objektima.
    - Istraživanje
    - Transportacija/navigacija
    - Industrijska automatizacija (npr. upravljanje procesima, zadaci montiranja, izvršni zadaci)
    - Sigurnosni sustavi
    - Vojska
    - Kućanstva
    - Drugo (agrikultura, pecanje, rudarenje, sanitacija, građevina itd.)

G. **Igre:** Sposobnost prihvaćanja formalnog seta pravila za igre poput šaha, igre dame i dr., te prevođenje tih pravila u opis ili strukturu koja omogućuje rješavanje problema i vještine učenja koja će se koristiti za postizanje adekvatne razine izvođenja.

3.2 Primjena umjetne inteligencije

Regulatori elektroenergetskih sustava

Regulatori elektroenergetskih sustava (eng. Power System Stabilizers, PSS) koriste se još od 1960-tih za prigušivanje elektromehaničkih oscilacija. PSS je dodatni upravljački sustav koji se često dodaje kao dio uzbudnog upravljačkog sustava. Osnovna funkcija PSS-a je dodavanje signala uzbudnom sustavu, stvarajući električne okretne momente rotoru u fazi sa razlikama brzina koji prigušuju oscilacije snaga. Umjetna inteligencija ima mogućnost djelovanja u upravljačkim sustavima visoke nelinearnosti.

Detekcija mrežnih upada

Detekcija mrežnih upada (eng. Intrusion Detection Systems, IDS) koristi razne tehnike umjetne inteligencije pri zaštiti računala i komunikacijskih mreža od uljeza i neželjenih upada. IDS je proces nadziranja događaja koji se javljaju unutar mreže, te detekcije znakova upada.

Umjetna neuronska mreža (eng. *artificial neural network*) je matematički model koji se sastoji od međusobno povezane grupe umjetnih neurona koja obrađuje informacije. Umjetne neuronske mreže se u IDS-u koriste za modeliranje složenih odnosa između ulaza i izlaza ili za pronalazak uzoraka u podacima. Pri tome neuron izračunava sumu množeći ulaze sa težinom, te dodaje neki prag. Rezultat se prenosi sljedećim neuronima.

Medicina

Umjetna inteligencija ima potencijala za primjenu u gotovo svakom polju medicine. Neizrazita logika (end. *fuzzy logic*) predstavlja metodologiju rukovođenja podacima koja dopušta dvosmislenost, te je stoga posebno pogodna za primjenu u medicini. Ona dohvaća te koristi koncept neizrazitosti u računalno-efektivnom obliku. Najčešće područje primjene za ovu teoriju leži u medicinskoj dijagnostici i u manjem opsegu, u opisu bioloških sustava. Neizraziti ekspertni sustavi koriste strukturu serije ako-onda (eng. *if-then*) pravila prilikom modeliranja.

1970. CDSS je razvio ekspertni sustav za identifikaciju bakterija koje uzrokuju infekcije, te preporučivanje antibiotika za liječenje tih infekcija prilikom medicinske dijagnoze stanja pacijenta. Pathfinder, koji je koristio Bayesove mreže, pomagao je patolozima pri preciznijoj dijagnozi bolesti limfnih čvorova. Umjetna inteligencija također je korisna pri računalno-potpomognutoj detekciji

tumora u medicinskim snimkama. Ovakvi pristupi pomažu prilikom dijagnoze različitih vrsta raka, te urođenih srčanih mana.

Baze podataka u računovodstvu

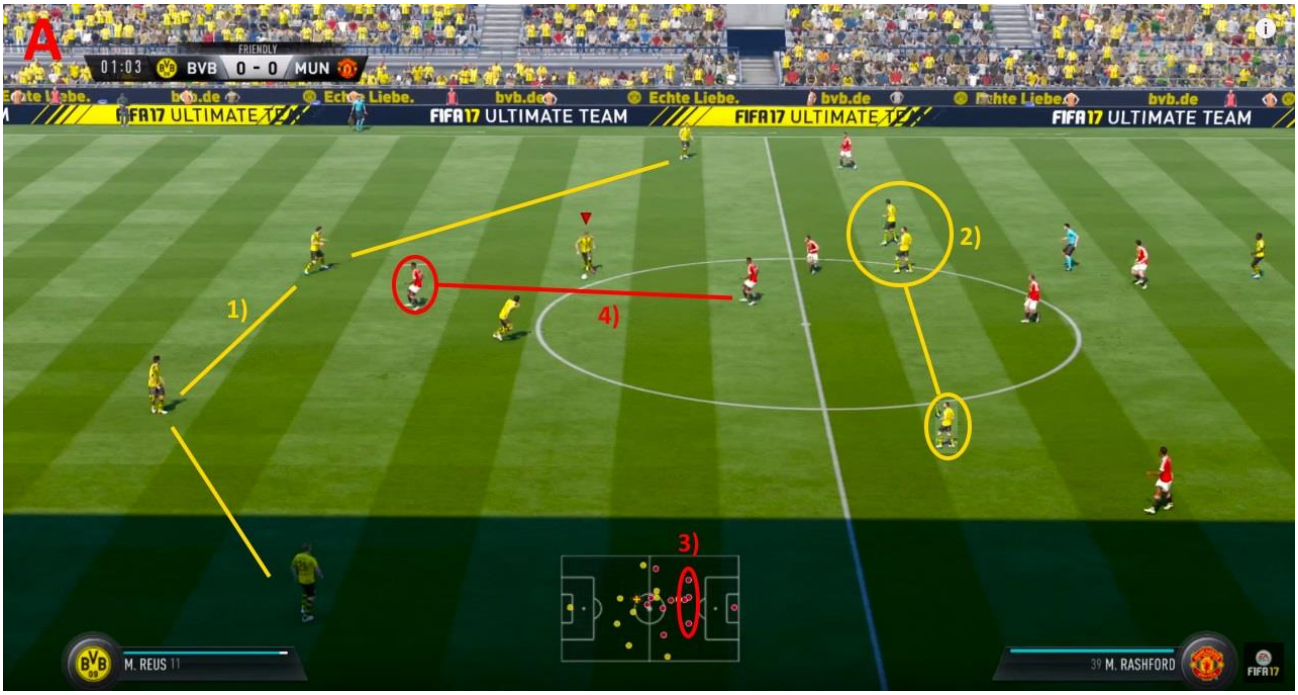
Upotreba umjetne inteligencije se pokazala kao osnovom pri ublaženju problema koji se mogu pojaviti u računovodstvenim bazama podataka. Potrebe donositelja odluka nisu ispunjene čistim podacima u računovodstvu. Čovjek ne razumije ili ne može procesirati digitalizirane baze podataka. Sustavi se ne mogu jednostavno koristiti, jer je fokus usmjeren na numeričkim podacima.

Integriranje inteligentnih sustava sa računovodstvenim bazama podataka može pomoći prilikom istrage velike količine podataka, sa ili neovisno o donositelju odluka. Stoga sustavi mogu analizirati podatke, te pomoći korisnicima u razumijevanju ili interpretaciji transakcija za određivanje računovodstvenih događaja detektiranih od strane sustava. Sa umjetnom inteligencijom spremamo i dohvaćamo znanje u prirodnom jeziku. Postoji veći fokus na simboličke ili tekstualne podatke nego na numeričke. Umjetna inteligencija i ekspertni sustavi grade inteligenciju unutar baze podataka za pomoć korisnicima. Takvi modeli pomažu korisnicima sortiranjem velike količine podataka, bez njihovog direktnog utjecaja. Takvi modeli također pomažu donositeljima odluka unutar vremenskih rokova, predlažu alternative prilikom pretraživanja, te procjene podataka.

3.3 Umjetna inteligencija u računalnim igrama

Kada se sustavi za grafičko renderiranje, produciranje zvuka, korisničko sučelje te sustav umjetne inteligencije povežu u jedno, dobiva se očekivana razina zabave koja se očekuje od jedne računalne igre. Umjetna inteligencija je jedno od najvažnijih dijelova svake računalne igre, te bi bez nje igranje bilo nezanimljivo. Ona rješava uobičajene probleme, te pruža igri određenu složenost. Specifično govoreći, to se odnosi na pronalaženje puta kretanja lika koji nije u kontroli igrača (eng. *non-playing character*, NPC), donošenje njegovih odluka, te učenje. Postoji nekoliko načina na koji umjetna inteligencija pridonosi modernim računalnim igrama. Najočitiji su kretnje likova, simulirana percepcija, analiza situacija, razumijevanje okoline, učenje, grupna koordinacija, alokacija resursa, upravljanje, skupljanje, odabiranje mete i mnogi drugi. Čak kontekstno-ovisne animacije i audio koriste umjetnu inteligenciju.

Umjetna inteligencija rješava tri česta problema: kretnja NPC-a, NPC-ovo donošenje odluka i učenje. Četiri korištene tehnike umjetne inteligencije su pronalaženje puta (eng. *path finding*), Bayesove mreže, neizrazita logika (eng. *fuzzy logic*) i genetski algoritmi koji pomažu video igri u pružanju NPC-u sposobnost njegovog djelovanja u okolini.



Slika 3.2. Jedno od najnaprednijih primjera AI-a u video igrama je u FIFA 17

Kretanje NPC-a korištenjem pronalaska puta

Video igra koja ima umjetnu inteligenciju mora omogućiti liku bez kontrole igrača da se kreće kroz virtualni svijet igre. Primjerice, ukoliko je igrač na jednoj strani zgrade, a čudovište na drugoj, kojim će putem kroz zgradu čudovište doprijeti do igrača? Ovo je problem kretanja NPC-a. Različite metode umjetno-inteligentnog pretraživanja se koriste za pronalazak puta u računalnim igrama. A* algoritam je najčešće korišten za pregovaranje puta zbog svoje fleksibilnosti i zato što odabire najkraći put između dvije točke. Tipični A* algoritmi imaju tri osnovna svojstva: fitnes, cilj i heuristika (eng. fitness, goal, heuristic, f,g,h). Cilj je cijena kretanja od početnog čvora (eng. *node*) do nekog čvora između cilja. Heuristika je procijenjena cijena da se dođe od tog čvora do cilja. Fitness je suma cilja i heuristike, odnosno ukupna procijenjena cijena puta kroz ovo čvorište. A* algoritam također sadržava otvorenu listu (eng. *open list*) čvorova koji još nisu bili istraženi i zatvorenu listu čvorova koji su bili istraženi.

Pseudo kod A* algoritma glasi:

1. Neka je P = početna točka
2. Dodijeli f,g i h vrijednosti u P
3. Dodijeli P u otvorenu listu. Trenutno je P jedini čvor u otvorenoj listi.
4. Neka je B = najbolji čvor u otvorenoj listi (najbolji čvor ima najmanju vrijednost f)
    a. Ako je B ciljni čvor, odustani. Put je pronađen.
    b. Ako je otvorena lista prazna, odustani. Put je pronađen.
5. Neka je C = važeći čvor povezan sa B
    a. Dodijeli f,g i h vrijednosti u C.
    b. Provjeri da li je C u otvorenoj i zatvorenoj listi.
        i. Ako jest, provjeri da li je novi put efikasniji (manja vrijednost f)
            1. Ako jest, ažuriraj put.
        ii. U suprotnom, dodaj C u otvorenu listu.
    c. Ponovi korak 5 za sve važeće nasljednike od B.
6. Pomakni B iz otvorene liste u zatvorenu listu i ponovi od koraka 4.

Donošenje odluka korištenjem Bayesovih mreža

U prethodnom primjeru odabira puta čudovišta do igrača, drugačiji problem je potrebno riješiti prije početka odabiranja puta. Problem glasi: da li čudovište uopće zna da je igrač prisutan u zgradi? Ako bi dizajneri video igara dali NPC-u potpune informacije o virtualnom svijetu igre, tada igranje nebi bilo zanimljivo. Ovo je primjer donošenja odluka NPC-a (eng. *NPC Decision making*). U ovom slučaju potrebna je umjetna inteligencija da bi NPC djelovao na ljudski način. Kada igrač uđe u zgradu s druge strane, čudovište neće biti svjesno njegove prisutnosti jer se između njih nalazi zid. Ako igrač uđe u zgradu i pri tome načini neku buku, tada će čudovište "osjetiti" njegovu prisutnost, te započeti odabir najkraćeg puta na način kako je prethodno objašnjeno. Jedna tehnika umjetne inteligencije koja se koristi za implementaciju ovakvog ponašanja je Bayesova mreža. Ona pomaže NPC-u pri izvođenju složenih promišljanja na ljudski način. U ovoj tehnici računalo izračunava vjerojatnost da će čudovište osjetiti igrača ako je igrač ušao u zgradu.

Učenje NPC-a

Računalne igre koriste genetske algoritme umjetne inteligencije (eng. *Artificial Intelligence Genetic Algorithms*) za pokušaj implementacije sposobnosti učenja u NPC. Genetski algoritam radi na sljedeći način:

1. Kreiraj populaciju prve generacije slučajnih organizama
2. Testiraj ih na problemu koji se rješava, te ih rangiraj po fitnesu. Ako su najbolji organizmi dosegli naše ciljeve izvedbe, tada stani.
3. Uzmi najbolje izvođače, te ih spari dodavajući genetske operatore poput križanja i mutacije. Dodaj nekoliko novih slučajnih organizama u populaciju da bi se predstavila nova varijacija, te pomoglo pri spriječavanju konvergencije lokalnog maksimuma.
4. Ponovi korak 2.

Genetski algoritmi pokušavaju izgraditi savršenu vrstu, te su vrlo složeni. Ova tehnika umjetne inteligencije se ne koristi u mnogim modernim računalnim igrama jer zahtjeva puno računalnih resursa i vremena za evoluciju vrsta ili NPC-a u nešto značajno.

4 UNITY ENGINE

Game engine je programska platforma (eng. *software framework*) namijenjena za razvoj video igara. Svrha game enginea je da apstrahira često ili stalno izvođene zadatke vezane uz samu igru, poput renderiranja grafike, rukovođenje inputom, fizikom, alokacijom resursa i dr., tako da se programer može više fokusirati na specifične stvari koje čine njegovu igru unikatnom, a da manje vremena troši u radu na već gotovim stvarima.

Prikaz i animacija modela, fizika i detekcija kolizije između objekata, input, korisničko sučelje, pa i dio umjetne inteligencije u igri su komponente koje čine engine. Sadržaj igre, specifični modeli i teksture, sami smisao inputa i fizike, te interakcija objekata sa svijetom su komponente koje čine stvarnu igru.

Slika 4.1. Unity logo

Unity je višeplatformski game engine, odnosno omogućuje razvoj video igara za računala, konzole, mobilne uređaje i web stranice. S obzirom na to, Unity koristi sljedeće API-je za renderiranje grafike:

- Direct3D za Windows operacijski sustav i Xbox
- OpenGL za Mac, Linux i Windows
- OpenGL ES za Android i iOS
- Posjedničke (specifične) API-je za konzole

Unity omogućuje jednostavan odabir različitih postavki grafike i rezolucije posebne za svaki od sustava na kojem se igra pokreće, tako da nije potrebno od početka reprogramirati igru primjerice za Android mobilni uređaj nakon što je ona prvotno programirana za Windows.

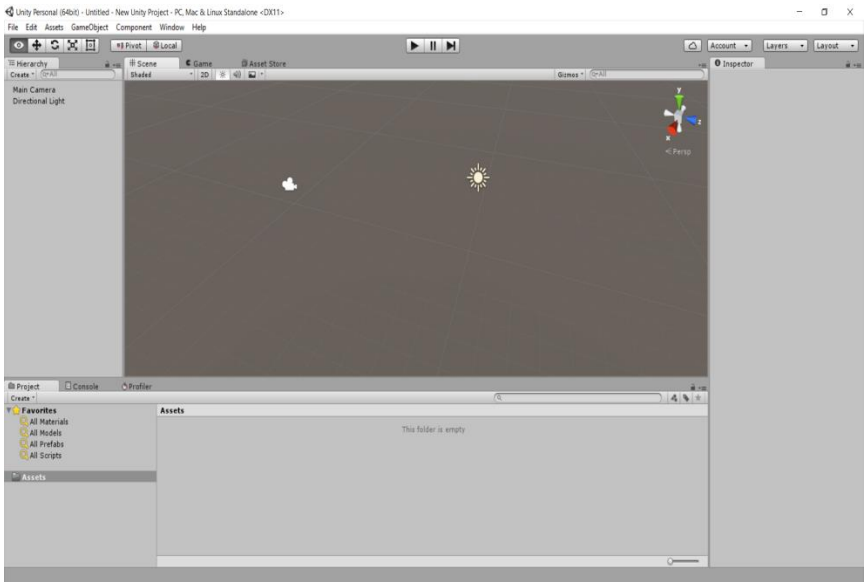
Igre u Unity-u se razvijaju pomoću besplatnog alata Unity Editor (trenutna verzija kada je pisan ovaj rad je 5.4.1f1). Rad u Unity-u bi se mogao podijeliti u nekoliko tema:

- Sučelje i uporaba Editor-a
- Pisanje skripti
- Grafika
- Fizika
- Zvuk
- Animacija
- Korisničko sučelje (UI)
- Navigiranje (pathfinding)
- Umrežavanje (za multiplayer)

Većina ovih tema će biti opisane kroz primjer 3D igre razvijene pomoću navedenog Unity Editor-a. Igra će biti žanra "Tower Defense", u kojoj se periodički stvaraju protivnički tenkovi koji pokušavaju doći do drugog kraja mape, a cilj igrača je što duže ih spriječavati gradnjom kula.

4.1 Unity Editor

Razvoj 3D video igre započinje kreiranjem novog projekta u Unity Editoru, prilikom čega se dobiva prazna scena i 2 osnovna game objekta: glavna kamera i usmjereno svjetlo (slika 4.2.).

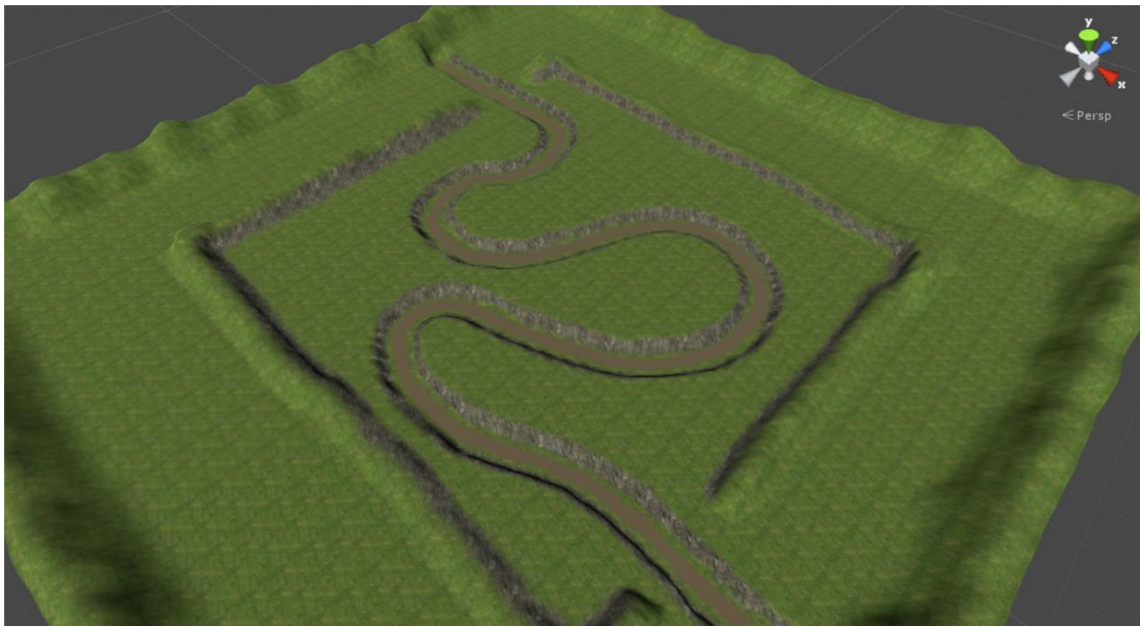


Slika 4.2. Prikaz scene nakon kreiranja novog projekta

Game objekti su entiteti (predmeti) koji čine neku scenu. Bilo što vidljivo (ili nevidljivo) u sceni je game objekt. To može biti teren (eng. *terrain*), svjetlo, kamera, 3D model pa čak i prazan objekt. Svaki objekt može biti sačinjen od više komponenti, no osnovna komponenta svakom objektu je transformacija (eng. *transform*) koja mu definira poziciju, rotaciju i veličinu unutar scene.

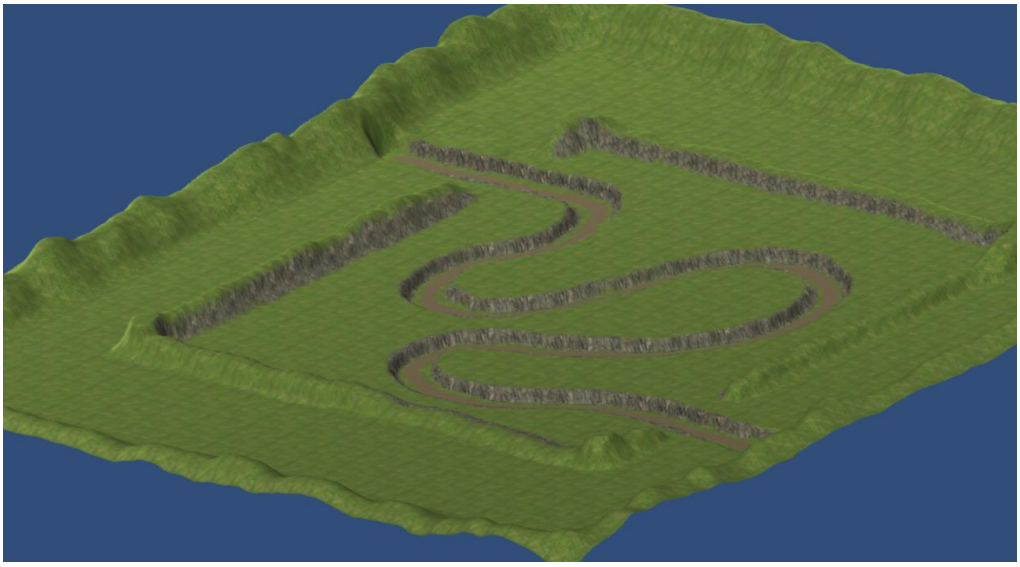
Svi objekti scene su strukturirani u hijerarhiji, tako da jedan objekt može sadržavati više podređenih objekata (eng. *child objects*) koji ga nasljeđuju i imaju relativne transformacije, pa se npr. pomicanjem nadređenog objekta pomiču i svi njegovi podređeni objekti.

Uobičajeno se u sceni prvo kreira osnovni teren i osvjetljenje. U primjeru prethodno opisane igre koristit će se usmjereno svjetlo (eng. *directional light*), koje predstavlja prirodan način osvjetljenja od strane sunca. Na kreirani teren se dodaju teksture, te se modeliraju neravnine. Na slici je prikazan gotov teren na koji će se dodavati ostali objekti.



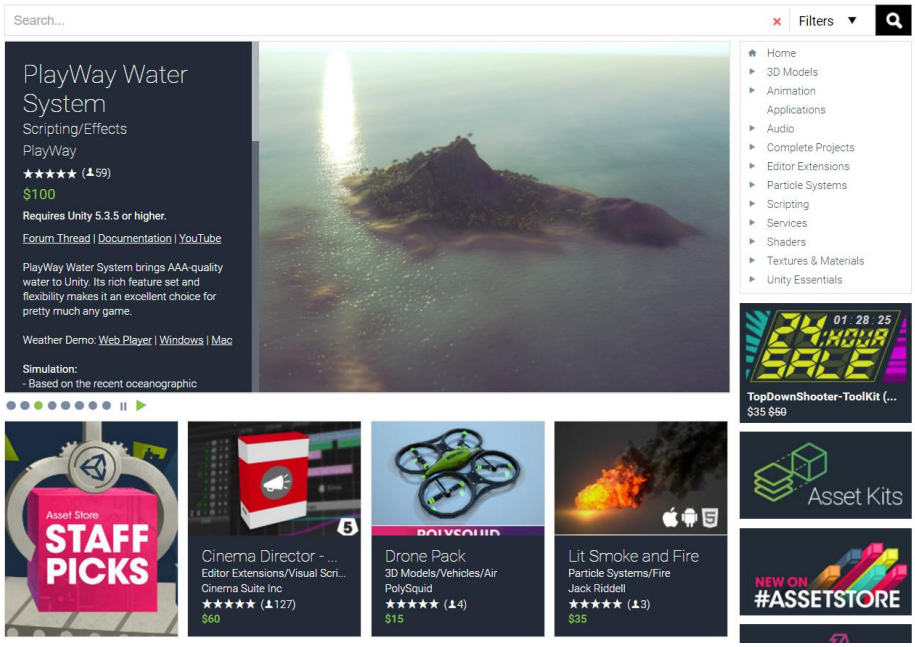
Slika 4.3. Osnovni teksturirani teren i usmjereno osvjetljenje

Sljedeći korak je dodavanje kamere sa perspektivnom projekcijom, koja će predstavljati osnovni pogled unutar igre. Za navedeni tip igre koristit će se RTS kamera, koja ima pogled odozgo, te se pogled pomiče pomicanjem kursora miša prema rubovima zaslona. Nekada se za ovakav tip igre (vrlo često u starim igrama) koristi ortogonalna projekcija, čime se dobije retro stil (Slika 4.4.). Na kameru se također uobičajeno dodaje Skybox komponenta za prikaz neba, jer se time dobije efekt "praćenja oblaka" tamo gdje se igrač kreće, iako je za RTS kameru ono nepotrebno jer se nebo niti ne vidi.



Slika 4.4. Ortogonalna projekcija tipična za starije RTS igre (Red Alert, Age of Empires)

Kada imamo pripremljen teren, osvjetljenje i kameru, dodavaju se objekti. Iako je u Unity Editoru moguće kreirati vlastite modele, najčešće se oni modeliraju u drugim alatima predviđenim za to u kojima je moguće dodavanje i animacija (npr. Blender), te se gotovi modeli uvoze (eng. *import*) unutar projekta. Svi modeli u ovoj igri će biti uvezeni sa službenog online asset store-a (Slika 4.5.). Osim modela, u projekt se uvoze i specijalni efekti, zvukovi i eventualna glazba. Svi preuzeti resursi su besplatni.



Slika 4.5. Unity-ev asset store za preuzimanje resursa

Putanja po kojoj će se kretati tenkovi je predefinirana, tako da nema potrebe za korištenjem nekog oblika navigacije ili traženja najboljeg puta (eng. *pathfinding*). Umjesto toga, na terenu je potrebno naznačiti točke predstavljene praznim objektima prema kojima će se tenkovi kretati.

Osim putanje, na scenu se dodaju i objekti za korisničko sučelje (eng. *User Interface*, UI) i meni, pri čemu se koristi objekt tipa canvas. Canvas je objekt koji služi za renderiranje UI objekata, poput panela, slika, gumbova, klizača i dr. Svi UI objekti moraju biti podređeni canvas objektu. Canvas prije svega definira način na koji će se korisničko sučelje renderirati na različitim veličinama zaslona. Primjerice, iako je poželjno da se na većim rezolucijama objekti renderiraju u više detalja, za korisničko sučelje je poželjnije da ono zadržava fiksnu veličinu, tako da npr. neki gumb ne postane previše sitan na većim rezolucijama.

Zadnja važna stvar prije početka pisanja skripti je definiranje input-a. Iako je moguće koristiti nativne key-kodove za dohvaćanje trenutno pritisnute tipke na tipkovnici ili tipke na mišu, često se inputi apstrahiraju imenima unutar InputManager-a, primjerice tipka escape se može definirati pod nazivom “Cancel“, tipka space pod nazivom “Jump“ ili se lijeva tipka miša može nazvati “Fire“. Ovi nazivi su već zadani prilikom kreiranja projekta, te ih je moguće odmah koristiti.

Konačan izgled scene (uz dodani efekt magle) je prikazan na sljedećoj slici:

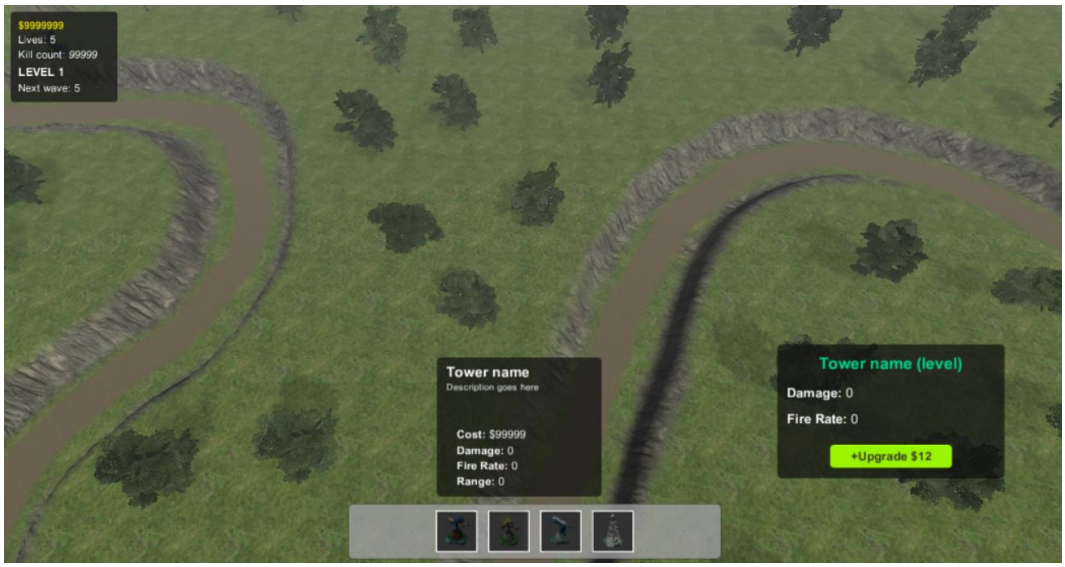
Slika 4.6. Konačna scena igre

Iako će u primjeru ove igre biti korištena samo jedna scena, Unity omogućuje kreiranje više različitih scena u jednom projektu, s čime se može postići igra sa više razina.

Korisničko sučelje

Korisničko sučelje predstavlja dio scene koji omogućava igraču interakciju sa igrom. Obično se sastoji od panela sa UI kontrolama, poput gumbova, klizača, padajućih izbornika i drugo. U ovoj igri korisničko sučelje sastoji se od dva "platna" (eng. canvas) – jedan za meni, a drugi za glavni UI.

Meni će sadržavati panel za pauziranu igru, panel koji se prikaže kada je igra gotova i panel za odabir brzine izvođenja igre. Glavni UI (Slika 4.7.) sastoji se od panela za prikaz kula koje se mogu sagraditi, panela sa rezultatom, tooltipova i panela odabrane kule.



Slika 4.7. Korisničko sučelje (UI)

3D fizika

Da bi se objekti u igri ponašali onako kako igrač očekuje u svakoj situaciji, te dočarala realističnost, u igri je poželjno, pa i nužno uvesti simulirane zakone fizike.

Collider-i su komponente koje omogućuju objektima da reagiraju u doticaju sa ostalim objektima koji također imaju collider komponente. Ovaj mehanizam se još naziva detekcija kolizije (eng. *collision detection*). Collideri su također nužni da bi se skriptom pronašli svi objekti u nekom području, što će u ovoj igri biti neophodno, npr. kada kula traži sve tenkove u blizini ili za spriječavanje gradnje kule preko druge kule.

Uz collider-e, postoji još i komponenta za kruta tijela (eng. *rigid body*) koja objektima daje masu, odnosno mogu padati pod utjecajem gravitacije, te mu simuliraju tromost u igri i druge fizikalne karakteristike. Objekti sa većom masom će reagirati slabije kada dođu u kontakt sa

objektima manje mase. Poželjno je da svi objekti koji se kreću unutar igre imaju ovu komponentu. Gravitacija i druge globalne fizikalne konstante se mogu podesiti unutar postavki fizike u Unity Editoru. Osim gravitacije, moguće je na objekte djelovati silama koje se generiraju skriptom.

Raycasting je proces slanja "zrake" iz neke točke prema definiranom smjeru kako bi se otkrilo da li postoji neki collider na putu zrake. Ovo je npr. vrlo korisno pri programiranju umjetne inteligencije, kada je potrebno implementirati mehanizam po kojemu će neprijatelj detektirati objekte ispred sebe.

4.2 Skripte

Sama scena ne predstavlja igru (eng. *gameplay*). Ona samo predstavlja prostor ili svijet u kojemu će se igra odvijati. Da bi igra dobila "život", odnosno da bi se u njoj nešto događalo, te da bi igrač imao nekakvu interakciju sa njom, potrebno je skriptirati ponašanje objekata u igri. Dakle, skripte predstavljaju komponente ponašanja objekata. Skripte se u Unity-u mogu pisati u C#-u ili Javascriptu. U ovoj igri skripte su pisane u C# programskom jeziku, a kao IDE korišten je Microsoft Visual Studio Ultimate 2013.

Kreiranjem nove skripte dobiva se klasa jednakog naziva kao i skripta (iako je ime klase moguće promijeniti), koja ima dvije osnovne metode:

- void Start() – poziva se kada je objekt instanciran u sceni
- void Update() – poziva se periodički jednom u svakom okviru (eng. *frame*)

Da bi se postigla jednaka brzina odziva igre na svakom uređaju neovisno o FPS-u, nužno je koristiti Time.deltaTime, odnosno vrijeme koje je bilo potrebno da se prikaže zadnji okvir na zaslonu. Primjerice, ukoliko je neki objekt potrebno pomicati 10 metara u sekundi umjesto 10 metara po okviru, taj broj je potrebno pomnožiti sa Time.deltaTime:

```
using UnityEngine;
using System.Collections;

public class ExampleClass : MonoBehaviour
{
    void Update()
    {
        float translation = Time.deltaTime * 10;
        transform.Translate(0, 0, translation);
    }
}
```

Prije nego se započne skriptiranje umjetne inteligencije, odnosno ponašanja tenkova i kula, poželjno je prvo programirati gameplay za igrača, počevši od onoga osnovnog – kamere.

RTS (eng. *Real Time Strategy*) kamera se pomiče kada igrač pomakne kursor miša do ruba zaslona. Uz to, poželjno je da kamera ima i "zoom" na srednju tipku miša.

Na kameru dodajemo novu skript komponentu, te u Update() metodi pišemo sljedeće:

```
void Update ()
{
       // Postavi referentnu translaciju na nulu
       var translation = Vector3.zero;

       //Zumiranje
       var zoomDelta = Input.GetAxis("Mouse ScrollWheel") * ZoomSpeed *
                   Time.unscaledDeltaTime; // Unscaled je neovisan o brzini izvođenja igre

       if (zoomDelta != 0) translation -= Vector3.up * ZoomSpeed * zoomDelta;

       // Pomicanje kamere pomoću tipkovnice
       translation += new Vector3(Input.GetAxis("Horizontal"), 0,
                         Input.GetAxis("Vertical"));

       // Pomiči kameru ako je kursor došao na rub zaslona
       if (Input.mousePosition.x < ScrollArea)
            translation += Vector3.right * -ScrollSpeed * Time.unscaledDeltaTime;

       if (Input.mousePosition.x >= Screen.width - ScrollArea)
            translation += Vector3.right * ScrollSpeed * Time.unscaledDeltaTime;

       if (Input.mousePosition.y < ScrollArea)
            translation += Vector3.forward * -ScrollSpeed * Time.unscaledDeltaTime;


       if (Input.mousePosition.y > Screen.height - ScrollArea)
            translation += Vector3.forward * ScrollSpeed * Time.unscaledDeltaTime;


       // Zadrži kameru unutar granica mape
       var desiredPosition = transform.position + translation;
       if (desiredPosition.x < -LevelAreaX+5 || LevelAreaX+5 < desiredPosition.x)
            translation.x = 0;

       if (desiredPosition.y < ZoomMin || desiredPosition.y > ZoomMax)
            translation.y = 0;

       if (desiredPosition.z < -LevelAreaY-20 || LevelAreaY < desiredPosition.z)
            translation.z = 0;

       // Konačno pomakni kameru zbrajanjem njene transformacije sa novom
       transform.position += translation;
}
```

Nakon toga se piše skripta za praćenje rezultata. Ova skripta se dodaje na prazan objekt i nema Update() metodu jer nema potrebe da se izvršava svaki okvir, nego se periodički poziva metoda updateScore() svake 2 desetinke korištenjem nativne funkcije InvokeRepeating().

```
public int lives = 20;
public int level = 1;
public int money = 100;
public int killCount = 0;

// Tekst prikazan na korisničkom sučelju
public Text killCountText;
public Text moneyText;
public Text livesText;
public Text cooldownText;
public Text levelText;

// Tekst prikazan na panelu kada je igra završena
public Text gameOverScore;
public GameObject gameOverPanel;

public void LoseLife(int l = 1) // Poziva se svaki puta kada tenk dođe na kraj mape
{
	lives -= l;
	if (lives <= 0) GameOver();
}

public void GameOver()
{
	Time.timeScale = 0;
	gameOverScore.text = "Kill count: <i>" + killCount.ToString() + "</i>"; ;
	gameOverPanel.SetActive(true);
}


private void updateScore()
{
	killCountText.text = "Kill count: <i>" + killCount.ToString() + "</i>";
	moneyText.text = "$" + money.ToString();
	livesText.text = "Lives: " + lives.ToString();
}

// Inicijalizacija
void Start()
{
	killCountText.text = "Kill count: <i>" + killCount.ToString() + "</i>";
	moneyText.text = "Money: $" + money.ToString();
	livesText.text = "Lives: " + lives.ToString();
	levelText.text = "LEVEL " + level.ToString();
	InvokeRepeating("updateScore", 0f, 0.2f);
}
```

Slika 4.8. UI panel koji prikazuje rezultat, razinu i dr.

Posljednja skripta koja se odnosi na igrača je skripta za gradnju ili odabir kule.

```
void FixedUpdate()
    {
        iconTimerRemaining -= Time.deltaTime;
        if (iconTimerRemaining <= 0)
        {
            iconTimerRemaining = iconTimer;
            if (selectedObject != null)
                updateSelectionValues(selectedObject.GetComponent<Tower>());
        }

        if (Input.GetMouseButtonDown(0))
        {
            // Gradnja nove kule
            if (buildingTower != null && getBuildAvailability(Input.mousePosition))
            {
                int cost = buildingTower.GetComponent<Tower>().cost;
                scoreManager.money -= cost;
                Tower tower = buildingTower.GetComponent<Tower>();
                tower.enabled = true;
                tower.enemySpawners = enemySpawners;
                towers.Add(tower);
                SetAllCollidersStatus(true);
                colorChildren(buildingTower, originalColor);
                removeCircle(buildingTower);
                buildingTower = null;
                Cursor.visible = true;
            }
            // Selektiranje postojeće kule
            else if (Tower.hoveredTower != null && selectedObject != Tower.hoveredTower)
            {
                clearSelection();
                selectedObject = Tower.hoveredTower;
                selection.selectionPanel.SetActive(true);
```

```
                Tower tower = selectedObject.GetComponent<Tower>();
                updateSelectionValues(tower);
                colorChildren(selectedObject, Color.blue);
                CirleDraw circle = selectedObject.AddComponent<CirleDraw>();
                circle.radius = tower.range;
             }

        }
        if (Input.GetMouseButtonDown(1)) {
            if (buildingTower != null) clearBuildingTower();
            else if (selectedObject != null) clearSelection();
        }

        if (buildingTower != null) {
            Ray ray = mainCamera.ScreenPointToRay(Input.mousePosition);
            RaycastHit hit;
            if (terrain.GetComponent<Collider>().Raycast(ray, out hit, Mathf.Infinity)) {
                buildingTower.transform.position = hit.point;
                // Mijenjanje boje ovisno o mogućnosti gradnje
                if (!getBuildAvailability(hit.point)) colorChildren(buildingTower,
                                                        Color.red);
                else colorChildren(buildingTower, Color.green);
            }
        }
}
public void SelectTowerType(GameObject prefab) // Odabir nove kule za gradnju
{
       Ray ray = mainCamera.ScreenPointToRay(Input.mousePosition);
       RaycastHit hit;
       if (terrain.GetComponent<Collider>().Raycast(ray, out hit, Mathf.Infinity)) {
             buildingTower = (GameObject)Instantiate(prefab, hit.point,
prefab.transform.rotation);

              Tower tower = buildingTower.GetComponent<Tower>();
              SetAllCollidersStatus(false); // Isključi collider za kreirajuću kulu

              originalColor =
buildingTower.transform.GetChild(0).GetComponent<Renderer>().material.color;

            CirleDraw circle = buildingTower.AddComponent<CirleDraw>(); // Isključi
            circle.radius = tower.range;

            if (scoreManager.money >= tower.cost) tower.enabled = false;
            else clearBuildingTower();
        }
    }
    public void upgradeTower() { // Nadogradnja kule
        if (selectedObject != null) {
            Tower tower = selectedObject.GetComponent<Tower>();
            if (tower != null && scoreManager.money >= (int)tower.upgradeCost) {
                scoreManager.money -= (int)tower.upgradeCost;
                tower.upgrade();
                updateSelectionValues(tower);
            }
        }
    }
```

Slika 4.9. Kula koju nije moguće graditi prikazana je crveno

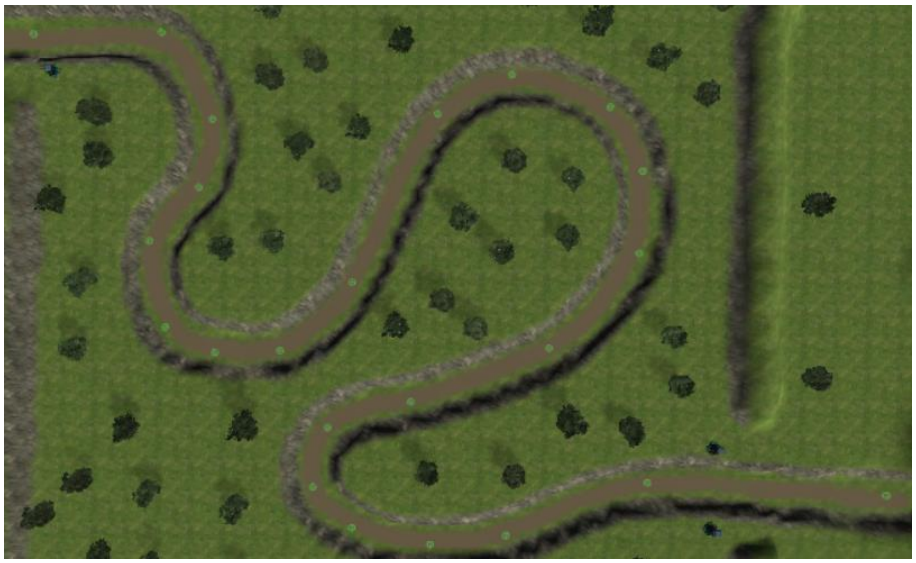
Slika 4.10. Kula koju je moguće graditi prikazana je zeleno

Slika 4.11. Klikom na pojedinu kulu prikazuje se njen domet, kao i njeni atributi

Skripta za kretanje tenkova je vrlo jednostavna i jedino je zadužena za pomicanje tenka od točke do točke. Sve točke se spremaju u listu i kada tenk dođe do prve točke, iz liste se dohvaća sljedeća te se pomicanje nastavlja prema novoj točki sve dok se ne dosegne kraj, prilikom čega se objekt tenka uništava, a igrač gubi život. Efekt health bar-a, odnosno prikaza preostalog života tenka se postiže korištenjem 2 UI text objekta koji je dodan na objekt tenka, pri čemu jedan UI text prikazuje preostali život (zeleno), a drugi oduzeti život (crveno).

```
void GetNextPathNode() {
	if (pathNodeIndex < mainPath.getNodeCount()) {
		targetPathNode = mainPath.getPathNode(pathNodeIndex);
		pathNodeIndex++;
	} else {
		targetPathNode = null;
		ReachedGoal();
	}
}

void Update()
{
	if (targetPathNode == null)
	{
		GetNextPathNode();
		if (targetPathNode == null)
		{
		    // Nema vise novih točaka
		    return;
		}
	}

	Vector3 dir = targetPathNode.position - transform.localPosition;

	float distThisFrame = speed * Time.deltaTime;

	if (dir.magnitude <= distThisFrame)
	{
		// Tenk je došao do točke
		targetPathNode = null;
	}
	else
	{
	    // Pomakni se prema točki
	    transform.Translate(dir.normalized * distThisFrame, Space.World);
	    Quaternion targetRotation = Quaternion.LookRotation(dir);
	    // Postizanje "glatke" rotacije
	    transform.rotation = Quaternion.Lerp(this.transform.rotation, targetRotation,
	                         Time.deltaTime * 5);
	}
}
void ReachedGoal() {
	scoreManager.LoseLife();
	Destroy(gameObject);
}
```

```
// Ova metoda se poziva kada god metak pogodi tenk
public void TakeDamage(float damage) {
       health -= damage;
       // Udio trenutnog healtha u maksimalnom
       int greenCount = (int)(health / maxHealth * healthValues);
       int redCount = healthValues - greenCount;
       string greenText = "";
       string redText = "";

       if (health > 0 && greenCount == 0) {
            greenCount = 1;
            redCount = healthValues - greenCount;
       }
       for (int i = 0; i < greenCount; i++) {
            greenText += "|";
       }
       for (int i = 0; i < redCount; i++) {
            redText += "|";
       }

       remainingHealthText.text = greenText;
       missingHealthText.text = redText;

       if (health <= 0) Die();
}
public void Die() {
       scoreManager.money += moneyValue + scoreManager.level;
       scoreManager.killCount++;
       GameObject explosionGO = (GameObject)Instantiate(explosionPrefab, transform.position,
                                transform.rotation);
       Destroy(explosionGO, 2f);
       Destroy(gameObject);
}
```



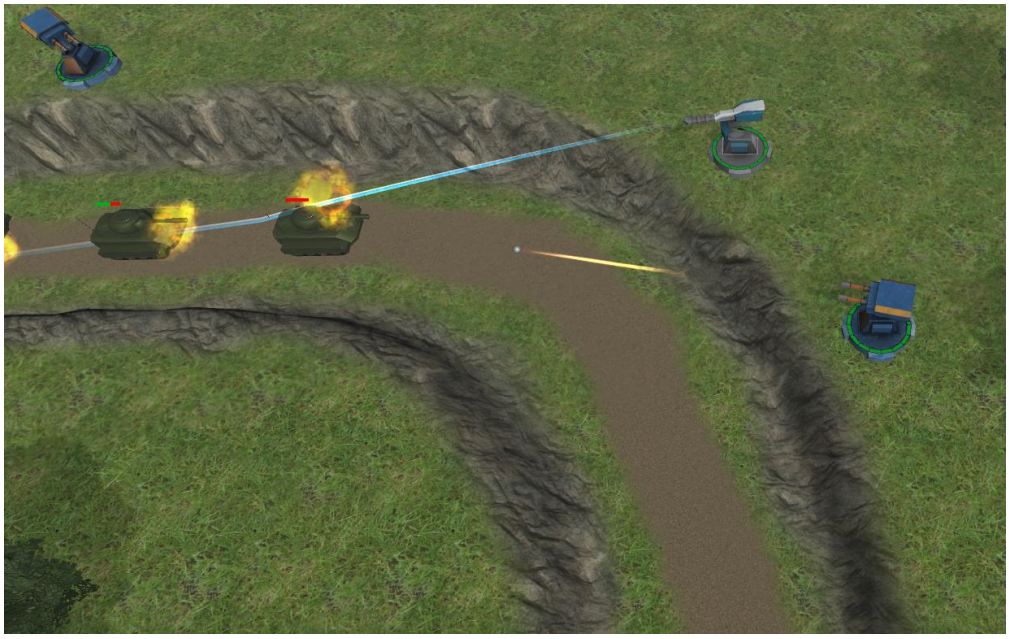
Slika 4.12. Putanja po kojoj se kreću tenkovi je određena praznim objektima (zelene točke)

Osim metoda, u klasi naravno još postoje atributi za život, maksimalni život, nanesenu štetu, brzinu kretanja, domet i dr. koje nema potrebe navoditi. Vrijednosti ovih atributa se mogu postaviti i u Inspector-u objekta. Skripta koja je zadužena za ponašanje kula je opisana sljedećim kodom:

```
void Start()
{
       turretTransform = transform.Find("Head"); // Head predstavlja glavu topa kule

       gunShot = GetComponent<AudioSource>(); // Zvuk pucanja

       baseDamage = damage;
       baseCost = cost;
       upgradeCost = baseCost * (1 + upgradeCostFactor * upgradeLevel);
}

private void findCloseEnemies(float distance)
{
       reactionTime += Time.deltaTime; // S ovime se smanjuje nepotreban broj kalkulacija
       if (reactionTime >= reactionCooldown) {
            // Traženje svih tenkova u blizini preko njihovih Collider komponenti
            Collider[] hitColliders = Physics.OverlapSphere(transform.position,
                                          acquisitionRange);

            foreach (Collider collider in hitColliders)
            {
                // Potrebno je razlikovati collider tenka od npr. collider-a stabla
                if (collider.name == "main") {
                    Enemy enemy = collider.GetComponentInParent<Enemy>();
                    if (enemy != null) {
                        float distanceToTarget = Vector3.Distance(this.transform.position,
                                                    enemy.transform.position);
                        if (distanceToTarget < distance) {
                            target = enemy;
                            // Kada je tenk pronađen, nema potrebe za daljnjom potragom
                            break;
                        }
                    }
                }
            }
            reactionTime = 0;
        }
    }
void Update()
{
       if (target == null)
            findCloseEnemies(acquisitionRange);

       else
       {
            fireCooldownLeft -= Time.deltaTime;
            float distanceToTarget = Vector3.Distance(this.transform.position,
                                        target.transform.position);

            if (distanceToTarget > acquisitionRange)
                findCloseEnemies(acquisitionRange);
```

```
        else {
            Vector3 dir = target.transform.position - this.transform.position;

            // Animacija glave topa (okretanje prema tenku)
            Quaternion lookRot = Quaternion.LookRotation(dir);
                    turretTransform.rotation = Quaternion.Lerp(turretTransform.rotation,
            Quaternion.Euler(lookRot.eulerAngles.x, lookRot.eulerAngles.y, 0),
                    Time.deltaTime * 5);

            if (distanceToTarget <= range && fireCooldownLeft <= 0 && dir.magnitude <= range)
            {
               fireCooldownLeft = fireCooldown;
               ShootAt(target);

            } else findCloseEnemies(distanceToTarget);
        }
    }
}

public int getMinDamage(float damageValue) {
        return (int)(damageValue / damageDeviation);
}

public int getMaxDamage(float damageValue) {
        return (int)(damageValue * damageDeviation);
}

public int getRandomDamage() {
        // Za dodatnu zanimljivost, izračunava se slučajna vrijednost damage-a oko osnovnog
        return random.Next(getMinDamage(damage), getMaxDamage(damage)+1);
}

void ShootAt(Enemy e) {
        gunShot.Play();
        Vector3 newPosition;

        // Lokacija na kojoj se instancira metak (na vrhu glave topa)
        newPosition = turretTransform.position + new Vector3(0, 2, 0) +
                        turretTransform.forward;

        GameObject bulletGO = (GameObject)Instantiate(bulletPrefab, newPosition,
                                turretTransform.rotation);

        Bullet b = bulletGO.GetComponent<Bullet>();
        b.target = e.transform;
        b.damage = getRandomDamage();
        b.radius = radius;
        b.speed = bulletSpeed;
        b.lifespan = (acquisitionRange / bulletSpeed) * 2;
        b.start = true;
}
```

Skriptu kretanja metka nema potrebe prikazivati, jer koristi sličan princip kretanja kao i tenk.



Slika 4.13. Po želji se dodaju specijalni efekti kada metak pogodi tenk

Zadnja važna skripta je skripta koja upravlja pojavljivanjem tenkova (eng. *spawn*), kao i njihovim pojačavanjem svaki novi level. Kreiraju se prazni objekti zaduženi za spawn tenkova, a pridružuju se glavnom objektu koji će kontrolirati uključivanje spawnera. Ovaj kontrolni objekt također određuje i broj tenkova, tip, te vrijeme za koje će se tenk pojaviti za svaki od spawner-a.

```
public class WaveComponent{
    public GameObject enemyPrefab;
    public int num;
    public float cd;
    public int spawned = 0;
}

public List<WaveComponent> waveComps = new List<WaveComponent>();

void Update () {
if (active)
{
       spawnCDremaining -= Time.deltaTime;
       if (spawnCDremaining <= 0)
       {
              WaveComponent wc = waveComps[wcIndex];
              GameObject enemyGO = (GameObject)Instantiate(wc.enemyPrefab,
                                   transform.position, transform.rotation);
              Enemy enemy = enemyGO.GetComponent<Enemy>();
              enemy.mainPath = mainPath;
              enemy.scoreManager = scoreManager;

              float baseHealth = enemy.health;
              // Određivanje health-a tanka za trenutni level
              float healthIncrease = baseHealth * difficultyFactor;
```

```
            for (int i = 0; i < scoreManager.level-1; i++)
                  enemy.health += baseHealth + healthIncrease * i;

            enemy.maxHealth = enemy.health;
            wc.spawned++;

            spawnCDremaining = wc.cd;

            if (wc.spawned >= wc.num) {
                 wcIndex++;
                 spawnCDremaining += spawnCD;

                 if (wcIndex >= waveComps.Count) {
                     foreach (WaveComponent waveComp in waveComps)
                          waveComp.spawned = 0;
                     wcIndex = 0;
                     active = false;
                 }
            }
       }
}
}
```

Glavni kontroler za spawn:

```
private void generateWaveComponents()
{
       int count = (int)(scoreManager.level * 0.25f) + 3;
       int spawnIndex = 3;
       foreach (EnemySpawner spawner in spawners)
       {
            spawner.waveComps.Clear();

            for (int i = 0; i < count; i++)
            {
                EnemySpawner.WaveComponent wc = new EnemySpawner.WaveComponent();
                int index = 0;
                // Slučajni odabir za tip tenka
                if (scoreManager.level > 1 && random.Next(3) == 1)
                {
                    index = 1;
                    wc.num = (int)((scoreManager.level + spawnIndex) * 0.25f);
                    wc.cd = heavyTankOffset; // Vrijeme između svakog spawna
                }
                else
                {
                    wc.num = scoreManager.level + spawnIndex;
                    wc.cd = basicTankOffset;
                }

                wc.enemyPrefab = enemyTypePrefab[index];

                spawner.waveComps.Add(wc);
            }
            spawnIndex++;
        }
}
```

```
void Start () {
       mainPath = GameObject.Find("Path").GetComponent<MainPath>();
       scoreManager = GameObject.Find("_SCRIPTS_").GetComponent<ScoreManager>();
       foreach (Transform child in transform) {
           EnemySpawner spawner = child.GetComponent<EnemySpawner>();

           if (spawner.gameObject.activeSelf) {
                spawner.mainPath = mainPath;
                spawner.scoreManager = scoreManager;
                spawners.Add(spawner);
           }
       }
       generateWaveComponents();
       int cdRemaining = (int)waveCDremaining - 1;
       TimeSpan time = TimeSpan.FromSeconds(cdRemaining);
       DateTime dateTime = DateTime.Today.Add(time);
       scoreManager.cooldownText.text = "Next wave: " + dateTime.ToString("mm:ss");
}

void Update () {
       waveCDremaining -= Time.deltaTime;
       timer += Time.deltaTime;
       if (timer >= 0.2f) {
            int cdRemaining = (int)waveCDremaining;
            TimeSpan time = TimeSpan.FromSeconds(cdRemaining);
            DateTime dateTime = DateTime.Today.Add(time);
            scoreManager.cooldownText.text = "Next wave: " + dateTime.ToString("mm:ss");
            timer = 0;
       }
       if (waveCDremaining <= 0) {
           if (init) {
                spawners[spawnerIndex].start(0);
                waveCDremaining = spawners[spawnerIndex].getWaveCooldown() + waveCD;
                init = false;
                if (spawners.Count == 1) waveCDremaining += scoreManager.levelCooldown;
           } else {
               spawnerIndex++;

               if (spawnerIndex < spawners.Count) {
                   waveCDremaining = spawners[spawnerIndex].getWaveCooldown() + waveCD;
                   spawners[spawnerIndex].start(0);

                   if (spawnerIndex == spawners.Count - 1) waveCDremaining +=
                        scoreManager.levelCooldown;
               } else {
                    spawnerIndex = 0;
                    scoreManager.level++;
                    generateWaveComponents();
                    scoreManager.levelText.text = "LEVEL " + scoreManager.level.ToString();
                    scoreManager.money += scoreManager.level * 10;
                    waveCDremaining = spawners[spawnerIndex].getWaveCooldown() + waveCD;
                    spawners[spawnerIndex].start(0);
               }
           }
       }
}
```

5 ZAKLJUČAK

Opći je stereotip da su računalne igre namijenjene isključivo mlađoj populaciji, međutim sve više dobivaju popularnosti i među onom starijom. Postoje mnogi mitovi da neke igre u kojima dominira nasilje potiče igrača na nasilje i u stvarnosti, te da igre negativno utječu na čovjekovu socijalnost i vezanost za stvarni svijet. No ne postoje baš nikakvi dokazi niti činjenice kojima se ove tvrdnje mogu i potkrijepiti[3]. Dapače, mnoge današnje popularne multiplayer igre potiču međusobnu suradnju i komunikaciju među igračima. Neke igre čak razvijaju njihove intelektualne i motoričke sposobnosti.

Osim igranja, posebno veliku čar ima i programiranje računalnih igara. Ono što ispunjava čovjeka je realizacija njegovih ideja u stvarnost. Tako je bilo od pamtivijeka, no uvijek su postojale granice jer je ideja mogla biti realizirana samo u opipljivom i materijalnom obliku. No, kada se prvi puta pojavio koncept virtualne stvarnosti, granice ljudske mašte i kreativnosti su nestale. Međutim, iz primjera igre opisane u ovom radu vidljivo je da je programiranje računalne igre vrlo mukotrpan posao, pogotovo kada se ona radi od nule. Stoga se one programiraju na već gotovim platformama koje se nazivaju game engine-ima, a vrlo složene igre se razvijaju od strane više desetaka ljudi u velikim kompanijama. Unity je jedan od najpristupačnijih game engine-a za razvoj 2D i 3D računalnih igara za više platformi, te je stoga ujedno i jedan od najpopularnijih game engine-a među indie (eng. *independent*, neovisan) populacijom developera, te može poslužiti kao velika odskočna daska za one koji traže karijeru u profesionalnom razvoju video igara, ali i svakako proširiti znanje kako iz objektno orijentiranog programiranja, tako i u primjeni 3D računalne grafike i matematike.

Računalne igre su od prvih pokusnih programa dostupnih samo onima koju su ih i kodirali, te namijenjenih uglavnom za simulacije i istraživanja danas prerasle u neophodan dio ljudske zabave i razonode. One mu omogućuju da, barem nakratko, napusti svoju repetitivnu svakodnevicu i uđe u svijet nepredvidivosti, izazova i mašte.

[3] https://www.youtube.com/watch?v=bIdfhczSSzc

LITERATURA

[1] http://read.pudn.com/downloads165/ebook/753982/3D_Computer_Graphics.pdf (Kolovoz 2016)

[2] http://www.fundza.com/rman_booklet/rman_booklet.pdf (Kolovoz 2016)

[3] http://www.ijeit.com/Vol%204/Issue%2010/IJEIT1412201504_15.pdf (Kolovoz 2016)

[4] http://www.gamecareerguide.com/features/529/what_is_a_game_.php (Rujan 2016)

[5] https://en.wikipedia.org/wiki/Unity_(game_engine) (Rujan 2016)

[6] https://unity3d.com/learn/tutorials (Rujan 2016)

[7] https://www.youtube.com/watch?v=fRED_-LvJKQ (Kolovoz 2016)

[8] https://www.youtube.com/watch?v=b7DZo4jA3Jo (Kolovoz 2016)

POPIS SLIKA

ŽIVOTOPIS

Dino Čatalinac rođen je 19.05.1992. u Đakovu, Republika Hrvatska. Od rujna 2007. do lipnja 2011. pohađao je srednju strukovnu školu Braće Radića u Đakovu, smjera Računalni tehničar za strojarstvo. U listopadu 2011. upisuje sveučilišni preddiplomski studij na Elektrotehničkom fakultetu u Osijeku, smjera Računarstvo, a u listopadu 2014. nastavlja studij na sveučilišnom diplomskom studiju Elektrotehničkog fakulteta, smjera Procesno računarstvo.

U listopadu 2015. prvi puta se zapošljava na rad preko studenstkog ugovora u Inovativo Marketing agenciji u Osijeku kao developer web aplikacija. U travnju 2016. zapošljava se u UHP Digital d.o.o. u Osijeku kao php i frontend developer, gdje radi i danas.

U međuvremenu Elektrotehnički fakultet u Osijeku mijenja naziv u Fakultet elektrotehnike, računarstva i informacijskih tehnologija, na kojemu Dino Čatalinac planira diplomirati u listopadu 2016.

Kroz studij, samostalni rad, te rad u struci stekao je praktična znanja i vještine programiranja profesionalnih aplikacija prvenstveno u web tehnologijama kao što su HTML, CSS, Javascript, PHP i SQL, uz korištenje Zend i Laravel PHP framework-a, te AngularJS kao Javascript framework-a.