Konstrukcija 2D pokretača igre

Orlić, Matej

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Matej Orlić

KONSTRUKCIJA 2D POKRETAČA IGRE

Diplomski rad

Voditelj rada:
Prof. dr. sc. Mladen Jurak

Zagreb, rujan 2017.

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. _____

Hvala obitelji, kolegama i zaručnici na svim savjetima, strpljenju i podršci.

Sadržaj

Uvod

Gotovo da nema osobe koja nikada nije igrala neku video igru. Od samih početaka, kada su bile izrađene potpuno pomoću hardvera, video igre prate razvoj računarstva i dio su njega. Mnoge komponente današnjeg računala razvijene su i poboljšane zahvaljujući industriji igara, a razvoj samih računala omogućio je gotovo fotorealistične igre sa fascinantnim mogućnostima.

Video igre dolaze u svim mogućim žanrovima, a igrive su na širokom spektru tehnologija - od računala i pametnih telefona do igračih konzola i specijalnih uređaja za simulaciju virtualne stvarnost. Današnja industrija video igara zapošljava ogroman broj ljudi, a u izradi nekih većih igara sudjelovalo je i do 500 ljudi. Zanimljiv pokazatelj je i zarada industrije igara, koja premašuje zaradu filmske i glazbene industrije zajedno.

U ovom diplomskom radu prvo ćemo reći nešto o razvoju industrije igara, te stanju u Hrvatskoj i u svijetu, a osim toga pogledat ćemo i tipičnu strukturu razvojnog tima. Nakon toga objasnit ćemo pojam pokretača igre, njegovu ulogu i arhitekturu, te ćemo navesti neke alate koji se koriste za izradu igara i poznate pokretače igre. Nakon što se upoznamo sa osnovama, obradit ćemo dva najveća sustava svakog pokretača igre, a to su sustav za iscrtavanje i sustav za fiziku. Sustav za iscrtavanje zadužen je za iscrtavanje igre na ekran računala, dok je sustav za fiziku zadužen za otkrivanje sudara između objekata u igru, te simulaciju njihove dinamike. Nešto manje reći ćemo i o ostalim dijelovima pokretača igre, kao što su sustav za upravljanje memorijom, sustav za zvuk i skriptni jezik.

Na kraju ćemo pogledati primjer implementacije jednostavnog 2D pokretača igre u programskom jeziku C++. Za implementaciju iscrtavanja i grafike koristit ćemo biblioteku SFML (Simple and Fast Multimedia Library), a za implementaciju fizike koristit ćemo biblioteku Box2D. Osim toga, lakši i brži razvoj igara omogućit ćemo implementacijom skriptnog jezika Lua.

Poglavlje 1

Industrija video igara

Industrija video igara je sektor koji se bavi razvojem, marketingom i monetizacijom video igara. Ona obuhvaća desetke različitih poslovnih disciplina, i zapošljava desetke tisuća ljudi širom svijeta.

Današnja moderna računala mnoge svoje inovacije i poboljšanja duguju upravo industriji video igara. Video igre su potaknule razvoj sve boljih zvučnih kartica, grafičkih kartica, procesora i softvera za ubrzanje 3D grafike. Naime, zvučne kartice prvotno su razvijene za poboljšanje kvalitete zvuka u video igrama. Grafičke kartice razvijene su za omogućavanje prikaza većeg broja različitih boja na ekranu, a danas se većim dijelom koriste upravo za iscrtavanje (engl. *Rendering*) video igara. Razvoj CD-a i DVD-a također je potaknut video igrama koje su se u svojim počecima prodavale isključivo na optičkim diskovima [11].

1.1 Povijest i razvoj

Razvoj prvih video igara omogućio je izum katodne cijevi, odnosno izum prvih televizora. Jedna od prvih video igara napravljena je 1947. godine, a patent za igru bio je prijavljen kao ”uređaj za zabavu katodnom cijevi”. Uređaj je imao katodnu cijev priključenu na osciloskopski ekran, a cilj je bio pogoditi metu na ekranu.

1950. godine u Kanadi je napravljen uređaj ”Bertie the Brain” koji je koristeći umjetnu inteligenciju igrao igru križić kružić protiv ljudskog igrača.

Slika 1.1: Bertie the Brain. Izvor: [13]

1972. godine izašla je prva komercijalno uspješna video igra "Pong" koju je napravio Atari. Izašla je u obliku arkadnog kabineta, a prodana je u više od 19.000 primjeraka.

U sljedećih par godina došlo je do pojave prvih igračih video konzola (Magnavox Odyssey), ali je tržište bilo toliko preplavljeno klonovima igre Pong da je došlo do stagnacije razvoja video igara. Tek 1978. godine izlazi igra Space Invaders, također u obliku arkadnog kabineta, a prodana je u preko 360.000 primjeraka. U sljedeće četiri godine ostvarila je prihod od dvije milijarde američnih dolara.
Ubrzo izlazi druga generacija igračih konzola čiji je predstavnik konzola Atari 2600, prodana u preko 30 milijuna primjeraka (1980. godina).

1980-ih godina industrija arkadnih kabineta doživljava zlatno doba, a 1982. godine zarada od prodaje arkadnih kabineta premašila je zaradu glazbene i filmske industrije zajedno. Najpopularnija igra ovog razdoblja bio je "Pac-Man", a od igračih konzola istaknuo se Nintendo.

1990-tih godina dolazi do razvoja CD-a i sve šire distribucije igara preko CD-a. Tijekom ovog razdoblja prodaja igara za igrače konzole i stolna računala dostigla je prodaju arkadnih kabineta. Razvojem tehnologije u ovom razdoblju omogućena je tranzicija iz 2D igara u 3D igre. Predstavnik igračih konzola ovog razdoblja bio je PlayStation.

2000-tih godina dolazi do pojave novih tehnologija kao što su pametni telefoni i virtualna stvarnost. Sve više se razvija industrija mobilnih igara, a posebno popularne postaju igre na mreži.

Danas su najpopularnije upravo mobilne koje su po zaradi prestigle igre za stolna

Slika 1.2: Pong. Izvor: [13]

računala i konzole. Sve je veći broj manjih, tzv. "indie" razvojnih timova koji uz pomoć današnjih alata sve lakše razvijaju igre i pronalaze svoje mjesto na tržištu. Više o povijesti i razvoju video igara može se vidjeti na poveznici [13].

1.2 Industrija video igara u svijetu

Procjenjuje se da je trenutna globalna vrijednost industrije video igara više od 100 milijardi američkih dolara. To je više nego vrijednost glazbene i filmske industrije zajedno. Trenutni rekorder u zaradi je igra "Call of Duty: Black Ops" koja je u prvih pet dana od izlaska zaradila preko 650 milijuna američkih dolara.

Današnje tržište, odnosno broj ljudi koji igraju video igre procjenjuje se na 2.5 milijarde ljudi širom svijeta. Prosječna starost igrača je 35 godina, a čak 72 posto igrača je starije od 18 godina. Iako je općenito mišljenje da igre igraju samo muškarci, i to mlađa

populacija, zanimljivo je da su gotovo trećina igrača žene. Isto tako, mlađih od 18 godina je svega 29 posto igrača, dakle većinu čine punoljetni igrači.

Samo u SAD-u, ukupan broj zaposlenika u industriji video igara procjenjuje se na 220.000. Vrijeme potrebno za razvijanje video igre je najčešće između jedne i tri godine. Za razvoj jednostavne igre dovoljna je jedna osoba, dok je za razvoj velikih igara potrebno čak i do 500 i više ljudi [1].

1.3 Industrija video igara u Hrvatskoj

Industrija video igara u Hrvatskoj ima rast od 50% godišnje kroz zadnjih pet godina, što ju čini jednom od najbrže rastućih industrija u Hrvatskoj. U 2014. godini prihodi su iznosili oko 50 milijuna kuna, a zaposlenih u ovoj industriji bilo je 250 osoba. Još toliko osoba bavilo se razvojem igara u slobodno vrijeme, bez stalnog zaposlenja.

1.4 Struktura razvojnog tima

Timovi koji razvijaju video igre obično se sastoje od sljedećih pet disciplina: inžinjeri (programeri), umjetnici, dizajneri igre, producenti i ostalo osoblje (marketing, tehnička podrška, administracija, itd.). Sada ćemo analizirati svaku navedenu disciplinu i njezine poddiscipline.

Programeri

Programeri su zaduženi za dizajn i implementaciju softvera koji će omogućiti da se igra pokrene i da radi ono što se od nje očekuje, ali i dodatnih alata potrebnih za razvoj igre. Zbog toga su programeri često kategorizirani u dvije grupe: programeri softvera u stvarnom vremenu (engl. *Runtime programmers*) i programeri alata. Programeri softvera u stvarnom vremenu rade na pokretaču igre i na samoj igri, dok programeri alata razvijaju alate koji omogućavaju ostatku tima da što efektivnije obavljaju svoj posao.

Iskusniji programeri obično postaju voditelji programerskog tima. Oni i dalje dizajniraju i programiraju, ali također brinu o rasporedu i vremenskim rokovima projekta, te općenito o tehničkom dijelu projekta. Neke tvrtke zapošljavaju jednog ili više tehničkih direktora, koji nadgledaju jedan ili više projekata sa više razina. Oni se brinu o tome da je tim svjestan potencijalnih tehničkih izazova, novih tehnologija i tako dalje. Najviša pozicija koju programer može imati je glavni tehnički direktor (engl. *Chief technical officer* -

CTO), a njegova je uloga slična ulozi tehničkog direktora, samo što dodatno ima i izvršnu ulogu u tvrtci.

Umjetnici

Postoji puno umjetničkih uloga koje sudjeluju u razvoju jedne video igre, a nije rijetkost da je jedan umjetnih zadužen za više uloga.

Konceptni umjetnik (engl. *Concept artist*) je zadužen za stvaranje skica i slika kojima će ostatku tima dočarati finalan izgled igre. Oni svoj rad započinju u vrlo ranim fazama projekta, ali obično su prisutni do samog kraja i pomažu da igra vizualno ide u dobrom smjeru. Ako je igra 2D, tada konceptni umjetnik može također stvarati finalni digitalni sadržaj koji će se koristiti u igri, u obliku slika.

3D modeler stvara trodimenzionalnu geometriju za sve što će se naći u virtualnom svijetu igre. 3D modeleri obično se dijele na modelere koji stvaraju statičan sadržaj kao što su zgrade, mostovi, planine i slično, i na modelere koji stvaraju dinamičan sadržaj kao što su likovi, vozila, oružja i ostali objekti.

Umjetnik tekstura (engl. *Texture artist*) zadužen je za stvaranje dvodimenzionalnih slika (tekstura) koje su zatim projicirane na površinu 3D modela. Na taj način 3D modelima pružaju se detalji i realizam.

Umjetnik osvijetljenja (engl. *Lighting artist*) zadužen je za postavljanje svih izvora svijetlosti u sceni, te podešavanju njihovih boja, intenziteta i smjera kako bi se maksimizirala kvaliteta vizualnog izgleda igre.

Animator unosi pokrete i kretnje objektima i likovima u igri. Animatori u igrama imaju jako sličnu ulogu animatorima animiranih filmova. Ipak, animatori u igrama imaju malo teži zadatak jer moraju naći optimalnu mjeru između kvalitete i težine svake animacije, kako bi se animacija mogla tehnički izvesti u pokretaču igre.

Glasovni glumci zaduženi su za pružanje glasova likovima unutar igre. Kompozitori stvaraju originalnu glazbu koja će se koristiti unutar igre, dok zvučni inženjeri pomažu programerima u stvaranju zvučnih efekata i glazbe unutar igre.

Dizajneri igre

Razvoj igre najčešće počinje od dizajnera igre. Njihova zadaća je dizajnirati interaktivno iskustvo koje će se prenijeti igraču kroz igranje igre. Mnoge igre zapošljavaju i posebnog pisca koji piše priču koja će pratiti samu igru. Iskusniji dizajneri obično dizajniraju igru na makro razini, tj. određuju tijek priče koja prati igru, raspored poglavlja i razina igre, te glavne ciljeve igre. Ostali dizajneri rade na individualnim razinama u igri, stvaraju virtualni svijet tako da postavljaju geometriju i ostale objekte unutar igre, a ponekad u suradnji sa programerima sudjeluju u pisanju skriptnog programskog koda kojima oblikuju logiku

objekata. Svaka igra ima jednu istaknutu osobu, voditelja tima dizajnera koji nadgleda sve aspekte dizajna igre, prati raspored i vremenske rokove, te usklađuje rad svih dizajnera igre.

Producenti

Uloga producenta razlikuje se od tvrtke do tvrtke, dok neke čak niti nemaju producenta kao takvog. U nekim tvrtkama, uloga producenta je da se brine o rasporedu i vremenskim rokovima, ali i da upravlja ljudskim resursima. U drugim tvrtkama, producenti zapravo imaju ulogu glavnog dizajnera igre. U većini tvrtki producent je zapravo posrednih između razvojnog tima i poslovnog dijela tvrtke, koji je zadužen za financije i marketing.

Ostalo osoblje

Timu koji razvija samu igru uvijek je dostupno pomoćno osoblje. To su izvršni tim za upravljanje tvrtkom, odjel za marketing, administrativni odjel i odjel za informacijske tehnologije koji pruža tehničku podršku.

Izdavači

Za marketing i distribuciju igre najčešće je zadužen izdavač, a ne razvojni tim. Izdavači su obično velike tvrtke, kao što su Electronic Arts, THQ, Vivendi, Sony, Nintendo i tako dalje. Razvojni timovi nisu nužno vezani za jednog izdavača, nego ulaze u dogovor sa izdavačem koji im ponudi za njih najbolju ponudu. S druge strane, neki razvojni timovi su u vlasništvu izdavača, koji mogu ili ne moraju direktno upravljati svojim razvojnim timovima.

Poglavlje 2

Pokretač igre

Prije nego što detaljnije opišemo pojam pokretača igre (engl. *Game engine*), prvo ćemo ukratko opisati pojam video igre. Svi bi svojim riječima znali opisati što je računalna igra, a ovdje ćemo ju opisati matematički. Nakon toga objasnit ćemo ulogu pokretača igre, te nabrojati i opisati njegove dijelove. Na kraju ćemo navesti nekoliko danas najpoznatijih pokretača igara, te ih međusobno usporediti.

2.1 Što je računalna igra?

Gledano iz perspektive računalne znanosti, video igre se definiraju kao "mekane" interaktivne računalne simulacije u realnom vremenu bazirane na agentima (engl. *Soft real-time interactive agent-based computer simulations*). Sada ćemo razdvojiti ovu frazu na dijelove kako bi ju lakše shvatili.

U većini video igara neki podskup realnog ili imaginarnog svijeta se matematički modelira, što mu omogućava izvođenje na računalu. Taj model je aproksimacija i pojednostavljenje stvarnosti, čak i ako je to imaginarna stvarnost, zato što je očito nemoguće na računalu prikazati svaki detalj stvarnosti sve do razine atoma. Zbog toga kažemo da je taj matematički model simulacija realnog ili imaginarnog svijeta igre. Glavni alati za svakog razvojnog programera igara su upravo aproksimacija i simplifikacija. Kada se ti alati koriste na pravi način, čak i jako pojednostavljeni modeli ponekad mogu biti gotovo identični stvarnosti koju opisuju.

Kažemo da je simulacija bazirana na agentima kada dolazi do interakcije između više različitih entiteta koje nazivamo agenti. U računalnim igrama ti agenti na primjer mogu biti razna vozila, likovi ili leteće vatrene kugle. Zbog činjenice da se većina igara bazira na agentima, za očekivati je da se igre najčešće implementiraju u objektno orijentiranim

programskim jezicima.

U svim video igrama stanje modela virtualnog svijeta konstantno se mijenja, kao reakcija na događaje u igri i razvoj priče. To znači da je taj model virtualnog svijeta dinamičan, mijenja se tijekom vremena, odnosno možemo reći da su video igre temporalne (vremenske) simulacije. Osim toga, video igra mora cijelo to vrijeme reagirati na nepredvidljive ulazne podatke koji dolaze od ljudskog igrača. Zbog toga su igre i interaktivne temporalne simulacije. I na kraju, video igre donose svoje priče i reagiraju na ulazne podatke dobivene od igrača trenutno, u stvarnom vremenu, što ih čini interaktivnim simulacijama u realnom vremenu.

Koncept vremenskog ograničenja ili vremenskog roka je u srži svakog računalnog sustava koji se odvija u realnom vremenu. Na primjer, u video igrama postoji vremensko ograničenje koje zahtijeva da igra na zaslon iscrtava najmanje 24 sličice u sekundi, kako bi se stvorila iluzija pokreta. Neki drugi dijelovi igre, kao što je simulacija fizike, imaju kraći vremenski rok za svoje ažuriranje (engl. *Update*) i zahtijevaju 120 ažuriranja u sekundi kako bi ostali u stabilnom stanju. Sustav umjetne inteligencije nekog lika u igri možda zahtijeva bar jedno ažuriranje u sekundi, kako bi izgledao ”inteligentno”. Kažemo da je sustav u realnom vremenu ”mekan” ako prekoračenje vremenskog roka nije katastrofalno. Ako broj sličica u sekundi padne ispod 24, ništa katastrofalno se neće dogoditi. Suprotnost mu je ”tvrd” sustav u realnom vremenu, u kojem prekoračenje vremenskog ograničenja može dovesti do ljudskih žrtava. Na primjer, to su sustavi u avionu ili kontrolni sustavi nuklearnih elektrana.

Žanrovi

Sada ćemo navesti nekoliko najpopularnijih žanrova video igara.

Pucačina iz prvog lica (engl. *First-person shooter*) je tip igre u kojoj igrač iz prvog lica upravlja likom, automobilom, brodom, avionom ili drugim vozilima. Glavni predstavnici ovog žanra su igre Quake, Unreal Tournament, Half-Life i Call of Duty.

Igre iz trećeg lica (engl. *Third-person games*) imaju puno sličnosti sa igrama iz prvog lica, ali naglasak je ovdje stavljen na sposobnosti glavnog lika i animacije cijelog lika (koji nije vidljiv u igrama iz prvog lica). Posebno popularan podžanr su platformer igre, čija je glavna mehanika igre skakanje sa platforme na plaformu. Najpoznatije igre ovog žanra su Super mario, Crash Bandicoot, Sonic the Hedgehog i Ratchet and Clank.

Borbene igre (engl. *Fighting games*) su igre u kojima (najčešće) dva igrača upravljaju likovima koji se bore jedan protiv drugoga. Glavni predstavnik ovog žanra je igra Tekken.

Trkaće igre (engl. *Racing games*) su sve igre čiji je primarni zadatak upravljanje automobilom ili drugim vozilom na nekoj vrsti trkače staze. Neke od popularnijih igara ovog žanra su Gran Turismo, Need for Speed i Mario Kart.

Strateške igre su igre u kojima igrač postavlja borbene postrojbe iz svog arsenala unutar velikog igračeg polja, sa ciljem pobjeđivanja protivnika. Neki predstavnici ovog žanra su Warcraft, Age of Empires i Starcraft.

Masivne mrežne online igre (engl. *Massively Multiplayer Online Games*) su igre koje podržavaju velik broj igrača koji istovremeno sudjeluju u velikom virtualnom svijetu. Broj igrača koji su istovremeno u igri može biti od tisuću, pa sve do stotinu tisuća.

Osim ovih, postoje mnogi drugi žanrovi kao što su sportske igre, igre uloga, klasične igre na ploči (šah, igre s kartama), logičke igre, i mnogi drugi.

2.2 Uloga pokretača igre

Pokretač igre je zapravo skup alata koji se koriste za razvoj i pokretanje video igara. On omogućava pokretanje računalne igre na platformama kao što su konzole, mobilni uređaji i osobna računala. Pokretač igre mora biti lako proširiv i u njemu mora biti moguće napraviti više različitih igara, bez većih modifikacija samog pokretača igre.

Postavlja se pitanje koja je njegova uloga, odnosno koje su prednosti pokretača igre u odnosu na pisanje programskog koda za samu igru? Zašto uložiti vrijeme u stvaranje pokretača igre, kada je to vrijeme moglo biti utrošeno u razvoj same igre?

Na samom početku, igre su se razvijale od nule, pisanjem programskog koda koji upravlja igrom. Pojam pokretača igre pojavio se 1990-ih godina, nakon razvoja prvih pucaških igara iz prvog lica (FPS igara). Jedan od primjera je popularna igra Doom koju je napravio razvojni studio id Software. Doom je imao arhitekturu sa jasnom granicom između baznih komponenti kao što su sustavi za grafiku, fiziku i zvuk, i ostalih sustava koji su oblikovali igračevo iskustvo igranja igre (3D modeli, svijet unutar igre). Vrijednost ove podjele postala je izraženija nakon što je razvojni tim počeo licencirati igre i stvarati nove igre mijenjajući samo ”assete” (3D modeli, teksture, glazba...), raspored levela (razina igre?), likove i vozila, dok su promjene na samom pokretaču igre bile minimalne. Mnogi individualci i manji neovisni razvojni timovi mogli su kreirati svoje igre modificirajući postojeće igre. Samim time pokretači igre omogućili su brži razvoj igara, a same

igre bile su stabilnije.

Dakle, glavna uloga pokretača igre je omogućiti razvojnom programeru jednostavan i brz razvoj igre kroz više područja kao što su grafika, fizika, zvuk i umjetna inteligencija. Osim toga, pokretač igre može pružiti apstrakciju platformi, što omogućava da se ista igra pokreće na više platformi, sa vrlo malo izmjena u izvornom programskom kodu.

Bitno je napomenuti da ne postoji pokretač igre u kojem bi se mogla napraviti bilo koja igra koju možemo zamisliti. Na primjer, program koji je dizajniran za iscrtavanje interijera, možda neće biti koristan za iscrtavanje velikih vanjskih prostora. Međutim, sa razvojem grafičkih kartica, bržih algoritama i struktura podataka, ta granica je sve manja, a mogućnosti su sve veće.

2.3 Arhitektura sustava za izvođenje igre

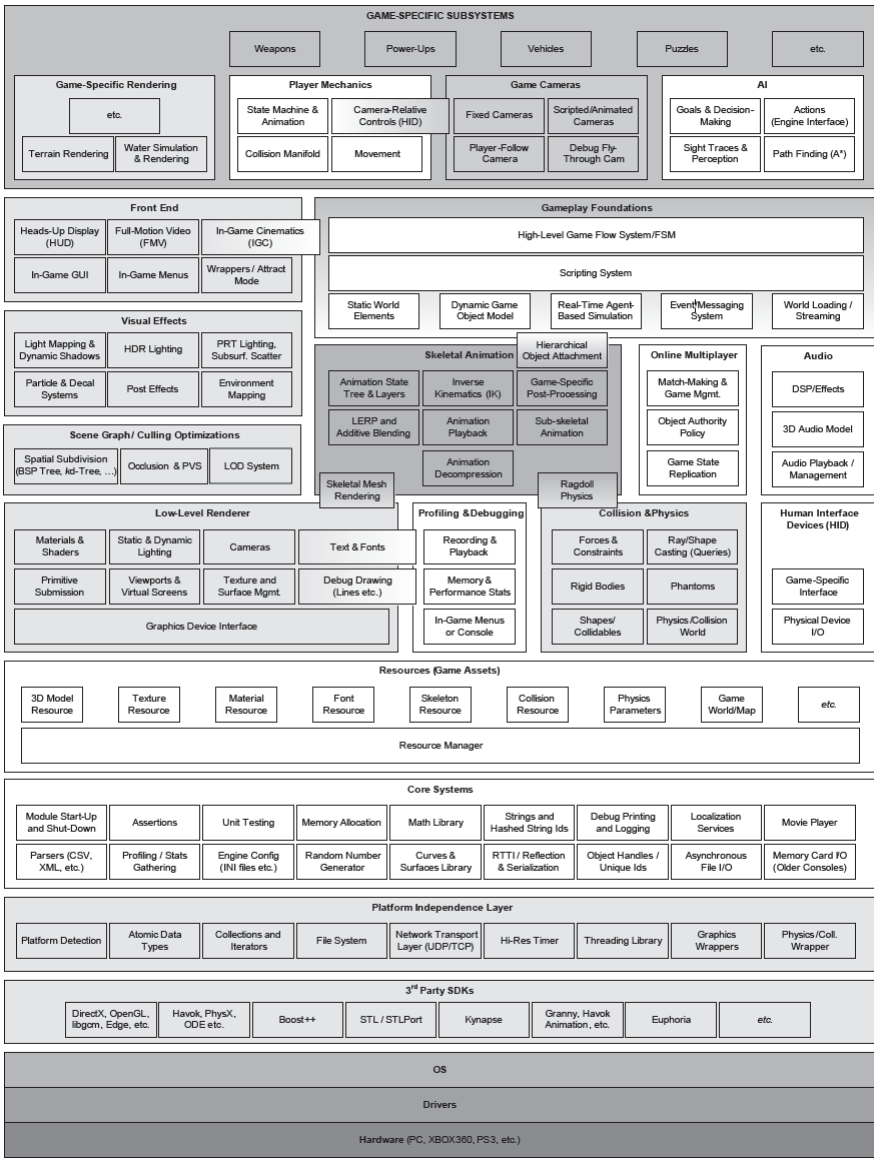
Pokretač igre obično se sastoji od dva glavna dijela: skupa alata za izradu igre i sustava za izvođenje igre u stvarnom vremenu (engl. *Runtime component*). U ovom poglavlju prvo ćemo opisati arhitekturu sustava za izvođenje igre, a zatim ćemo se upoznati sa alatima koji se koriste za izradu same igre.

Kao i svi softverski sustavi, pokretači igre su građeni u slojevima. Gornji slojevi ovise o donjim slojevima sustava, ali ne i obrnuto. Na slici 2.1 možemo vidjeti sve veće sustave od kojih se sastoji jedan tipičan pokretač igre. Po veličini slike 2.1 odmah možemo vidjeti da su pokretači igre definitivno veliki softverski sustavi.

Sada ćemo ukratko opisati komponente navedene na slici 2.1, a kasnije ćemo detaljnije analizirati veće podsustave kao što su sustavi za iscrtavanje i fiziku.

Hardver

Hardverska komponenta predstavlja računalni sustav ili konzolu na kojoj će se igra pokretati. Obično se radi o osobnim računalima baziranim na Microsoft Windows ili Linux sustavu, i Apple-ovim pametnim telefonima iPhone ili računalima Macintosh, ili konzolama kao što su Microsoft-ov Xbox, Sony-jev PlayStation i PlayStation Portable ili Nintendo-ov Wii i Switch.



Slika 2.1: Arhitektura sustava za izvođenje igre. Slika je preuzeta iz [2].

Upravljački programi uređaja

Upravljački programi uređaja (engl. *Device drivers*) se nalaze u vrlo niskom sloju, a obično ih pružaju proizvođači operativnog sustava ili određenog hardvera. Upravljački programi upravljaju resursima hardvera i služe kao posrednik u komunikaciji između gornjih slojeva

pokretača igre i samog hardvera. Na taj način gornji slojevi i operacijski sustav mogu jednostavno komunicirati sa hardverom, bez da znaju detalje o kojoj se točno varijanti hardvera radi.

Operativni sustav

Na osobnim računalima, operativni sustav je cijelo vrijeme pokrenut. On istovremeno upravlja sa više programa na jednom računalu, od kojih jedan može biti i igra. Ta igra mora dijeliti resurse računala sa svim ostalim trenutno pokrenutim programima i nikada ne može preuzeti punu kontrolu nad hardverom. Na igračim konzolama situacija je malo drukčija. Operativni sustav na konzolama obično zauzima vrlo malo memorije, pa je kompiliran direktno u izvršnu datoteku same igre. Igre na konzolama obično mogu preuzeti puno kontrolu nad harverom. Međutim, sa dolaskom novijih konzola na tržište kao što su PlayStation 3 i Xbox 360, ovo više nije u potpunosti točno. Kod tih konzola moguće je da operativni sustav prekine izvođenje igre, ili da zauzme određene resurse sustava. Dakle, granica između konzola i osobnih računala se sve više smanjuje, pa tako i razlika u razvijanju igara za te platfome.

Alati za razvoj softvera trećih strana

Mnogi pokretači igre, ali i drugi softverski sustavi koriste softvere trećih strana (engl. *Third-party SDK*) unutar svojih sustava. Pogledajmo nekoliko primjera.

Strukture podataka i algoritmi koriste se u svim softverskim sustavima, pa tako i u pokretačima igre. Primjer je standardna biblioteka programskog jezika C++ koja sadrži brojne strukture podataka i algoritme za njihovu manipulaciju.

Sustavi za iscrtavanje igre na ekran najčešće su građeni na gotovim bibliotekama sučelja. Najpoznatije biblioteke za iscrtavanje su OpenGL i Microsoft-ov DirectX.

Fizika u igrama obuhvaća dvije stvari, a to su detekcija kolizija (engl. *Collision detection*) i dinamika krutog tijela (engl. *Rigid body dynamics*). Popularni sustavi za upravljanje fizikom u igrama su Havok i PhysX.

I mnogi drugi sustavi mogu se naći u ovom sloju, kao što su sustavi za upravljanje animacijama likova, ili sustav za umjetnu inteligenciju.

Neovisnost o platformi

Sve češće na pokretače igre postoji zahtjev za mogućnost pokretanja na više od jedne platforme. Na taj način povećava se tržište i broj ljudi koji će moći igrati tu igru. Iz tog

razloga većina pokretača igre napravljena je sa slojem neovisnosti o platformi. Ovaj sloj štiti gornje slojeve, na način da ne moraju znati ništa o platformi na kojoj se nalaze. On je potreban jer postoji velik broj razlika između platformi, čak i unutar same standardne biblioteke programskog jezika C++.

Jezgreni sustavi

Svaki pokretač igre, ali također i svaki veći softverski sustav, zahtjeva "vreću" korisnih i često korištenih klasa, funkcija i softverskih alata (engl. *Utilities*). To mogu biti posebne strukture podataka i algoritmi, matematičke biblioteke ili sustav za alokaciju i dealokaciju memorije.

Sustav za upravljanje memorijom (engl. *Resource manager*)

Pokretač igre mora znati raditi sa puno različitih tipova podataka, kao što su 3D modeli, teksture, glazbene datoteke ili animacije likova. Bilo koju datoteku koju pokretač igre koristi nazivamo resurs igre (engl. *Game asset*). Ovaj sustav omogućava pristup svim mogućim tipovima resursa igre, i svim ostalim oblicima ulaznih podataka koje pokretač igre koristi.

Sustav za iscrtavanje (engl. *Rendering engine*)

Sustav za iscrtavanje (vizualizaciju) je jedan od najvećih i najkompleksnijih sustava pokretača igre. Ovaj sustav može se implementirati na više različitih načina, ali svaki takav sustav je sličan jer uvelike ovisi o grafičkom hardveru računala. Sami sustav dijeli se na više podsustava koje ćemo ovdje kratko opisati, a u kasnijim poglavljima detaljnije analizirati.

Iscrtavanje na nižoj razini (engl. *Low-level rendering*)

Ovaj podsustav zadužen je za iscrtavanje objekata koji su mu proslijeđeni, što brže i kvalitetnije moguće. Ti objekti mogu biti 2D slike, 3D modeli, linije, tekst ili bilo što drugo što se može iscrtati na ekran. Iscrtavanje na nižoj razini također je zaduženo za prikaz različitih materijala na objektima, te utjecaju dinamičkog osvijetljenja na te objekte.

Uklanjanje objekata (engl. *Culling*)

Podsustav iscrtavanja na nižoj razini samo iscrtava svaki objekt koji mu je proslijeđen. Međutim, neki objekti možda nisu trenutno vidljivi. Na primjer, sve što je iza lika u igri njemu je nevidljivo, pa nema smisla da na takve objekte trošimo vrijeme za iscrtavanje.

Ovaj podsustav je potreban kako bi identificirao samo one objekte koji su vidljivi i koje ima smisla slati na iscrtavanje.

Vizualni efekti

Moderni pokretači igre podržavaju velik broj vizualnih efekata. Neki od njih su:

- Sustavi čestica (engl. *Particle system*) - koriste se za efekte vatre, dima, kiše, itd.
- Sustavi naljepnica (engl. *Decal system*) - koriste se za rupe od metaka, otiske stopala, itd.
- Sjene
- Efekti cijelog zaslona - primjenjuju se na cijeli zaslon, nakon što je trenutna sličica (engl. *Frame*) iscrtana.

Frond end iscrtavanje

Mnoge igre zahtijevaju iscrtavanje 2D grafike unutar 3D svijeta. To mogu biti:

- Heads-up display (HUD) - prikaz informacija o stanju igre na zaslonu (npr. zdravlje lika, broj preostalih života, trenutna razina igre)
- Izbornici unutar igre
- Grafičko sučelje unutar igre

Alati za otkljanjanje grešaka i profiliranje

Računalne igre su sustavi koji se odvijaju u stvarnom vremenu, zbog čega se javlja potreba za profiliranjem performansi same igre u svrhu optimizacije performansi. Osim toga, memorijski resursi računala su ograničeni, pa razvojni programeri koriste razne alate za analizu memorije koju igra zauzima. Ti alati mogu tijekom izvođenja igre statistiku prikazivati direktno na ekran, ili ju mogu zapisati u neku vanjsku datoteku.

Kolizije i fizika

Detekcija kolizija važna je za svaku igru. Bez nje, objekti bi prolazili jedni kroz druge, a bilo kakva interakcija sa virtualnim svijetom bila bi nemoguća. Osim detekcije kolizija, neke igre imaju sustav za realistične ili polu-realistične dinamične simulacije. Taj sustav u industriji igara nazivamo sustav za fiziku, ili jednostavno fizika igre. U stvarnosti radi se o

dinamici krutog tijela, a zanimaju nas kretanja (kinematika) krutog tijela, te sile i okretni momenti (dinamika) koji uzrokuju kretanja krutog tijela.

Kolizije i fizika su dva odvojena sustava, ali su obično usko povezani. Nakon što smo detektirali koliziju dva objekta, gotovo uvijek tu koliziju rješavamo pomoću fizike. Danas gotovo svi razvojni timovi koriste gotove sustave za fiziku koji se integriraju u sami pokretač igre. Najpoznatija komercijalna riješenja su Havok i PhysX, dok je Open Dynamics Engine (ODE) jedan od najpoznatijih sustava otvorenog koda za fiziku.

Animacije

Svaka igra koja sadrži organske ili polu-organske likove kao što su ljudi, životinje ili roboti, zahtijeva sustav za animaciju tih likova. Pet je osnovnih tipova animacije: animacija slika u 2D, te animacija hijerarhije krutih tijela, skeletalne animacije, animacije vrhova i preobražaji (postupne promjene) u 3D.

Uređaji za ljudsko sučelje (engl. *Human interface devices*)

Svaka igra mora procesirati ulazne podatke od igrača, a ti ulazni podaci mogu dolaziti od raznih uređaja. Najčešće je to tipkovnica i miš ili igrači kontroler (engl. *Joystick*), ali postoje i mnogi drugi specijalizirani kontroleri kao što su volani, štapovi za pecanje ili plesne podloge. No ti uređaji nisu samo ulazni uređaji. Mnogi od njih su u nekom obliku i izlazni uređaji, jer sadrže značajke kao što su vibracija igračeg kontrolera ili produciranje zvuka.

Sustav koji je zadužen za kontrolu ovih uređaja nazivamo ulazno/izlazni sustav. Njegova zadaća je sakriti detalje samog ulaznog uređaja i njegove platforme, od kontrola igre na višoj razini. On obrađuje sirove podatke koji dolaze od uređaja, na način da detektira kada je neka tipka pritisnuta ili puštena, interpolira vrijednosti gljive na igračem kontroleru i slično. Također omogućava igraču da prilagodi kontrole, odnosno poveže fizičke kontrole sa logičkim funkcijama igre.

Zvuk

Iako često zapostavljen, zvuk u igrama je jednako važan kao i grafika igre. Svaki dobar pokretač igre mora sadržavati i sustav za upravljanje zvukom. On je zadužen za sviranje glazbenih tema u pozadini igre, razgovora među likovima, ali i mnogobrojnih zvučnih efekata unutar igre.

Umrežavanje

Mnoge igre dozvoljavaju da više ljudskih igrača igra igru unutar istog virtualnog svijeta. Igrači mogu zajedno igrati na istom računalu (obično 2 do 4 igrača), ili mogu igrati na vlastitom računalu u virtualnim svijetovima zajedno sa stotinama tisuća drugih igrača. Serveri na kojima se odvija igra obično su u vlasništvu proizvođača igre, ali postoje igre u kojima računalo jednog od igrača postaje server i tada to računalo nadgleda izvođenje igre.

Temeljni sustav *gameplay-a* (engl. *Gameplay foundation system*)

Izraz *gameplay* označava skup događaja koji se odvijaju u igri, pravila koja upravljaju virtualnim svijetom u kojem se igra odvija, mogućnosti likova i objekata u igri (mehanika igre), te ciljeva igre. *Gameplay* se obično implementira na jedan od dva načina: u originalnom programskom jeziku u kojem je implementiran i sami pokretač igre, ili u nekom višem skriptnom programskom jeziku, a ponekad i oboje.

Kako bi se odvojio programski kod *gameplay-a* od programskog koda pokretača igre, uvodi se sloj zvan temeljni sustav *gameplay-a*.
Svaki objekt koji se pojavljuje u igri nazivamo objekt igre (engl. *Game object*). Ti objekti mogu biti:

- statična pozadinska geometrija kao što su zgrade ili ceste
- dinamična kruta tijela kao što su kamenje ili limenke
- igrajući ili neigrajući likovi
- vozila, oružja ili projektili
- svijetla
- kamere (kroz koje igrač vidi virtualni svijet)
- i tako dalje

Osim objekata igre, u ovom sloju nalazi se i sustav događaja. Objekti igre moraju na neki način međusobno komunicirati, a to se može postići na više načina. Najjednostavniji od njih je da objekt koji šalje poruku pozove neku člansku funkciju objekta koji prima poruku. Drugi način je da objekt koji šalje poruku stvori malu strukturu podataka koju nazivamo događaj ili poruka i koja sadrži svoj tip i argumente, te ju zatim proslijedi objektu koji prima poruku. Prednost ovog načina je što se takvi događaji mogu spremati u red, pa izvesti u nekom budućem vremenu.

Sustav skriptnog jezika također se nalazi u ovom sloju. Skriptni jezik omogućava puno brži i jednostavniji razvoj igre, njezinog sadržaja i pravila. Bez skriptnog jezika, igra se nakon svake promjene mora ispočetka kompilirati. Međutim, sa korištenjem integriranog skriptnog jezika, promjene u logici igre mogu se napraviti samo modificiranjem skriptnog programskog koda. Promjene su odmah vidljive ponovnim pokretanjem igre. Neki pokretači igre čak dozvoljavaju izmjene u skriptnom programskom kodu za vrijeme izvođenja igre, a promjene su vidljive odmah, bez ponovnog pokretanja.

Podsustavi specifični za igru (engl. *Game-specific subsystems*)

Na samom vrhu slojeva nalaze se podsustavi sprecifični za konkretnu igru. Oni uključuju mehaniku i sposobnosti likova igre, sustav umjetne inteligencije neigrajućih likova, sustav kamera unutar igre, sustavi vozila, oružja i tako dalje. Možemo reći da se linija koja odvaja pokretača igre od same igre nalazi upravo između podsustava specifičnih za igru i temeljni sustav *gameplay-a*, iako ona nikada nije potpuno jasna.

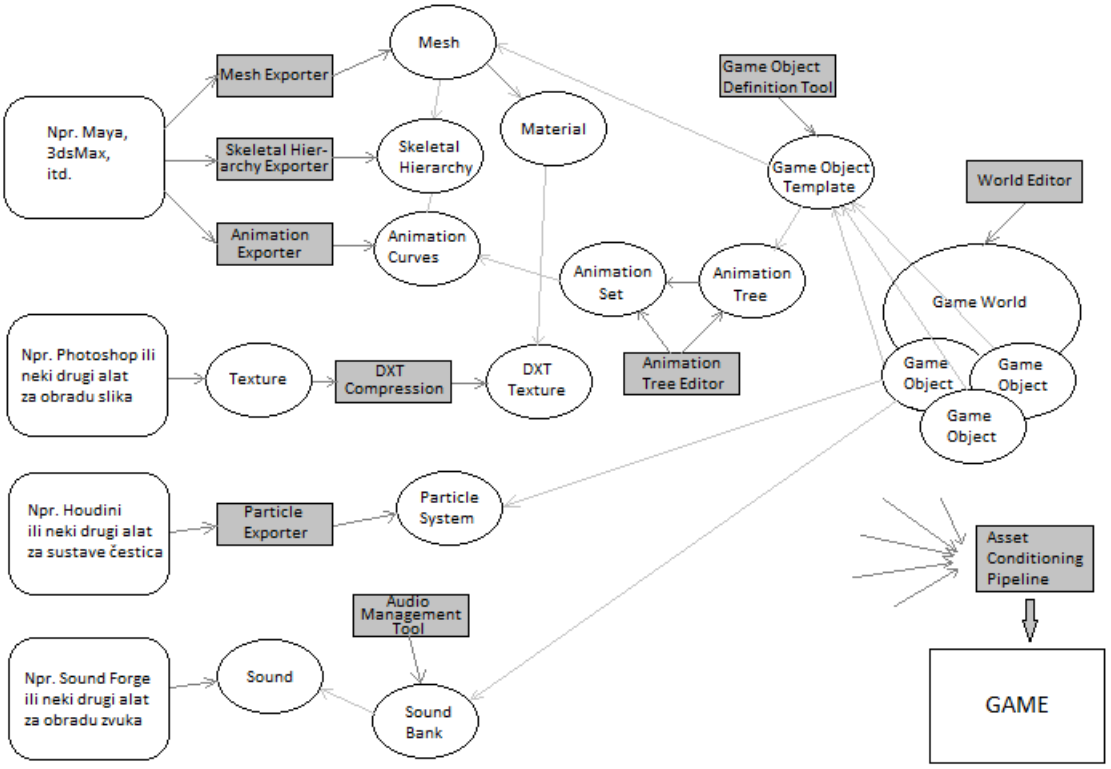
2.4 Alati i proces izrade resursa igre (engl. *Tools and the Asset pipeline*)

Svaki pokretač igre koristi veliki broj različitih podataka i datoteka, a oni su u obliku resursa igre, konfiguracijskih datoteka, skripti skriptnog jezika, i tako dalje. Većina tih datoteka nastaje u različitim alatima koji nisu dio pokretača igre. Na primjer, jednom kada napravimo neku sliku ili 3d model, možemo ju koristiti u bilo kojem pokretaču igre koji podržava taj tip podatka. Na prethodnoj slici 2.2 možemo vidjeti tipove resursa igre koji se tipično mogu naći u modernim pokretačima igre.

Alati za izradu digitalnog sadržaja

Igre su same po sebi multimedijske aplikacije. Sve ulazne datoteke koje pokretač igre koristi izrađene su u nekom od alata za izradu digitalnog sadržaja. To mogu biti 3D modeli, slike (teksture), animacije ili zvučne datoteke.

Iako je većina alata namijenjena za izradu jednog tipa podataka, postoje alati koji u kojima je moguća izrada više tipova podataka. Na primjer, Autodesk-ovi programi Maya i 3D Studio Max koriste se za izradu 3D modela i animacija. Za izradu i obradu slika (tekstura) vjerojatno je najpoznatiji Adobe-ov program Photoshop, dok je za izradu datoteka zvuka popularan SoundForge.



Slika 2.2: Alati i proces izrade resursa igre

Proces uvoza resursa igre

Datoteke izrađene u alatima za izradu digitalnog sadržaja često nisu u formatu koji je odmah moguće iskoristiti u pokretaču igre. Naime, format datoteke iz tih alata obično sadrži puno više informacija nego što je to potrebno. Na primjer, može sadržavati povijest svih izmjena u datoteci, što pokretaču igre nije potrebno. Kada bi takve formate obrađivali prilikom pokretanja igre, to bi oduzelo puno vremena. Zato se podaci stvoreni u tim alatima obično izvezu u jednostavniji i standardizirani format koji se onda koristi u igrama.

Jednom kada se ti podaci izvezu u standardizirani format, obično se prilikom uvoza u pokretač igre moraju dodatno obraditi. Na primjer, ako želimo uvesti sliku u naš pokretač igre, tada konačan format slike ovisi o platformi za koji želimo napravit igru. Na mobilnim uređajima možda nije potrebna slika velike rezolucije, pa se uvezena slika smanjuje, dok je za stolna računala ta slika potrebna u većoj rezoluciji.

3D modeli

U 3D igrama sva geometrija koju vidimo su 3D modeli. 3D modeli se sastoje od više dijelova. Najvažniji dio 3D modela je njegova mreža (engl. *Mesh*). Mreža je kompleksno geometrijsko tijelo sastavljeno od trokuta i vrhova. Mreža je ono što daje oblik 3D modelu, a jedan 3D model može se sastojati od više pod-mreža. Svakoj mreži 3D modela pridružen je jedan ili više materijala (engl. *Material*). Pomoću materijala definiramo vizualna svojstva površine objekta (boju, refleksivnost, hrapavost, itd.). 3D model u sebi također može sadržavati i animacije, te još neke meta-podatke koje pokretač igre koristi. Najčešći formati koji se koriste za datoteke 3D modela u igrama su .fbx, .3ds i .obj.

Skeletne animacije

Kod skeletnih animacija koristi se poseban tip mreže 3D modela koji nazivamo skeletna mreža. Skeletnoj mreži pridružena je skeletna hijerarhija, koja omogućava realistične animacije 3D modela. Kažemo da je mreži 3D modela pridružen kostur, a svaki vrh mreže sadrži listu svih zglobova tog kostura kojima je pridružen. Zapravo animiramo kostur modela, a mreža modela prati taj kostur. Za iscrtavanje animacije i skeletne mreže, pokretaču igre su potrebna tri podatka: mreža 3D modela, skeletna hijerarhija (imena zglobova i odnosi roditelj-dijete), te jedan ili više animacijskih isječaka (engl. *Animation clip*) koji određuju kako se zglobovi kostura pomiću tijekom vremena.

Ostali alati

Postoji puno alata u kojima se izrađuju zvučni isječci, ali i puno formata zvučnih isječaka. Popularan program za izradu zvučnih isječaka je SoundForge, a najčešći formati su .wav, .ogg i .mp3.

Današnje igre koriste sve veći broj specijalnih efekata, a najčešće su to sustavi čestica. To su sustavi kojima se izrađuju vizualni efekti vatre, dima, eksplozija, lišća koje pada sa stabla, vode i slično. Za to koriste veliki broj malih sličica ili 3D modela.

Virtualni svijet igre je mjestu u kojem se cijela igra odvija. Alati za oblikovanje virtualnog svijeta obično su dio samog pokretača igre, a ponekad se oblikuju u nekom od alata za izradu 3D modela.

2.5 Poznati pokretači igre

Danas postoji veliki broj pokretača igre, koji su implementirani u različitim programskim jezicima i namijenjeni za pokretanje na različitim platformama. Oni također dolaze sa različitim licencama. Neki od njih su otvorenog koda, neki su besplatni za korištenje, neki su komercijalni i zahtijevaju plaćanje, dok su neki u vlasništvu tvrtki koje se bave razvojem igara.

Vjerojatno najpoznatiji i najkorišteniji pokretač igre danas je Unity3D. Jedna od glavnih značajki Unity-a je mogućnost kompiliranja igre na više platformi. Jednom razvijena igra može se uz manje promjene pokrenuti na više od 20 različitih platformi koje Unity podržava. Unity se odlikuje brojnim funkcionalnostima, zbog čega je u njemu moguće napraviti skoro svaki postojeći tip igre. Za osobno korištenje je potpuno besplatan, što je privuklo velik broj novih razvojnih programera.

Isto tako poznat i često korišten pokretač igre Unreal Engine korišten je za izradu brojnih visoko kvalitetnih igara, a njegova prva verzija dostupna je još od 1998. godine. Unreal Engine je otvorenog koda, što omogućava lake izmjene samog pokretača igre.

GameMaker je pokretač igre koji za svoje korištenje ne zahtijeva nikakvo programersko znanje. Upravo zbog toga je zanimljiv velikom broju ljudi koji želi razvijati igre, ali nemaju znanja o programiranju. Međutim, to svojstvo ga ujedno i ograničava, odnosno smanjuje broj mogućnosti koje igra može imati.

Nešto noviji pokretač igre koji se nedavno pojavio na tržištu je CryEngine 3. On i njegovi prethodnici bili su u vlasništvu tvrtke Crytek koja je u njima razvila mnoštvo svojih igara. Zbog kratkog vremena na tržištu još nema veliku zajednicu korisnika, ali postaje sve popularniji zbog odličnih grafičkih mogućnosti. Više o ovim i ostalim pokretačima igre može se saznati na poveznici [12].

Poglavlje 3

Iscrtavanje

Prva stvar koja većini ljudi padne na pamet kada razmišljaju o pojmu video igre vjerojatno je zadivljujuća trodimenzionalna grafika. Za prikaz te grafike na ekranu računala zadužen je sustav za iscrtavanje. Iscrtavanje grafike je jako široka tema, a u ovom poglavlju obradit ćemo samo osnove. Prvo ćemo predstaviti osnovne koncepte i matematiku koja stoji iza njih. Zatim ćemo ukratko analizirati proces iscrtavanja, iz softverske i hardverske perspektive. Na kraju ćemo reći nešto o vizualnim efektima, te o osvijetljenju koje je uvelike zaslužno za postizanje realizma unutar video igre.

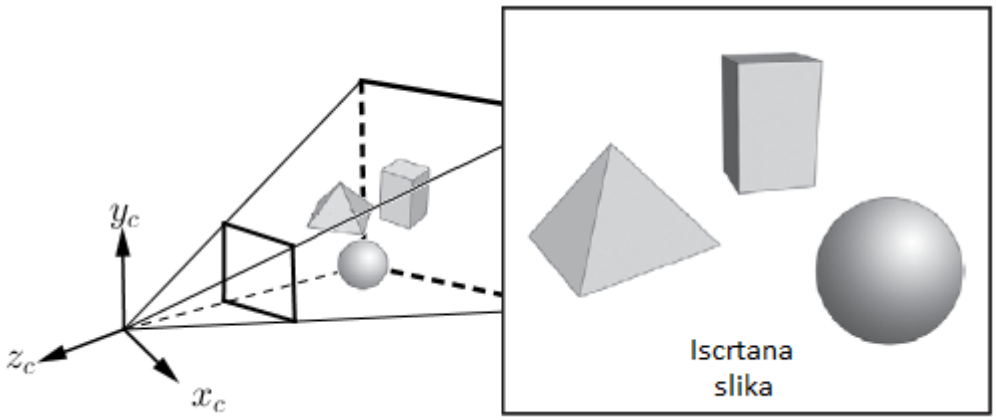
U ovom poglavlju većinom ćemo govoriti o 3D iscrtavanju, jer danas gotovo da ne postoji igra koja koristi samo 2D grafiku. Neke 2D igre koriste pojedine mogućnosti 3D iscrtavanja, dok neke igre samo prividno izgledaju kao 2D igre, a zapravo su u potpunosti napravljene 3D grafikom. Osim toga, gotovo svi koncepti 3D iscrtavanja mogu se primijeniti i na 2D iscrtavanje.

3.1 Što je iscrtavanje?

Iscrtavanje je proces transformacije nekog objekta u njegovu vizualnu reprezentaciju. Cijeli proces sa stoji se od sljedećih osnovnih koraka:

- **Virtualna scena** se opisuje u terminima 3D površina koje su reprezentirane u nekoj matematičkoj formi.
- **Virtualna kamera** postavlja se na određenu poziciju i pod određenom rotacijom kako bi reproducirala željeni pogled na scenu. Kamera je obično modelirana kao idealizirana žarišna točka u prostoru sa pravokutnom površinom za snimanje koja se nalazi malo ispred nje, a ta površina sastoji se od virtualnih svjetlosnih senzora koji odgovaraju pikselima ekrana.

- Definiraju se razni izvori svijetlosti. Oni stvaraju zrake koje će se odbijati od površina objekata virtualnog svijeta i u konačnici doći do svjetlosnih senzora na površini virtualne kamere.
- Opisuju se vizualna svojstva površina u sceni. Ta svojstva opisuju način na koji svijelost djeluje na koju površinu.
- Za svaki piksel na površini za snimanje, sustav za iscrtavanje računa boju i intenzitet zrake svijetlosti koja konvergira na žarišnu točku kamere kroz taj određeni piksel. Ovaj proces nazivamo rješavanje jednadžbe iscrtavanja.



Slika 3.1: Princip iscrtavanja

Postoji više različitih tehnologija koje se mogu koristiti za izvođenje prethodno opisanih koraka. Glavni cilj obično je postizanje fotorealizma, iako neke igre ciljaju na stilizirani izgled. Dakle, razvojni programeri igre i umjetnici koji stvaraju virtualni svijet pokušavaju opisati scenu što je realističnije moguće i koriste modele osvijetljenja koji su što sličniji modelima u stvarnom svijetu. Gledajući iz ovog konteksta, postoji više vrsta tehnologija za iscrtavanje, a kreću se od tehnologija dizajniranih za izvođenje u stvarnom vremenu pod cijenu kvalitete vizualnog izgleda, do tehnologija dizajniranih za iscrtavanje fotorealističnih scena koje nisu namijenjene za izvođenje u stvarnom vremenu.

Sustavi za iscrtavanje u stvarnom vremenu ponavljaju prethodno opisane korake i prikazuju iscrtane slike brzinom od 30 do 60 sličica u sekundi, kako bi stvorili iluziju pokreta. To znači da sustav za iscrtavanje u realnom vremenu ima najviše 33.3 milisekundi za iscrtavanje jedne slike, kako bi održao brzinu od 30 sličica u sekundi. U stvarnosti ima

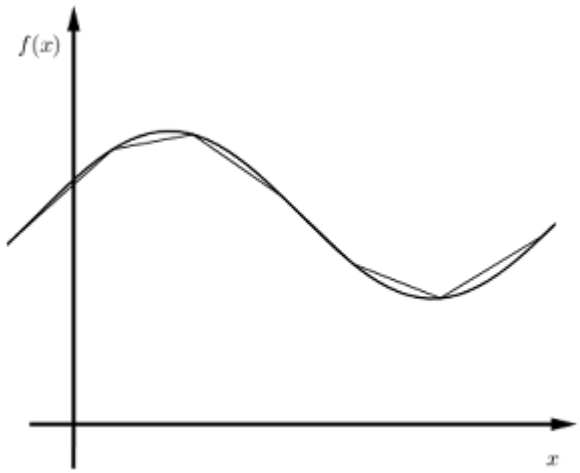
mnogo manje vremena, jer je vrijeme za izvođenje potrebno i ostalim sustavima kao što su sustav za animaciju, umjetnu inteligenciju, detekciju kolizija, simulaciju fizike i ostali. Uzimajući u obzir da je sustavima za iscrtavanje u filmskoj industriji za iscrtavanje jedne slike potrebno između nekoliko minuta, pa čak i do nekoliko sati, možemo reći da je kvaliteta današnje računalne grafike iscrtane u stvarnom vremenu stvarno zapanjujuća.

3.2 Virtualna scena

Objekti u stvarnom svijetu mogu biti čvrsti, kao što je cigla, ali mogu biti i amorfni, kao što je oblak dima. U svakom slučaju, možemo reći da zauzimaju neki volumen u trodimenzionalnom prostoru. Dakle, objekti mogu biti neprozirni (svijetlost ne može proći kroz njih), prozirni (svijetlost prolazi kroz njih, objekti iza njih vide se jasno) i poluprozirni (svijetlost prilazi kroz njih, ali objekti iza njih ne vide se jasno nego su mutni).

Neprozirni objekti iscrtavaju se samo koristeći svojstva svoje površine. Ne moramo znati kako izgleda unutrašnjost neprozirnog objekta kako ga iscrtati, jer svijetlost ne može proći kroz njegovu površinu. Za iscrtavanje prozirnog ili poluprozirnog objekta moramo modelirati kako će se svijetlost odbijati, prelomiti, raspršiti i apsorbirati tijekom svog prolaska kroz volumen objekta. Međutim, ova računanja bila bi prekompleksna, pa većina pokretača igara iscrtava prozirne i poluprozirne objekte na način sličan iscrtavanju neprozirnih objekata. Uvodi se jednostavna numerička mjera prozirnosti koji nazivamo *alpha*, koja opisuje koliko je objekt proziran ili neproziran. Ovaj pristup može dovesti do nekih vizualnih anomalija, ali u većini slučajeva postoje aproksimacije koje ih umanjuju. Čak se i amorfni objekti kao što su oblaci ili dim obično iscrtavaju pomoću velikog broja poluprozirnih pravokutnih objekata. Dakle, možemo reći da je glavni i primarni zadatak većine sustava za iscrtavanje upravo iscrtavanje površina.

Površine u igrama obično su modelirane korištenjem mreža trokuta. Trokuti služe kao dijelovi koji aproksimiraju površinu u trodimenzionalnom prostoru, slično kao što segmenti pravaca aproksimiraju krivulju u dvodimenzionalnom prostoru.



Slika 3.2: Linearna aproksimacija funkcije segmentima (dužinama)

Trokuti su odabrani za iscrtavanje u stvarnom vremenu jer imaju sljedeća svojstva:

- Trokut je najjednostavniji mnogokut. Površinu ne možemo odrediti sa manje od tri vrha.
- Trokut je uvijek planaran. Mnogokuti sa više od tri vrha ne moraju imati ovo svojstvo, odnosno mogu imati vrh koji ne leži u istoj ravnini kao i ostali vrhovi.
- Trokut ostaje trokut nakon gotovo svih vrsta afinih transformacija. U najgorem slučaju, trokut će se degradirati u segment pravca (dužinu), ako na njega gledamo pod određenim kutem.
- Sav komercijan hardver za ubrzanje grafike (grafičke kartice) je dizajniran za iscrtavanje trokuta. Počevši od najranijih grafičkih kartica i od najranijih 3D igara kao što je Doom, svi su koristili trokut kao odabir.

Konstrukcija mreže trokuta

Trokut je u prostoru definiran pomoću tri radij-vektora koja predstavljaju položaj njegovih vrhova, nazovimo ih $p1$, $p2$ i $p3$. Bridovi trokuta lako se mogu izračunati oduzimanjem vektora susjednih vrhova. Na primjer:

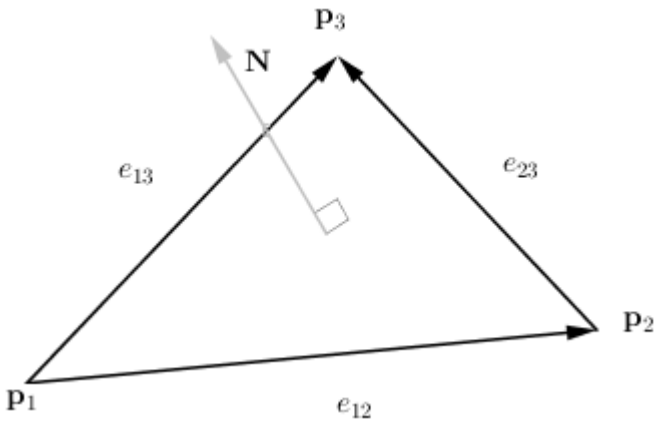
$$e_{12} = p_2 - p_1 \tag{3.1}$$

$$e_{13} = p_3 - p_1 \tag{3.2}$$

$$e_{23} = p_3 - p_2 \tag{3.3}$$

Osim ovih vektora, bitan nam je i vektor normale. On nam pokazuje u kojem smjeru je trokut okrenut. Računanjem normaliziranog vektorskog produkta bilo koja dva brida dobivamo normalu N.

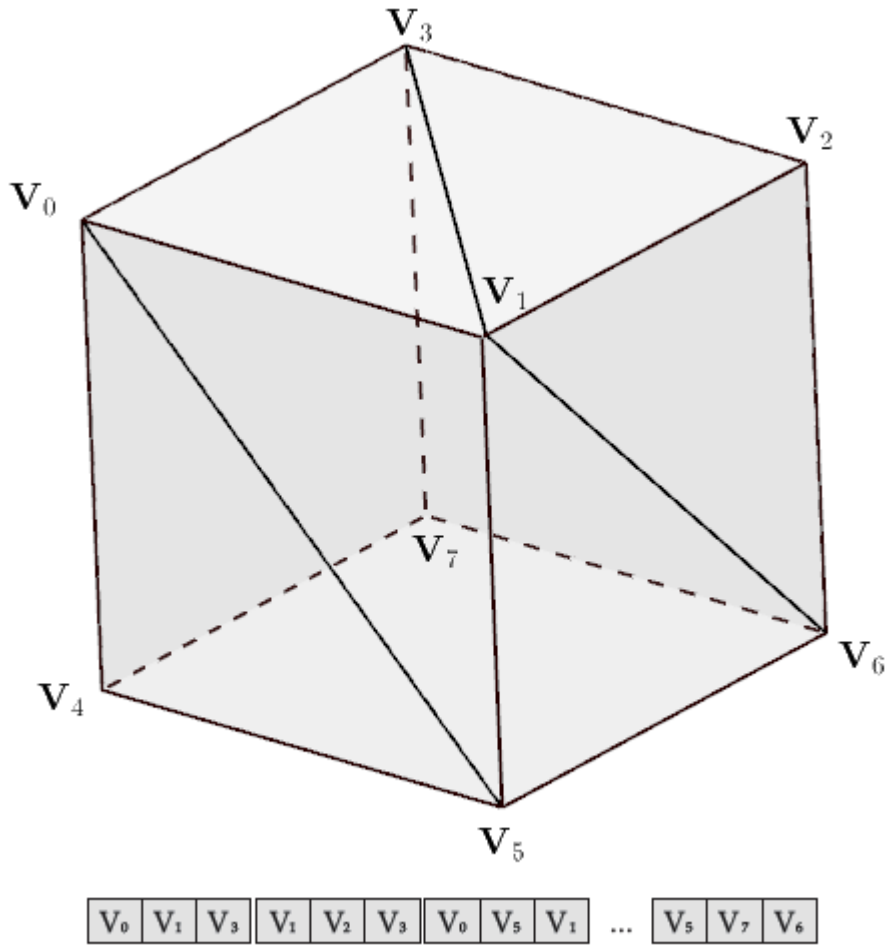
$$N = \frac{e_{12} \times e_{13}}{|e_{12} \times e_{13}|} \tag{3.4}$$



Slika 3.3: Prikaz stranica trokuta i vektora normale izračunatih iz vrhova

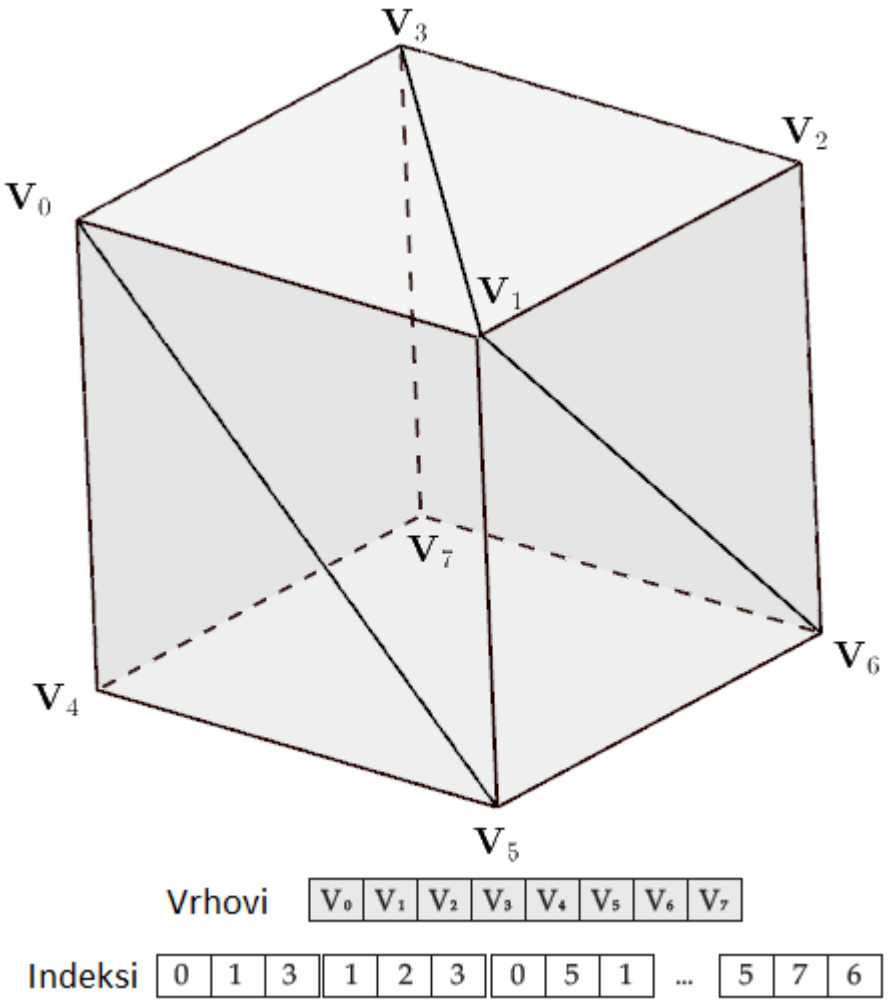
Kako bi znali smjer vektora normale, moramo definirati koju stranu trokuta ćemo smatrati prednjom (vanjska površina objekta), a koju stražnjom (unutarnja površina objekta). Kako bi definirali strane trokuta, moramo odrediti smjer obilaska vrhova (engl. *winding order*), koji može biti u smjeru kazaljke na satu ili u smjeru suprotnom od kazaljke na satu. Odabir samog smjera nije bitan, ali je bitno da isti smjer koristimo kod svih 3D modela unutar cijele igre. Bitno je odrediti strane trokuta, jer obično ne želimo trošiti vrijeme na iscrtavanje obje strane trokuta, one nikada nisu vidljive istovremeno. Ovaj postupak zovemo uklanjanje pozadine trokuta (engl. *backface culling*).

Nakon što imamo pojam trokuta, možemo definirati mrežu trokuta. Najlakši način za definirati mrežu trokuta je jednostavno zapisati niz vrhova u grupama po troje, gdje svaka trojka predstavlja jedan trokut. Ovu strukturu podataka nazivamo lista trokuta.



Slika 3.4: Lista trokuta

Možemo primijetiti da se mnogi vrhovi ponavljaju više puta, dakle zapravo zauzimamo puno više memorije nego što je to potrebno. To je jedan od razloga zašto većina sustava za iscrtavanje koristi strukturu podataka koju zovemo indeksirana lista trokuta. Ona se sastoji od dva niza. Prvi niz sadrži sve vrhove, bez duplikata. Drugi niz sadrži indekse vrhova, a služi za definiranje trojki vrhova koji čine trokute. U drugom nizu opet imamo ponavljanja indeksa vrhova, ali ovaj puta ponavljaju se samo brojevi, a ne cijele strukture podataka koje predstavljaju jedan vrh.

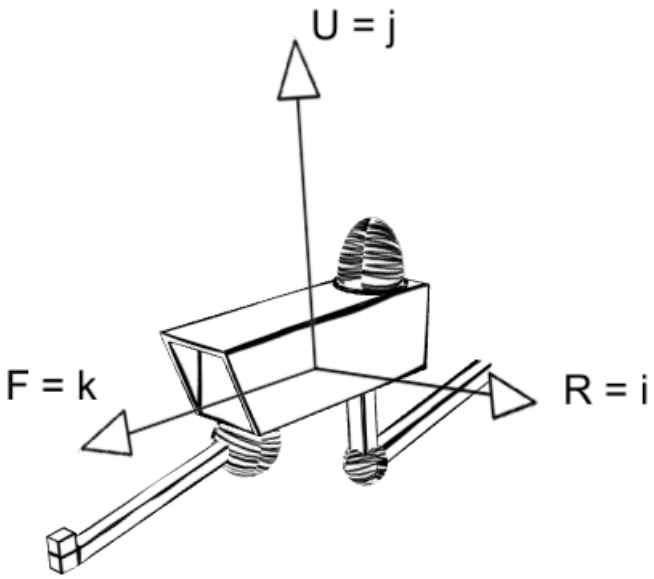


Slika 3.5: Indeksirana lista trokuta

Prostor modela i prostor svijeta

Radij-vektori vrhova mreže trokuta obično su zadani relativno sa lokalnim koordinatnim sustavom koji nazivamo prostor modela ili lokalni prostor (engl. *model space*). Ishodište ovog sustava obično je u centru objekta, ili na nekom drugom odgovarajućem mjestu, kao što je npr. dno staklene čaše. Koordinatne osi orijentiraju se s obzirom na prirodne ”naprijed”, ”lijevo” (ili ”desno”) i ”gore” smjerove modela. Tada možemo pridružiti te koordinatne osi trima vektorima baze: i, j i k. Vektori se obično pridružuju na način: ”$lijevo$” = i, ”$gore$” = j, ”$naprijed$” = k. Pridruživanje je proizvoljno, ali bitno je da je konzistentno za

sve modele u igri. Na slici 3.6 možemo vidjeti jedan primjer pridruživanja koordinatnih osi.



Slika 3.6: Jedan mogući odabir koordinatnih osi prostora modela

Unutar scene smješten je veliki broj mreža trokuta koje pozicioniramo i rotiramo unutar zajedničkog koordinatnog sustava koji nazivamo prostor svijeta ili globalni prostor (engl. *world space*). Svaka mreža u sceni se može pojaviti više puta, na primjer u jednoj ulici možemo imati više identičnih rasvjetnih stupova. Svaki takav objekt zovemo instanca mreže. Svaka instanca mreže sadrži referencu na svoju mrežu trokuta, te matricu transformacije koja transformira vrhove te instance mreže iz prostora modela u prostor svijeta. Ova matrica naziva se matrica **model-u-svijet** (engl. *model-to-world matrix*). Ona služi za rotaciju, skaliranje i translaciju objekta iz prostora modela u prostor svijeta.

3.3 Vizualna svojstva površine

Kako bi se površina pravilno osvijetlila i iscrtala, moramo sustavu za iscrtavanje dati vizualna svojstva te površine. Svojstva površine uključuju informacije o geometriji, kao što su smjerovi vektora normale na raznim točkama površine. Osim toga opisuju način na koji bi svijetlost trebala djelovati na površinu, uključujući boju površine, refleksivnost, hrapavost, stupanj prozirnosti i ostala svojstva.

Uvod u boju

U računalnoj grafici najčešće se koristi RGB model boja. Kratica RGB odgovara crvenoj, zelenoj i plavoj boji (engl. *red, green, blue*). Boje su predstavljene uređenim trojkama, gdje trojka $(0, 0, 0)$ predstavlja crnu boju, $(1, 1, 1)$ predstavlja bijelu boju, a $(1, 0, 0)$, $(0, 1, 0)$, $(0, 0, 1)$ redom predstavljaju crvenu, zelenu i plavu boju. Za prozirne objekte uvodi se četvrta komponenta koja predstavlja prozirnost boje. Takav model nazivamo RGBA ili ARGB, a kratica A dolazi od *alpha*.

Atributi vrhova

Najjednostavniji način za opisivanje površine je diskretno odrediti točke na toj površini i za njih odrediti svojstva. Vrhovi mreže trokuta su dobro mjesto za spremanje svojstava površine, a u tom slučaju zovemo ih atributi vrhova. Atribute vrhova određuju razvojni programeri, ali postoje neka atributi koji se gotovo uvijek koriste. To su:

- Vektor pozicije ($p_i = [p_{ix}, p_{iy}, p_{iz}]$). Ovo je 3D pozicija i-tog vrha u mreži. Obično je zadana u lokalnom koordinatnom sustavu modela.
- Normala vrha ($n_i = [n_{ix}, n_{iy}, n_{iz}]$). Ovaj vektor definira jedinični vektor normale površine na poziciji i-tog vektora. On se koristi za računanje kuta pod kojim će se odbiti svijetlost koja dolazi do tog vrha.
- Tangenta ($t_i = [t_{ix}, t_{iy}, t_{iz}]$) i bitangenta vrha ($b_i = [b_{ix}, b_{iy}, b_{iz}]$). Ova dva jedinična vektora međusobno su okomita, a također su okomita i na normalu vrha n_i. Ova tri vektora (n_i, t_i, b_i) zajedno čine koordinatni sustav koji zovemo tangentni prostor.
- Difuzna boja (engl. *diffuse color*) ($d_i = [d_{Ri}, d_{Gi}, d_{Bi}, d_{Ai}]$). Ovaj vektor opisuje boju površine.
- Boja refleksije (engl. *specular color*) ($s_i = [s_{Ri}, s_{Gi}, s_{Bi}, s_{Ai}]$). Ovaj vektor opisuje boju refleksije koja se pojavi kada se svijetlost reflektira sa površine direktno u kameru.
- Koordinate teksture ($u_{ij} = [u_{ij}, v_{ij}]$). Koordinate teksture omogućuju da se slika koju nazivamo tekstura mapira (projicira) na površinu mreže 3D modela. Koordinata teksture (u, v) opisuje lokaciju unutar teksture koja je pridružena određenom vrhu. Jedan vrh može biti pridružen i više nego jednoj teksturi, pa tako možemo imati i više od jedne koordinate teksture. Razlikujemo ih preko indeksa j.

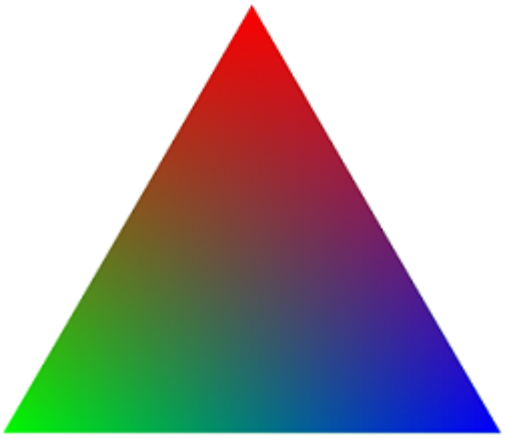
Tipična struktura koja sadrži atribute vrha izgleda ovako:

```
struct Vertex1P1N1UV {
  Vector3 m_p;       // pozicija
  Vector3   m_n;     // normala
  F32 m_uv[2];        // (u, v) koordinate teksture
};
```

Interpolacija atributa

Atributi vrhova su diskretna aproksimacija svojstva površine, ali kada na ekran iscrtavamo trokut, nas zanima kako izgledaju točke na površini tog trokuta koje odgovaraju pikselima na ekranu. Kažemo da nas u konačnici zanimaju atributi po pikselu, a ne atributi po vrhu.

Vrijednosti atributa površine mreže trokuta po pikselu određujemo jednostavnom linearnom interpolacijom vrijednosti atributa po vrhu. Ovu interpolaciju primjenjenu na boje vrhova nazivamo Gouraud-ovo sjenčanje. Primjena interpolacije primjenjuje se i na ostale vrste atributa vrhova, kao što su vektori normale i koordinate teksture.



Slika 3.7: Gouraud-ovo sjenčanje trokuta čiji vrhovi su u različitim nijansama boje

Teksture

Kako bi se poboljšala kvaliteta izgleda 3D modela, koriste se slike koje nazivamo **teksture**. Tekstura najčešće sadrži informacije o bojama i projicirana je na površinu 3D modela, odnosno na njegovu mrežu trokuta. Moguće je da tekstura sadrži i druge informacije koje opisuju površinu 3D modela, a ne samo njegovu boju.

Najčešći tip teksture je difuzna mapa (engl. *diffuse map*). Ona opisuje boje na površini 3D modela. Osim nje često se koristi i mapa normala (engl. *normal maps*). Mapa normala sadrži informacije o vektoru normale koji je zakodiran kao RBG vrijednost. Mapa refleksije sadrži podatke koji govore koliko je površina reflektirajuća. Teksture možemo koristiti za pospremanje bilo kakvih informacija koje nam trebaju. Na primjer, vrijednosti neke kompleksne matematičke funkcije možemo zakodirati u teksturu i koristiti nju umjesto da izračunavamo samu funkciju tijekom izvođenja igre.

Slika 3.8: Primjer korištenja teksture. Tekstura modela prikazana je u trodimenzionalnom prostoru (lijevo) i u prostoru teksture (desno).

Materijali

Konačan izgled površine 3D modela izračunava kratki program koji nazivamo shader. Shader program koristi podatke o mreži trokuta, teksture i ostale parametre kako bi izračunao konačnu boju piksela koja će se iscrtati na ekranu. Malo više o shader-ima govorit ćemo kasnije.

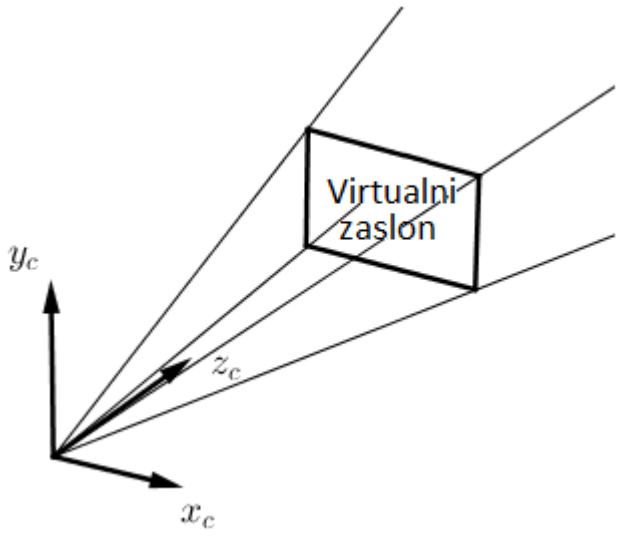
Kompletan opis vizualnih svojstava površine nazivamo materijal. On sadrži specifikacije tekstura koje koristi za određivanje konačnog izgleda površine, shader program koji koristi za izračunavanje iscrtavanja i sve potrebne ulazne parametre. 3D model može imati

više materijala na sebi. Na primjer, 3D model čovjeka može imati različite materijale za kožu, kosu i odjeću.

3.4 Virtualna kamera

Virtualnu kameru definiramo kao idealiziranu žarišnu točku sa pravokutnom površinom za snimanje koja se nalazi malo ispred nje. Taj pravokutnik sastoji se od virtualnih svjetlosnih senzora koji odgovaraju pikselima ekrana. Proces iscrtavanja može se zamisliti kao proces određivanja koja boja i u kojem svijetlosnom intenzitetu će biti zabilježena u svakom od tih virtualnih senzora.

Žarišna točka virtualne kamere ishodište je trodimenzionalnog koordinatnog sustava kojeg nazivamo prostor kamere ili prostor pogleda (engl. *view space*). Kamera obično gleda u smjeru pozitivne z osi u prostoru kamere, dok se os y pruža u smjeru prema gore, a os x u smjeru prema lijevo ili desno.



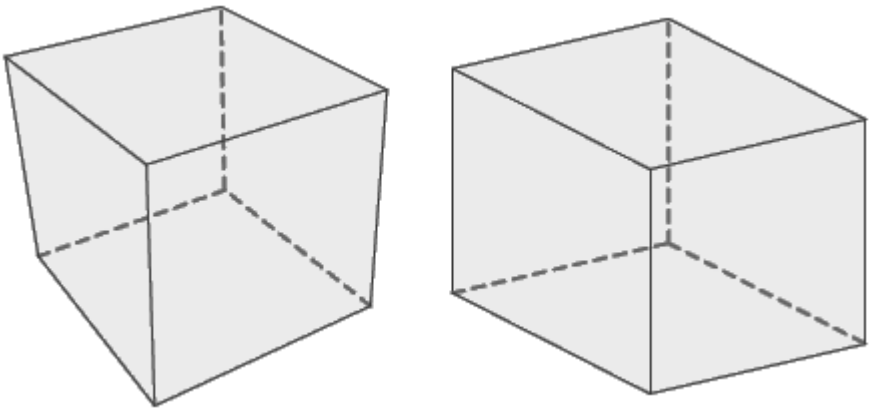
Slika 3.9: Osi kamere

Pozicija i rotacija kamere može se odrediti pomoću kamera-u-svijet (engl. *camera-to-world*) matrice, slično kao što se 3D model smješta u scenu pomoću njegove model-u-svijet matrice.

Kada se neka mreža trokuta treba iscrtati na ekran, trebaju nam njezini podaci izraženi pomoću koordinata prostora kamere. Dakle, vrhovi te mreže trokuta prvo se transformiraju

iz prostora modela u prostor svijeta, a zatim iz prostora svijeta u prostor kamere. Za prvu operaciju koristimo matricu model-u-svijet pridruženu mreži trokuta, dok za drugu operaciju koristimo matricu svijet-u-kameru koja je zapravo inverz matrice kamera-u-svijet pridružene kameri. Konačna matrica model-u-kameru dobiva se umnoškom matrica model-u-svijet i svijet-u-kameru.

Postoje dvije vrste projekcije scene koju kamera vidi na 2D površinu zaslona. Prva vrsta projekcije je perspektivna projekcija. Ona se koristi u većini slučajeva, a zapravo je slična kameri u stvarnom životu. Kod ove projekcije, objekti koji su udaljeniji od kamere izgledaju manje od objekata bliže kameri. Druga vrsta projekcije je ortografska projekcija, kod koje je isti objekt uvijek iste veličine, neovisno o udaljenosti od kamere.

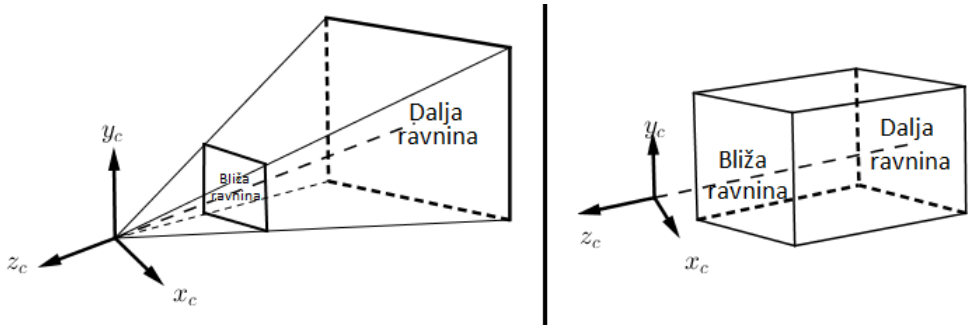


Slika 3.10: Kocka iscrtana korištenjem prespektivne projekcije (lijevo) i ortografske projekcije (desno)

Volumen kamere

Dio prostora koji kamera može vidjeti nazivamo volumen kamere. Taj volumen definiran je sa šest ravnina. Ravnina koju zovemo bliža ravnina je određena virtualnim senzornim pravokutnikom. Druga ravnina, koju zovemo dalja ravnina, paralelna je bližoj ravnini, a koristi se za optimizaciju iscrtavanja. Naime, objekte jako udaljene od kamere nije potrebno iscrtavati, jer gotovo nikada nisu vidljivi.

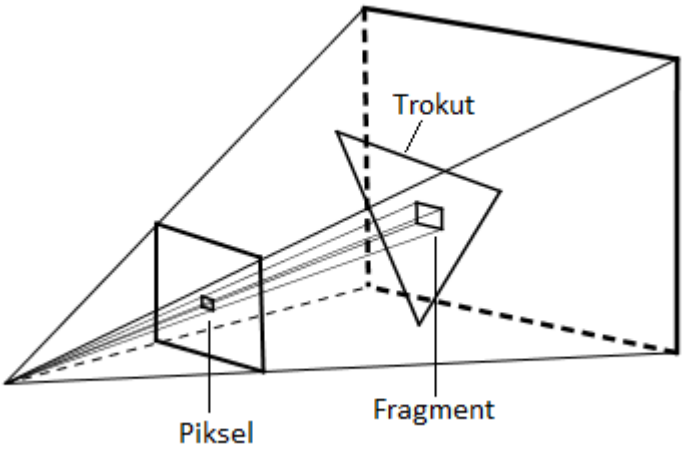
Kod perspektivne projekcije, volumen kamere ima oblik krnje piramide, dok kod ortografske projekcije volumen kamere ima oblik kvadra.



Slika 3.11: Volumen kamere perspektivne projekcije (lijevo) i ortografske projekcije (desno)

Fragmenti i rasterizacija trokuta

Kako bi prikazali trokut na ekranu, moramo ispuniti piksele koje taj trokut prekriva odgovarajućim bojama. Ovaj proces nazivamo rasterizacija. Na početku rasterizacije, površina trokuta dijeli se na dijelove koje nazivamo **fragmenti**. Svaki fragment predstavlja malo područje površine trokuta koje odgovara jednom pikselu na ekranu. Fragmenti prvo prolaze kroz niz testova, a ako ne prođe na nekom od testova, tada se fragment odbacuje. Na primjer, fragment se odbacuje ako se nalazi izvan volumena kamere. Ako fragment prođe na svim testovima, tada se njegova boja upisuje u odgovarajući piksel, ili se miješa sa već postojećom bojom piksela (kod prozirnih i poluprozirnih objekata).



Slika 3.12: Fragment je malo područje trokuta koje odgovara pikselu na ekranu

Međuspremnik dubine

Kada naiđemo na dva trokuta unutar scene koja se preklapaju, tada moramo odrediti koji dio kojeg trokuta će se iscrtati, odnosno koji dio kojeg trokuta je bliže kameri i samim time zaklanja udaljeniji trokut.

Kako bi uspješno proveli navedeni test, sustavi za iscrtavanje koriste međuspremnik dubine. Međuspremnik dubine je matrica koja za svaki piksel na ekranu čuva jedan decimalni broj. Naime, svaki fragment ima z koordinatu koja mjeri njegovu dubinu unutar ekrana, odnosno udaljenost od žarišne točke kamere. Kada se boja fragmenta upiše u odgovarajući piksel, njegova dubina upisuje se u međuspremnik dubine, na mjesto koje odgovara tom pikselu. Kada se neki novi fragment iz drugog trokuta želi iscrtati na ekran, prvo se uspoređuje njegova dubina sa dubinom zapisanom u međuspremniku dubine. Ako je taj novi fragment bliže kameri, odnosno ima manju dubinu, tada se njegova boja upisuje u odgovarajući piksel. Inače se taj novi fragment odbacuje.

2D iscrtavanje

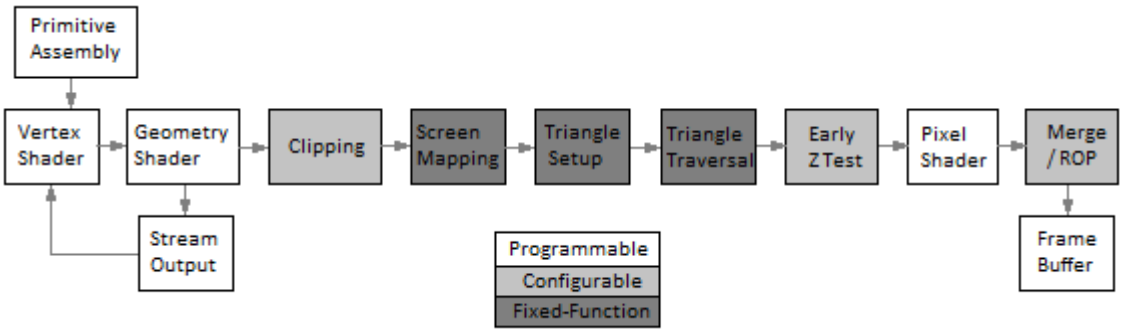
2D iscrtavanje može se shvatiti kao poseban slučaj 3D iscrtavanja. Naime, umjesto 3D modela i njihovih mreža trokuta, možemo koristiti dvodimenzionalne pravokutnike koje čine po dva trokuta. Svi pravokutnici nalaze se na dubini 0, i uvijek su okrenuti tako da gledaju prema kameri. Detaljnije o 2D iscrtavanju može se vidjeti u knjizi [4].

3.5 Grafički procesor i shader programi

Na samom početku razvoja industrije igara, iscrtavanje je odrađivao procesor. Porastom popularnosti igara dolazi do pojave prvih grafičkih kartica. Na početku njihove funkcije bile su ugrađene u samu karticu, a mogle su se konfigurirati u manjoj mjeri. Tek kasnije pojedine faze u procesu iscrtavanja postale su programibilne. Postalo je moguće pisati **shader programe** koji kontroliraju na koji način se procesiraju vrhovi i fragmenti. Međutim, grafički procesor nije potpuno programibilan, niti bi to trebao biti. Visok stupanj paralelnosti i velike performanse postižu se upravo fiksiranjem nekih faza u procesu iscrtavanja i nemogućnošću njihovog modificiranja. U ovom poglavlju reći ćemo nešto o programibilnim fazama procesa iscrtavanja, te o shader programima pomoću kojih programiramo te faze.

Faze iscrtavanja grafičkog procesora

Većina današnjih grafičkih procesora dijele proces iscrtavanja u nekoliko određenih faza na način da su izlazni podaci jedne faze zapravo ulazni podaci u sljedeću fazu. Neke od tih faza potpuno su fiksirane i ugrađene u grafički procesor, dakle ne mogu se modificirati niti na jedan način. Neke od faza su također fiksirane, ali se mogu donekle konfigurirati, dok su neke faze programabilne i mogu se potpuno prilagoditi zahtjevima.



Slika 3.13: Proces iscrtavanja geometrije iz perspektive grafičkog procesora. Baze bijele boje su programabilne, svijetlo sive se samo mogu konfigurirati, dok su tamno sive potpuno fiksirane

Na slici3.13 možemo vidjeti sljedeće tri programibilne faze: *vertex shader*, *geometry shader* i *pixel shader*. One su nam bitne jer nam omogućavaju da preko njih oblikujemo konačan izgled 3D modela.

Vertex shader je faza odgovorna za transformacije, osvijetljenje i određivanje boje pojedinih vrhova mreže trokuta. Ulazni podatak za ovu fazu je jedan vrh zajedno sa svojim svojstvima. On se zatim tranformira iz prostora svijeta u prostor kamere, te mu se pomoću podataka o osvijetljenju i koordinata teksture računa boja vrha. Izlazni podatak ove faze je potpuno osvijetljen i transformiran vrh.

Geometry shader je faza u kojoj modificamo postojeće ili stvaramo nove geometrijske oblike, kao što su trokuti, linije ili točke. Na primjer, možemo od točke napraviti pravokutnik koji se sastoji od dva trokuta. Najčešće se koristi za subdiviziju mreže trokuta, simulacije tkanine, simulacije krzna, itd.

Pixel shader je faza pri samom kraju procesa iscrtavanja, a njezin posao je procesiranje fragmenata. U ovoj fazi koriste se teksture, izračunavanja osvijetljenja i sve ostalo potrebno za određivanje boje fragmenta. Fragment je moguće i odbaciti, ako je, na primjer, potpuno

proziran. Ulazni podaci u ovu fazu su atributi fragmenta koji su interpolirani iz atributa pripadnih vrhova. Izlazni podatak je vektor koji predstavlja željenu boju fragmenta.

Shader programi

Shader programi najčešće se pišu u programskim jezicima sličnima programskom jeziku C, kao što su Cg (C for graphics), HLSL (High-Level Shading Language) ili GLSL (OpenGL Shading Language). Sva tri tipa shader programa (vertex, geometry i pixel) imaju otprilike isti set instrukcija i otprilike iste mogućnosti.
Grafički procesor pristup radnoj memoriji drži pod strogom kontrolom, iz istog razloga zašto neke faze u procesu iscrtavanja ne možemo modificirati. Shader programi memoriji pristupaju na dva načina: kroz registre grafičkog procesora i kroz teksture. Registri grafičkog procesora obično su 128-bitni, odnosno sastoje se od četiri 32-bitna dijela. Na taj način jedan registar može sadržavati podatke o vektoru sa četiri elementa ili podatke o boji u RGBA formatu, gdje svaka komponenta odgovara 32-bitnom decimalnom broju.

Razlikujemo četiri tipa registara:

- Ulazni registri su registri koji sadrže podatke o atributima vrhova. U *pixel shader* programu, ulazni registri sadrže interpolirane atribute vrhova vezanih za određeni fragment.
- Konstantni registri također djeluju kao ulazni registri, ali sadrže podatke koji nisu dostupni kao atributi vrhova. Ti registri obično sadrže model-u-kamera matricu i parametre osvijetljenja.
- Privremene registre koristi sami shader program. U njih se pospremaju trenutni rezultati izračunavanja.
- Izlazni registri služe kao jedini oblik izlaza shader programa. Kod *vertex shader* programa, izlazni registri mogu sadržavati transformiranu poziciju vrha, boju vrha ili koordinate teksture. Izlazni registar *pixel shader* programa sadrži konačnu boju fragmenta koji se trenutno obrađuje.

Ovdje možemo vidjeti primjer jednostavnog *pixel shadera* koji izračunava boju fragmenta koristeći pridruženu teksturu.

```
1 struct FragmentOut {
2   float4 color : COLOR;
3 };
4
5 FragmentOut myShader(float2 uv : TEXCOORD0, uniform sampler2D texture) {
6   FragmentOut out;
```

```
7   out.color = tex2D(texture, uv); // look up texel at (u,v)
8   return out;
9 }
```

Uređeni par uv spremljen je u ulaznom registru, dok je referenca na teksturu `texture` spremljena u konstantni registar. Metoda `tex2D` vraća boju piksela teksture `texture` na koordinatama uv. Struktura out tipa `FragmentOut` posprema se u izlazni registar.

3.6 Osvijetljenje scene

Za iscrtavanje fotorealističnih scena potrebni su fizički točni algoritmi za globalno osvijetljenje. Postoji puno tehnika koje to omogućavaju, a ovdje ćemo ukratko opisati najčešće korištene tehnike u današnjoj industriji igara.

Osvijetljenje bazirano na slikama

Velik broj naprednih tehnika za osvijetljenje koriste podatke iz slika koje su obično u obliku dvodimenzionalnih tekstura. Te algoritme nazivamo algoritmi osvijetljenja bazirani na slikama.

Mapa normala

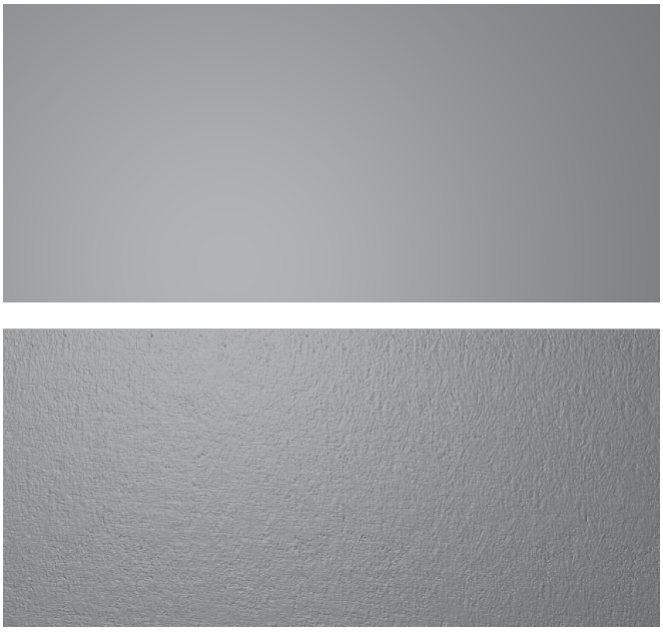
Mapa normala definira smjer vektora normale na svakoj točki teksture. Ovo omogućava 3D modeleru da pokretaču igre prenese visoko kvalitetan opis oblika površine 3D modela, bez potrebe za velikim brojem vrhova i trokuta u mreži trokuta (koji bi bili potrebni jer se vektori normale mogu spremati u vrhove). Koristeći mapu normale, moguće je napraviti da samo jedan ravan trokut izgleda kao da je sastavljen od više milijuna sitnih trokuta.

Mapa refleksije

Jačina refleksije svijetlosti od površine 3D modela ovisi o međusobnim kutevima pod kojima se nalaze kamera, izvor svijetla i vektor normale površine. Mnoge površine nemaju jednak sjaj na cijeloj svojoj površini. Na primjer, mokre površine su većeg sjaja nego suhe površine istog objekta. Detaljne informacije o sjaju i refleksivnosti površine 3D modela možemo zakodirati u teksturu koju nazivamo mapa refleksije

Mapa okoliša

Mapa okoliša izgleda kao panoramska slika okoliša slikana sa položaja nekog objekta unutar scene, a pokriva punih 360 stupnjeva horizontalno, i 180 ili 360 stupnjeva vertikalno. Mapa okoliša obično se koristi za efikasno iscrtavanje refleksija okoliša.



Slika 3.14: Prikaz korištenja mape normala na potpuno ravnoj površini

Trodimenzionalne teksture

Moderne grafičke kartice također imaju podršku za trodimezionalne teksture. 3D tekstura može se zamisliti kao niz 2D tekstura naslaganih jedna na drugu. 3D teksture mogu biti korisne kod opisivanja volumena prozirnih objekata. Na primjer, prozirni objekt možemo presjeći proizvoljnom ravninom, a objekt će i dalje izgledati ispravno duž presjeka, zato što je tekstura definirana kroz cijeli volumen objekta.

Globalno osvijetljenje

Globalno osvijetljenje obuhvaća cijelu klasu algoritama za osvijetljenje koji u obzir uzimaju utjecaj svijetla na razne objekte unutar scene, kroz cijeli put svijetlosti od izvora do virtualne kamere. Globalno osvijetljenje brine o efektima kao što su sjene do kojih dolazi kada jedan objekt zaklanja drugi, refleksije, i sve ostale pojave kod kojih boja jednog objekta utječe na izgled drugog objekta.

Iscrtavanje sjena

Do pojave sjena dolazi kada se neka površina nađe na putanji svijetlosti. Točkasti izvori svijetlosti čiji modeli se koriste u video igrama produciraju sjene oštrih rubova. Međutim,

u stvarnosti sjene imaju mekane, mutne rubove, čije iscrtavanje povećava kompleksnost modela.

Kod iscrtavanja sjena, objekte unutar scene dijelimo na tri kategorije: objekti koji bacaju sjenu, objekti koji trebaju primiti sjenu, i ostali objekti koji ne sudjeluju u iscrtavanju. Za svaki pojedini izvor svijetlosti unutar scene može se odrediti hoće li generirati sjene ili ne. Ova optimizacija je bitna jer smanjuje broj mogućnosti koje moraju biti procesirane kako bi se iscrtale sjene unutar scene.

Ambijentalna okluzija

Ambijentalna okluzija je tehnika modeliranja posebne vrste mekanih sjena koja nastaje kada je objekt osvijetljen samo ambijentalnim svijetlom. Ambijentalna okluzija opisuje koliko je svaka točka na površini modela općenito dostupna za osvijetljenje. Na primjer, unutrašnjost cijevi je manje dostupna za osvijetljenje nego vanjska površina cijevi. Kad bi cijev bila smještena u vanjski svijet za vrijeme oblačnog vremena, unutrašnjost cijevi bila bi puno manje osvijetljena nego njezin vanjski dio.

3.7 Vizualni efekti

Do sada smo opisivali proces iscrtavanja trodimenzionalnih objekata. Međutim, osim 3D objekata ponekad želimo iscrtavati dvodimenzionalne objekte kao što su sustavi čestica, naljepnice (mali objekti kojima reprezentiramo rupe od metaka, pukotine i druge detalje površine), kosa i krzno, kiša, snijeg ili voda. Osim toga, možda želimo neki efekt upotrijebiti preko cijelog ekrana,kao što su zamućenje dubine područja, zamućenje pokreta, efekti nad bojama i drugi. Na kraju, izbornici unutar igre obično su realizirani iscrtavanjem teksta preko već iscrtane trodimenzionalne scene.

Sustavi čestica

Sustavi čestica zaduženi su za iscrtavanje objekata kao što su dim, vatra, iskre ili oblaci. Glavna svojstva koja razlikuju sustave čestica od ostale geometrije u sceni su:

- sastavljeni su od velikog broja relativno malih objekata, najčešće jednostavnih četverokuta koji su sastavljeni od po dva trokuta.
- obično gledaju direktno prema kameri, što znači da pokretač igre mora osigurati da vektor normale svakog četverokuta uvijek gleda u smjeru žarišne točke kamere.
- materijali su im gotovo uvijek poluprozirni.

- čestice su animirane na mnoštvo različitih načina. Njihove pozicije, rotacije, veličine, koordinate tekstura i ostali parametri variraju između svakog koraka iscrtavanja.
- čestice se obično kontinuirano stvaraju i brišu. Odašiljač čestica zadužen je za stvaranje čestica u nekoj unaprijed određenoj brzini, te za brisanje tih čestica u pogodnom trenutku. Čestice se obično brišu kada dotaknu neku određenu ravninu u prostoru, ili kada prođe neko određeno vrijeme od njihovog stvaranja.

Čestice se mogu iscrtavati korištenjem regularnog iscrtavanja geometrije mreže trokuta, ali zbog jedinstvenih prethodno opisanih svojstava, gotovo uvijek se implementira poseban sustav za iscrtavanje čestica.

Naljepnice

Naljepnice se najčešće koriste kada želimo dinamički modificirati vizualni izgled površine. Na primjer, to mogu biti otisci stopala ili rupe od metaka.

Naljepnica se obično modelira kao pravokutna površina koja se projicira duž polupravca unutar scene. Tu projekciju možemo zamisliti kao pravokutnu prizmu unutar 3D prostora. Ona površina koju prizma prvu dotakne postaje površina naljepnice. Na tom mjestu stvara se četverokut sa određenom teksturom koji predstavlja naljepnicu.

Izbornici

Izbornici unutar igre i korisničko sučelje iscrtavaju se preko trodimenzionalne scene koja se nalazi u pozadini. Veći dio izbornika čini tekst, pa se zbog toga razvija sustav za iscrtavanje teksta. Taj sustav mora biti u mogućnosti iscrtati nizove znakova, a za to koristi posebnu teksturu koja u sebi sadrži sve moguće znakove. Postoji i dodatna datoteka koja sadrži podatke o pozicijama znakova unutar teksture.

Efekti cijelog ekrana

Efekti koji obuhvaćaju cijeli ekran djeluju na već iscrtani prikaz scene kako bi se dodatno povećao realizam scene ili stilizirani izgled scene. Ovi efekti obično su implementirani na način da se sadržaj ekrana proslijedi pixel shader programu koji zatim primjenjuje željeni efekt. Primjeri efekata cijelog ekrana su:

- Zamućenje pokreta (engl. *motion blur*). Za postizanje ovog efekta koriste se vektori brzine, pomoću kojih se selektivno zamućuju dijelovi ekrana.

- Zamućenje dubine polja. Za ovaj efekt koristi se međuspremnik dubine pomoću kojeg se za svaki piksel ekrana određuje količina zamućenja.
- Efekti nad bojama. Boje piksela mogu se modificirati na velik broj načina. Na primjer, sve boje osim crvene možemo modificirati u nijanse sive, pa će se svi objekti scene koji nisu crveni iscrtati u nijansama sive boje.

Poglavlje 4

Fizika u igrama

U stvarnom svijetu čvrsti objekti sami po sebi ne mogu prolaziti jedan kroz drugog. Međutim, u virtualnom svijetu, objekti ne rade ništa sve dok im mi to ne kažemo. Sustav za otkrivanje kolizija jedan je od najbitnijih sustava svakog pokretača igre, a njegova uloga je osigurati da objekti ne prolaze jedan kroz drugog.

Sustav za otkrivanje kolizija najčešće je integriran sa većim sustavom, sustavom za fiziku. Fizika je široko područje, a u kontekstu pokretača igre pod pojmom fizika obično mislimo na simulaciju dinamike krutih tijela. Kruto tijelo je idealiziran, beskonačno krut i nedeformirajući čvrsti objekt. Pojam dinamike referira na proces određivanja kako se kruta tijela kreću i djeluju jedno na drugo tijekom nekog vremena i pod utjecajem sila. Simulacija dinamike krutih tijela omogućuje nam prikaz složenih kretnji objekta koje su u konačnici mnogo prirodnije i detaljnije nego što bi bile upotrebom animacija.

Simulacija dinamike krutih tijela koristi sustav za otkrivanje kolizija kako bi ispravno simulirala razna fizička ponašanja objekata kao što su odbijanje objekata, klizanje objekata niz kosinu pod utjecajem sile trenja, valjanje objekata i dolazak objekta u stanje mirovanja.

U ovom poglavlju prvo ćemo objasniti zašto uopće trebamo fiziku u igrama i što nam ona donosi. Zatim ćemo reći nešto više o otkrivanju kolizija, te o samoj simulaciji dinamike krutog tijela.

4.1 Zašto trebamo fiziku u igrama

Većina današnjih pokretača igre sadrži neki oblik simulacije fizike. Neke fizičke efekte igrači jednostavno očekuju u igrama, dok neki složeniji efekti kao simulacije užeta, tkanine, kose i sličnog odvajaju igru od konkurencije. U zadnje vrijeme razvija se sve više složenih efekata kao što su simulacije fluida ili simulacije deformirajućih objekata.

Navedimo neke od stvari koje možemo imati implementacijom sustava za fiziku:

- Otkrivanje kolizije između dinamičkih objekata i statične geometrije unutar scene.
- Simulacija krutih tijela pod utjecajem gravitacije i ostalih sila.
- Sustav opruge.
- Destruktivne zgrade i ostale strukture.
- Otkrivanje ulaska objekta unutar nekog unaprijed definiranog volumena unutar scene.
- Kompleksni strojevi kao što su dizalice.
- Zamke, na primjer lavina gromada stijena.
- Vozila sa realističnim ovjesom.
- Njihajući objekti.
- Simulacija tkanine.
- Simulacija površine vode.

Implementacija fizike u igru može imati razne utjecaje na projekt i na *gameplay* igre. Navest ćemo nekoliko primjera kroz različite discipline razvoja igre.

Utjecaj na dizajn igre

Nepredvidivost je jedna od negativnih posljedica fizike u igrama. Za razliku od animacija, utjecaj fizike, tj. gravitacije i ostalih sila donosi mnogo novih nepredvidivih ponašanja objekata.

U stvarnom svijetu, zakoni fizike su fiksirani. Međutim, u igrama možemo mijenjati jačinu gravitacije ili druge parametre. Na taj način možemo dodatno prilagoditi fiziku samoj igri.

Utjecaj na implementaciju

Sustav za fiziku jedan je od kompliciranijih sustava pokretača igre, tako da zahtjeva određeno vrijeme za samu implementaciju. Osim toga, utječe još i na sljedeće:

- Umjetna inteligencija možda neće raditi ispravno u prisutnosti fizički simuliranih objekata. Na primjer, pronalaženje najkraćeg puta od točke A do točke B možda neće ispravno raditi jer se neki fizički simulirani objekt našao na putu.

- Uvođenjem, na primjer, destruktivnih građevina u igru može utjecati na sustav za iscrtavanje scene. Ako građevina nije destruktivna, tada uvijek ima fiksne koordinate, pa je lako odrediti kada zgrada izađe iz volumena kamere, tj. kada se treba iscrtavati, a kada ne. Osim toga, sjena koju statična građevina baca uvijek je ista. Ako je građevina destruktivna, tada ne vrijedi prethodno navedeno.
- Kod mrežnih igara sa više igrača, ista fizička simulacija može se simulirati na različite načine na različitim računalima. Zbog toga potrebno je sve fizičke simulacije koje utječu na *gameplay* simulirati na serveru, a zatim ih replicirati na računalima igrača.

Ostali utjecaji

Za neke objekte potrebno je napraviti više različitih verzija. Na primjer, originalnu verziju objekta i oštećenu verziju zgrade ili automobila. Osim toga, potrebno je odrediti mase, sile trenja i ostale parametre za svaki objekt.

4.2 Otkrivanje sudara

Primarna zadaća sustava za otkrivanje sudara je odrediti je li između ikoja dva objekta u virtualnom svijetu došlo do kontakta. Da bi to bilo moguće, prvo moramo svaki objekt u sceni reprezentirati jednim ili više geometrijskih oblika. Ti oblici su obično vrlo jednostavni, kao što je sfera, kocka ili kapsula. Sustav za otkrivanje sudara zapravo određuje sijeku li se neka dva oblika u nekom trenutku. Možemo reći da je sustav za otkrivanje sudara zapravo tester geometrijskih presjeka.

Sustav za otkrivanje sudara ne daje samo odgovor ”da” ili ”ne” na pitanje o presjeku oblika, nego i puno više. On također pruža informacije o kontaktima svakog presjeka. Te informacije zatim se koriste za spriječavanje neželjenih vizualnih anomalija, kao na primjer ulazak jednog objekta u drugi objekt. Ovo se obično postiže tako da se svi takvi objekti odmaknu jedan od drugog, prije nego što se sljedeća sličica iscrta na ekran. Osim toga, pomoću sudara određujemo i kada je objekt došao u stanje mirovanja.

Reprezentacija objekata geometrijskim oblikom

Ako želimo nekom objektu omogućiti sudaranje sa ostalim objektima, moramo ga reprezentirati nekim **geometrijskim oblikom**. Taj geometrijski oblik opisuje oblik objekta, njegovu poziciju i rotaciju u virtualnom svijetu. Ova reprezentacija objekta potpuno je odvojena od njegove *gameplay* reprezentacije (programski kod i ostali podaci koji definiraju ulogu i ponašanje objekta u igru) i od njegove vizualne reprezentacije (koja može biti

u obliku mreže trokuta, sustava čestica ili nečeg drugog).

Općenito, preferiramo oblike koji su geometrijski i matematički jednostavni. Na primjer, kamen može biti reprezentiran kao sfera, vrata od auta mogu biti reprezentirana kao kvadar, a čovjek može biti reprezentiran kao kolekcija povezanih kapsula. Korištenje kompleksnijih oblika preporučeno je samo u slučajevima kada ovi jednostavni oblici nisu primjereni i uzrokuju nepoželjno ponašanje. Na slici 4.1 možemo vidjeti primjer reprezentacije dva objekta geometrijskim oblicima koji aproksimiraju oblike tih objekata za potrebe otkrivanja sudara.

Slika 4.1: Jednostavni geometrijski oblici često se koriste za aproksimaciju volumena objekata u igri

Svijet sudara

Sustav za otkrivanje sudara obično sprema sve potrebne podatke u jednu singleton strukturu podataka poznatu kao **svijet sudara** (engl. *collision world*). Svijet sudara je kompletna reprezentacija virtualnog svijeta igre namjenjena za korištenje od strane sustava za otkrivanje sudara.

Ovakva struktura podataka i čuvanje svih podataka u privatnom objektu ima puno prednosti nad čuvanjem podataka o reprezentacijama unutar samih objekata igre. Jedan od razloga je što neke objekte ne želimo reprezentirati unutar svijeta sudara, jer ne želimo da se sudaraju jedni sa drugima. Osim toga, podatke svijeta sudara možemo smjestiti blizu u memoriji, što povećava performanse.

Svijet sudara često je zajednički sustavu za otkrivanje sudara i sustavu za simulaciju dinamike krutog tijela, koji još nazivamo sustav fizike. Svako kruto tijelo sustava fizike

povezano je sa jednim objektom sustava za otkrivanje sudara. Baš zbog te povezanosti, nije rijetkost da sustav fizike zapravo upravlja sustavom za otkrivanje sudara, pozivajući ga da izvrši testove sudara barem jednom, a često i više puta u jednom koraku simulacije.

Primitivni oblici za reprezantaciju objekata

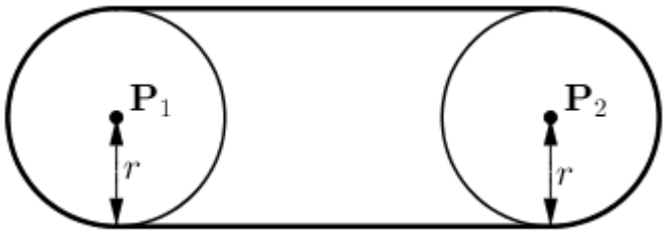
Sada ćemo navesti oblike za reprezentaciju objekata koji su obično podržani u sustavima za otkrivanje sudara. Taj skup oblika je relativno malen, ali pomoću njega možemo izgraditi kompleksnije oblike. Skup ovih jednostavnih oblika zovemo primitivni oblici.

Sfera

Najjednostavniji trodimenzionalan oblik je sfera. Ona je također i najefikasniji primitivni oblik za otkrivanje sudara. Reprezentirana je središtem i radijusom. Ova dva podatka mogu se zapakirati u vektor sa četiri komponente, a takvi formati imaju veliku efikasnost u današnjim matematičkim bibliotekama.

Kapsula

Kapsula je oblik koji se sastoji od jednog cilindra i dvije polusfere na njegovim krajevima. Možemo ju shvatiti kao oblik koji nastaje kada sferu u prostoru translatiramo od točke A do točke B. Kapsula je reprezentirana sa dvije točke i radijusom. One su druge po efikasnosti, pa se često koriste za reprezentaciju objekata koji su otprilike cilindrični, kao što su udovi ljudskog tijela.



Slika 4.2: Kapsula je reprezentirana sa dvije točke i radijusom

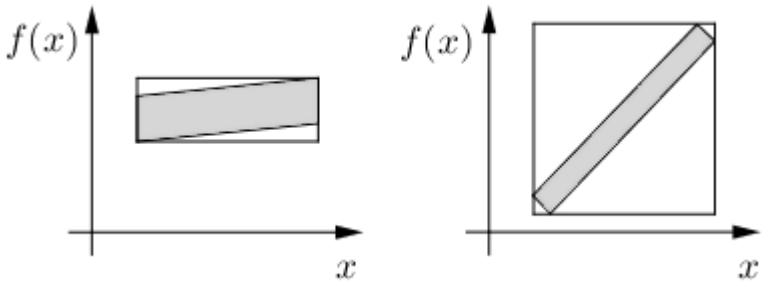
Kvadar poravnat sa koordinatnim osima

Kvadar poravnat sa koordinatnim osima (engl. *axis-aligned bounding box (AABB)*) je pravokutni volumen čije strane su paralelne sa koodinatnim osima. Naravno, kvadar poravnat

sa osima jednog koordinatnog sustava ne mora biti poravnat sa osima nekog drugog koordinatnog sustava. Zato o takvim oblicima govorimo samo u kontekstu određenog koordinatnog sustava sa kojim su poravnati.

Ovaj oblik može se reprezentirati sa dvije točke. Prva točka sadržava minimalne koordinate oblika na svakoj od tri koordinatne osi, a druga točka sadržava maksimalne koordinate oblika na svakoj od tri koordinatne osi.

Glavna prednost ovog oblika je što se vrlo efikasno može testirati na sudare sa ostalim oblicima iste vrste. Najveći nedostatak je to što ovaj oblik mora ostati poravnat sa koordinatnim osima cijelo vrijeme, što zahtjeva dodatno izračunavanje svaki puta kada se objekt rotira. Osim toga, čak i ako je objekt kojeg reprezentiramo približno pravokutan, ovaj oblik može biti loša aproksimacija ako je objekt rotiran pod određenim kutevima.



Slika 4.3: Primjeri kvadra poravnatog sa koordinatnim osima. Aproksimacija je to bolja što je sami objekt više poravnat sa koordinatnim osima

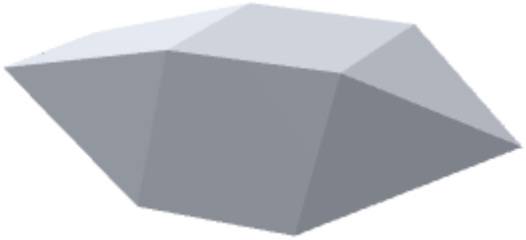
Orijentirani kvadar

Ako omogućimo rotaciju prethodno opisanog oblika kvadra, tada dobivamo orijentirani kvadar. On je reprezentiran sa tri vektorom polu-dimenzija (polu-širinom, polu-dubinom i polu-visinom), i sa matricom transformacije koja pozicionira centar kvadra u prostor svijeta i rotira ga u odgovarajuću orijentaciju. Orijentirani kvadar je često korišteni oblik jer bolje aproksimiraju pravokutne objekte, a i dalje su relativno jednostavni.

Proizvoljni konveksni oblici

Većina sustava za otkrivanje sudara dopušta upotrebu proizvoljnih konveksnih oblika kontruiranih u nekom od alata za izradu 3D modela. 3D umjetnici izrađuju takve oblike pomoću poligona, obično trokuta ili četverokuta. Ako oblik prođe test konveksnosti koji

potvrđuje da je oblik stvarno konveksan, tada se njegovi trokuti konvertiraju u skup od k ravnina reprezentiran sa k jednadžbi ravnine, ili sa k točaka i k vektora normale. Otkrivanje sudara konveksnih oblika manje je efikasno nego kod prethodno opisanih oblika, ali i dalje je dovoljno efikasno za korištenje u igrama, baš zbog svoje konveksnosti.



Slika 4.4: Konveksan oblik

Proizvoljna mreža trokuta

Neki sustavi za otkrivanje sudara imaju podršku za potpuno proizvoljne, nekonveksne mreže trokuta. One se koriste za modeliranje kompleksne statične geometrije, kao što je teren. Ovo je najskuplji oblik otkrivanja sudara. Kod njega sustav za otkrivanje sudara mora testirati svaki trokut mreže trokuta, pa se preporučuje koristiti što manji broj ovakvih oblika.

Složeni oblici

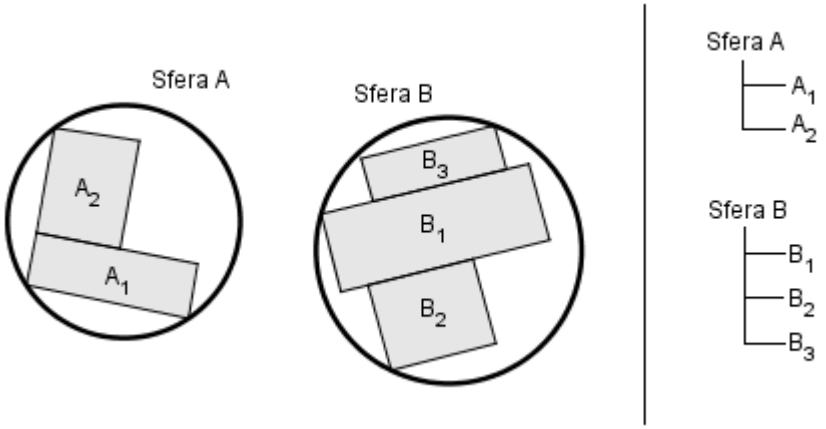
Neki objekti ne mogu biti reprezentirani samo jednim oblikom, ali mogu se reprezentirati skupom oblika. Na primjer, stolac sa slike 4.5 reprezentiran je pomoću dva kvadra, jedan za donji dio stolca, koji obuhvaća sjedalo i četiri noge stolca, i jedan za naslon stolca.

Neki sustavi za otkrivanje sudara koriste složene oblike za optimizaciju procesa pronalaska sudara. Naime, moguće je dvije grupe oblika okružiti sa po jednom sferom za svaku grupu. Ako se te dvije sfere ne sijeku, tada nije potrebno međusobno provjeravati sudaraju li se objekti iz jedne grupe sa objektima iz druge grupe.

Testiranje sudara

Za pronalaženje oblika koji se sijeku i za otkrivanje sudara, sustav za otkrivanje sudara koristi se analitičkom geometrijom. Ovdje ćemo navesti neke kombinacije oblika koji se mogu sudariti, te navesti neke primjere. Važno je primjetiti da broj kombinacija raste kvadratno sa brojem različitih oblika, pa većina sustava za otkrivanje sudara pokušava smaniti broj različitih primitivnih oblika.

Slika 4.5: Reprezentacija stolca korištenjem dva kvadra



Slika 4.6: Grupiranje oblika

Točka i sfera

Jednostavno je dati odgovor na pitanje leži li točka unutar sfere. Potrebno je samo usporediti duljinu vektora s kojeg određuju točka p i centar sfere c, sa radijusom sfere r. Ako je ta duljina veća od radijusa, točka nije u sferi. Inače se točka nalazi unutar sfere.

$$s = c - p; \tag{4.1}$$

$$\text{ako } |s| \leq r, \text{onda je tocka } p \text{ unutar sfere} \tag{4.2}$$

Sfera i Sfera

Otkrivanje sijeku li se dvije sfere gotovo je iste težine kao otkrivanje leži li točka unutar sfere. U ovom slučaju prvo računamo vektor s koji spaja centre dviju sfera. Zatim ga uspoređujemo sa sumom radijusa tih dviju sfera. Ako je duljina vektora manja od sume radijusa, tada se sfere sijeku. Inače se ne sijeku.

$$s = c_1 - c_2; \tag{4.3}$$

$$\text{ako } |s| \leq (r_1 + r_2), \text{onda se sfere sijeku} \tag{4.4}$$

Često se koristi sljedeća optimizacija, kod koje umjesto računanja korijena prilikom računanja duljine vektora koristimo kvadrat duljine vektora. Tada prethodna jednadžba postaje:

$$s = c_1 - c_2; \tag{4.5}$$

$$|s|^2 = s \cdot s; \tag{4.6}$$

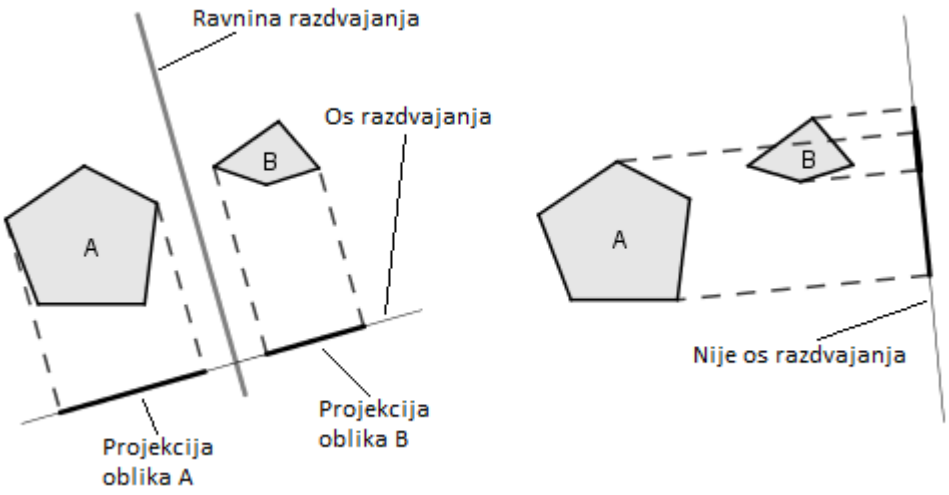
$$\text{ako } |s|^2 \leq (r_1 + r_2)^2, \text{ onda se sfere sijeku} \tag{4.7}$$

Teorem o osi razdvajanja

Mnogi sustavi za otkrivanje sudara koriste teorem o osi razdvajanja. On kaže da ako postoji os na kojoj se projekcije dva konveksna oblika ne sijeku, tada se niti ti oblici ne sijeku. Ako takva os ne postoji, a oblici su konveksni, tada se oblici sijeku. Drugim riječima, ako postoji ravnina koja odvaja dva konveksna objekta na način da je jedan objekt u potpunosti sa jedne strane, a drugi u potpunosti sa druge strane ravnine, tada se oni ne sijeku. Teorem je najlakše vizualizirati u dvodimenzionalnom prostoru. Više o ovom teoremu i njegovom korištenju u otkrivanju sudara nalazi se na poveznici [5].

Dva kvadra poravnata sa koordinatnim osima

Za testiranje presjeka između dva kvadra koja ćemo nazvati A i B, promatramo minimalne i maksimalne koordinate za svaku od tri koordinatne osi zasebno. Na primjer, imamo dva intervala duž osi x $[x^A_{min}, x^A_{max}]$ i $[x^B_{min}, x^B_{max}]$, te odgovarajuće intervale za osi y i z. Ako se ti intervali preklapaju duž sve tri osi, tada se ova dva kvadra sijeku. Inače se ne sijeku.

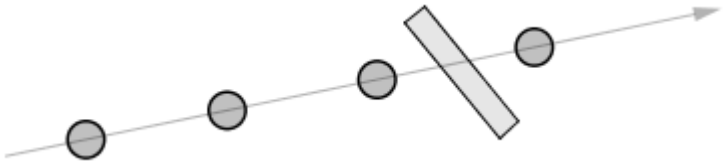


Slika 4.7: Projekcija oblika na os razdvajanja (lijevo) i projekcija na os koja ne razdvaja oblike (desno)

Otkrivanje sudara objekata u pokretu

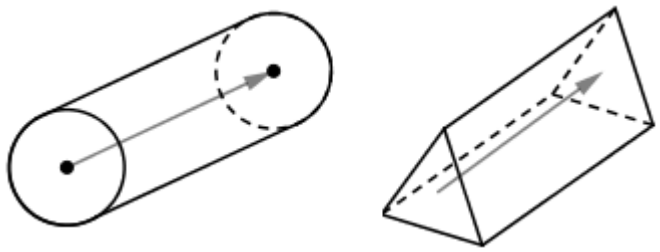
Do sada smo promatrali samo statične objekte i presjeke statičnih objekata. Otkrivanje sudara kod objekata u pokretu nešto je kompliciranije. Pokret se u igrama simulira u diskretnim vremenskim trenucima. Ideja koja se nameće je objekte smatrati statičnima u svakom pojedinom vremenskom trenutku i na njima provoditi testove. Većina sustava za otkrivanje sudara koristi upravo ovaj pristup.

Međutim, ovaj pristup ne radi za male objekte koji se kreću velikom brzinom. Ako imamo objekt koji se kreće toliko velikom brzinom da između dva vremenska trenutna pređe veću udaljenost nego što je sam velik, tada je moguće da ”preskoči” neki objekt. Ovaj problem poznat je pod nazivom tuneliranje, a primjer možemo vidjeti na slici 4.8.



Slika 4.8: Sudar malog objekta velike brzine nije uspješno detektiran

Ovaj problem može se riješiti na dva načina. Prvi način je da na sudar ne testiramo samo objekt u pokretu, nego oblik koji nastaje kada se oblik pomakne od točke A iz prošlog vremenskog trenutka do točke B u sadašnjem vremenskom trenutku. Primjer takvih oblika možemo vidjeti na slici 4.9.



Slika 4.9: Od sfere nastaje kapsula, dok od trokuta nastaje prizma

Drugi način je tehnika neprekidnog otkrivanja sudara (eng. continous collision detection). Cilj ove tehnije je pronaći najranije vrijeme sudara između dva objekta tijekom danog intervala vremena, te na taj način spriječiti tuneliranje.

Optimizacija performansi

Otkrivanje sudara je procesorski zahtjevno iz dva glavna razloga:

- Računanja potrebna za određivanje presjeka dva oblika su kompleksna sama po sebi.
- Većina scena u igrama sadrži velik broj objekata, a broj mogućnosti sudara i broj potrebnih testova na sudare brzo raste kako se povećava broj objekata u sceni.

Za otkrivanje sudara između n objekata, u najgorem slučaju moramo svaki par objekata testirati, što daje složenost $O(n^2)$. Međutim, postoje mnogi efikasniji algoritmi koji se primjenjuju u praksi.

Kratkoročna vremenska koherentnost

Jedna od čestih optimizacijskih tehnika je korištenje vremenske koherentnosti. Naime, pozicija i rotacija objekata je obično slična između dva vremenska koraka. Zbog toga često možemo izbjeći računanja nekih podataka, tako što ih pospremamo i čuvamo u memoriji tijekom više vremenskih koraka. Na taj način u nekom vremenskom koraku možemo koristiti podake izračunate u prethodnom vremenskom koraku.

Prostorna particija

Osnovna ideja prostornog grupiranja je smanjiti broj mogućnosti za sudar koje treba procesirati na način da se prostor podijeli na više manjih dijelova. Ako na efikasan način uspijemo odrediti da se par objekata ne nalazi u istom dijelu, tada znamo da se sigurno ne sijeku, pa nije potrebno provoditi detaljnije testove.

Široka faza, međufaza i uska faza

Neki sustavi koriste sljedeći pristup:

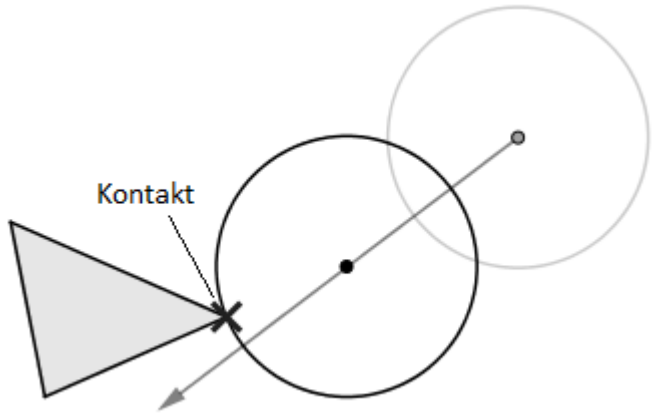
- Prvo, provodi se test pomoću kvadara poravnatih sa koordinatnim osima. Na ovaj način određuju se objekti koji se potencijalno sijeku. Ova faza naziva se široka faza.
- Drugo, složeni objekti se testiraju na sudare. Na primjer, ako se objekt sastoji od tri sfere, tada se može promatrati četvrta sfera koja okružuje sve tri sfere, i ona se testira na sudare. Ovu fazu nazivamo međufaza.
- Na kraju, individualni primitivni oblici se testiraju na sudare. Ovu fazu zovemo uska faza.

Upiti o sudarima

Još jedna od zadaća sustava za otkrivanje sudara je hipotetsko ispitivanje sudara. Neki od primjera su:

- Ako metak putuje iz igračevog oružja u određenom smjeru, koji objekt će pogoditi i hoće li uopće pogoditi neki objekt?
- Može li se vozilo kretati od točke A do točke B bez da se sudari sa nekim objektom na putu?
- Odrediti sve neprijatelje koji se od igrača nalaze unutar određenog radijusa.

Ovakve operacije nazivamo upiti o sudarima (engl. *collision queries*). Najčešći oblik upita je određivanje hoće li se hipotetski objekt sudariti sa nekim drugim objektom ako ga pomićemo duž polupravca ili segmenta pravca. Ovi upiti ne utječu na ostale objekte u svijetu sudara, jer zapravo nisu dio njega. Na slici 4.10 možemo vidjeti primjer upita o sudaru gdje hipotetsku sferu pomičemo u smjeru d.



Slika 4.10: Upit o sudaru sfere duž pravca u smjeru vektora d

4.3 Dinamika krutog tijela

Tijelo u fizici je skup čestica tvari koje su povezane tako da se zajedno gibaju. Tijelo može biti kruto, plastično ili elastično, a nas trenutno zanimaju kruta tijela. **Kruto tijelo** je idealno tijelo u kojem je međusobni položaj čestica konstantan, tako da se oblik ne može promijeniti, a da pritom ne dođe do destrukcije tijela. Ovakva idealna tijela u prirodi ne postoje, ali ovaj model se svejedno koristi jer dovoljno dobro aproksimira fizikalna svojstva tijela.

Mnogi pokretači igre sadrže sustav za fiziku u svrhu simulacije pokreta objekata unutar virtualnog svijeta na fizički realističan način. Tehnički gledano, sustavi za fiziku zaduženi su za dio fizike koji nazivamo mehanika. Mehanika proučava načine na koje sile djeluju na objekte.

Međutim, nas najviše zanima dinamika objekata, odnosno njihovi pokreti tijekom vremena. Sustav za fiziku osigurava neka ograničenja nad krutim tijelima u pokretu. Najčešće od tih ograničenja je zabrana prolaska jednog objekta kroz drugi. Dakle, sustav za fiziku zadužen je za realistično rješavanje sudara objekata. Ovo je jedan od glavnih razloga uske povezanosti sustava za otkrivanje sudara i samog sustava za fiziku.

Osim ovog ograničenja, sustav za fiziku dozvoljava postavljanje raznih drugih ograničenja u svrhu definiranja realističnih interakcija između fizički simuliranih krutih objekata, kao što su njihala, klizači, opruge, kotači i drugi.

Kruta tijela sustava za fiziku obično nisu direktno povezana sa logičkim objektima koji tvore virtualni svijet. Sustav za fiziku prvo procesira sva kruta tijela, a zatim se promjene

u poziciji i rotaciji primjenjuju na odgovarajuće objekte virtualnog svijeta.

Linearna i kutna dinamika

Kruto tijelo bez ograničenja je ono kruto tijelo koje je moguće translatirati duž sve tri koordinatne osi i koje se može slobodno rotirati duž sve tri koordinatne osi. Kažemo da takvo kruto tijelo ima šest stupnjeva slobode.

Kretnje krutog tijela mogu se odvojiti u dvije potpune nezavisne komponente: linearnu i kutnu dinamiku. Linearna dinamika zadužena je za kretnje tijela kada ignoriramo sve efekte rotacije. Linearnom dinamikom zapravo opisujemo kretnje idealizirane točke mase, tj. mase tijela koje je beskonačno malo i ne može se rotirati. Kutna dinamika opisuje rotacijske kretnje tijela. Ovo svojstvo nezavisnosti omogućuje nam zasebno analiziranje linearne i kutne komponente kretnje tijela, što je vrlo korisno kod simuliranja ponašanja krutog tijela.

Korak simulacije

Ovdje ćemo opisati jedan korak simulacije sustava za fiziku. Svaki sustav za fiziku implementiran je na različiti način, ali ovo su zadaci zajednički svim implementacijama:

1. Primjenjuju se sile koje djeluju na objekte u sceni, tj. računa se njihov utjecaj kroz vrijeme [delta] t proteklo od prošlog koraka. Na taj način određuju se nova privremena pozicija i rotacija tijela.

2. Poziva se sustav za otkrivanje sudara kako bi se odredili novi sudari nastali pomicanjem tijela na nove privremene pozicije i rotacije. Svi dosadašnji kontakti između objekata su pospremljeni u memoriju, pa se dodatno provjerava koji od tih kontakata više nisu aktivni. Također se svi novi kontakti spremaju u memoriju.

3. Rješavaju se sudari, najčešće primjenom impulsa sile.

4. Primjenjuju se sva ograničenja vezana za tijelo.

Na kraju zadnjeg koraka, moguće je da su neka kruta tijela dodatno pomaknuta sa trenutnih pozicija određenih u prvom koraku. Ova promjena u poziciji možda uzrokuje nove sudare između objekata, ili možda narušava neka ograničenja. Zbog toga, koraci od 1 do 4 (ponekad samo od 2 do 4) se ponavljaju, sve dok se svi sudari nisu uspješno riješili, ograničenja su zadovoljena i nema novih sudara, ili dok nije postignut unaprijed određen broj iteracija. Ako je postignut unaprijed određen broj iteracija, tada ovaj korak simulacije jednostavno završava, u nadi da će se ostatak sudara riješiti u sljedećim koracima

simulacije. Ovo pomaže u povećanju performansi sustava, a cijena izračunavanja sudara amortizira se kroz više koraka. Međutim, ponekad može dovesti do neočekivanih i vizualno pogrešnih ponašanja ako su greške velike ili ako je vremenski korak prevelik.

Poglavlje 5

Ostali dijelovi pokretača igre

Kao što smo već vidjeli, pokretač igre sastoji se od mnogobrojnih podsustava. Sustavi za iscrtavanje i sustavi za fiziku obično su najveći podsustavi pokretača igre, ali postoji još velik broj podsustava koje pokretači igre moraju sadržavati.

U ovom poglavlju navest ćemo još bitne dijelove koje sadrži većina pokretača igre. Reći ćemo nešto o sustavu za upravljanje memorijom koji je potreban za efikasno pospremanje resursa igre u radnu memoriju računala. Zatim ćemo reći nešto o audio sustavu, odnosno o zvukovima unutar igre. Nakon toga objasniti ćemo pojam skriptnog jezika i navesti neke često korištene skriptne jezike, a na kraju ćemo reći nešto više o ulaznim uređajima i načinima na koje igrač ostvaruje interakciju sa video igrom.

5.1 Upravljanje memorijom

Resursi koje igre koriste obično su velike multimedijske datoteke, kao što su slike, glazbene teme ili fontovi teksta. Te datoteke zauzimaju puno memorije, a operacije nad njima, kao što je kopiranje, izvode se sporo. Ovo utječe na načine na koje ih koristimo u igrama, zato što pokušavamo smanjiti broj takvih sporih operacija na minimum.

Neke datoteke kao što su skripte koje opisuju svijet igre, sadržaji izbornika ili umjetna inteligencija također se smatraju resursima igre, kao i konfiguracijske datoteke koje sadrže korisnikove postavke kao što su rezolucija ekrana ili glasnoća glazbe. Svi ovi resursi obično se učitavaju iz datoteka na disku, a ponekad i iz radne memorije ili sa mreže.

Dakle, brzina izvođenja naše igre ne ovisi samo o brzini izvođenja algoritama, već i o načinu na koji igra upravlja memorijom. Memorija utječe na performanse na sljedeća dva načina:

1. Dinamičke alokacije memorije pomoću funkcije `malloc()` ili operatora `new` su jako

spore operacije. Performanse našeg koda možemo poboljšati izbjegavanjem korištenja dinamičke alokacije, ili korištenjem vlastitih alokatora memorije koji će smanjiti cijenu alokacije.

2. Na modernim procesorima, brzina izvođenja nekog programa često je određena i njegovim obrascima pristupa memoriji. Naime, podatke koji su smješteni u mali, uzastopni blok memorije procesor puno efikasnije koristi nego podatke raširene u velikom rasponu memorijskih adresa. Čak i najefikasniji algoritmi mogu postati iznenađujuće spori ako koriste podatke koji su neefikasno smješteni u memoriju.

Optimizacija dinamičke alokacije memorije

Dinamičke alokacije memorije pomoću funkcija `malloc()` i `free()` ili globalnih operatora `new` i `delete` su obično vrlo spore. Ovu vrstu alokacija nazivamo alokacije na hrpi (engl. *heap allocation*). Dva su glavna razloga visoke cijene alokacije na hrpi. Prvo, alokator hrpe je namijenjen za opću namjenu, dakle mora biti u mogućnosti alocirati bilo koju količinu memorije, od jednog bajta do jednog gigabajta. A drugo, na većini operacijskih sustava pozivi funkcijama `malloc()` ili `free()` zahtijevaju prebacivanje procesora iz korisničkog načina rada u povlašteni način rada (engl. *kernel mode*), zatim obradu zahtjeva, i na kraju povratak u korisnički način rada. Dakle, u razvoju igara općenito vrijedi pravilo: Alociranja memorije na hrpi svesti na minimum.

Naravno, dinamička alokacija memorije ne može se u potpunosti izbjeći, zbog čega većina pokretača igre implementiraju jedan ili više vlastitih alokatora. Vlastiti alokator može imati bolje performanse nego sistemski alokator hrpe iz dva razloga. Prvi je taj što vlastiti alokator zahtjeve za alokaciju može zadovoljiti korištenjem unaprijed alociranog bloka memorije, kojeg je unaprijed alocirao, ili je blok deklariran kao globalna varijabla. Ovaj način omogućava alokaciju u korisničkom načinu rada procesora i ne zahtjeva prebacivanja iz jednog načina u drugi. Drugi razlog je što korištenjem raznih pretpostavki o korištenju memorije možemo povećati efikasnost u odnosu na alokator hrpe opće namjene.

Popularni izbori implementacije alokatora su:

- Alokator baziran na stogu. Kad god igra učitava novu scenu, alocira se memorija za nju. Nakon što je scena učitana, vrlo je mala potreba za dodatnom dinamičkom alokacijom memorije. Na kraju, kada se scena treba obrisati iz memorije, svi podaci se brišu i memorija se oslobađa. Dakle, ima smisla koristiti ovakav tip alokatora zbog prirode video igara.

- *Pool* alokator. Ovaj pristup često se koristi i u softverskom inženjerstvu općenito. *Pool* alokator alocira puno malih blokova memorije koji su jednake veličine. Pokazivači na prazne blokove memorije smješteni su u vezanu listu. Kada alokator primi

zahtjev za alokacijom, jednostavno će vratiti prvi element u vezanoj listi pokazivača na prazne blokove memorije. Kada se alocirana memorija oslobodi, pokazivač se vraća u vezanu listu. Ove operacije imaju složenost $O(1)$.

- *Single-frame* alokator. Ovaj alokator služi za alokaciju privremene memorije potrebne samo za vrijeme iscrtavanja trenutne sličice. Podaci alocirani ovim alokatorom automatski se brišu nakon što je sličica potpuno iscrtana, dakle nije potrebna ručna dealokacija.

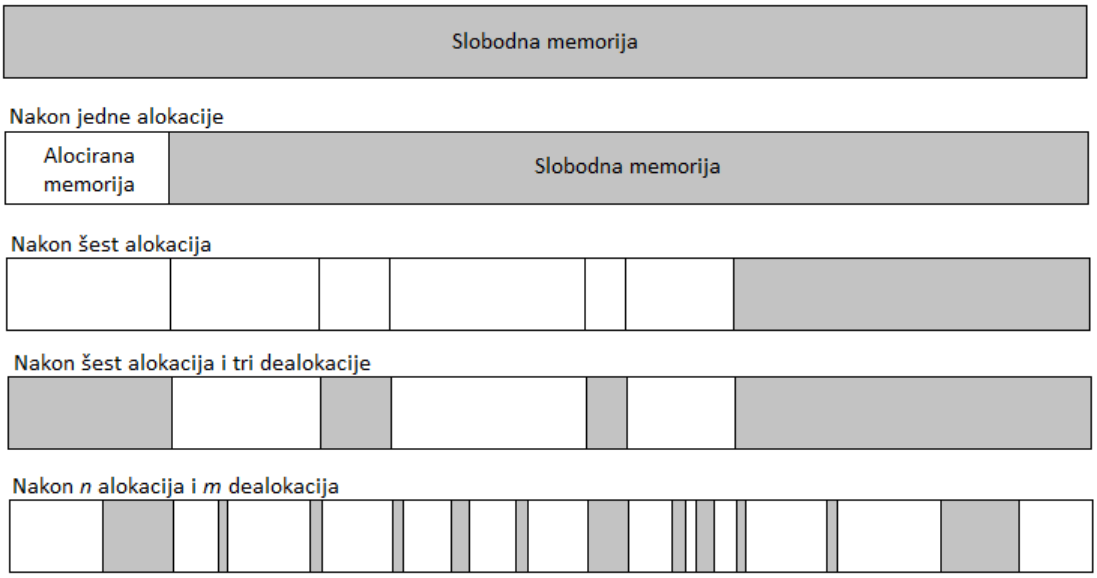
- Alokator dvostrukog međuspremnika (engl. *double-buffered alocator*). Ovaj alokator sličan je prošlom alokatoru, a dodatna prednost mu je što su podaci alocirani tijekom prošlog iscrtavanja sličice dostupni i tijekom sljedećeg iscrtavanja sličice. Za implementaciju ovog alokatora koriste se dva *single-frame* alokatora koja se međusobno izmjenjuju.

Fragmentacija memorije

Još jedan problem dinamičke alokacije memorije na hrpi je fragmentacija memorije tijekom vremena. U trenutku pokretanja programa, memorija je potpuno slobodna. Međutim, tijekom vremena dolazi do mnogobrojnih alokacija i dealokacija memorije, u različitim veličinama i na različitim mjestima u memoriji. Zbog toga se tijekom vremena pojavljuje mnogo slobodnih blokova memorije relativno male veličine, okruženih alociranom memorijom. Tada kažemo da je memorija fragmentirana.

Problem kod fragmentacije je što alokacija ne mora uspjeti, iako je ukupno gledajući dostupno dovoljno slobodne memorije. Na primjer, ako želimo alocirati 128 kB, tada mora postojati slobodan blok memorije te veličine. Ne možemo umjesto njega odabrati dva odvojena bloka memorije od po 64 kB veličine.

Problem realokacije sprječava se korištenjem alokatora baziranog na stogu ili *pool* alokatora. Ako ne koristimo jedan od ova dva tipa alokatora, pa do fragmentacije ipak dođe, tada primjenjujemo postupak defragmentacije kod kojeg se svaki blok alocirane memorije pomiče na najnižu moguću adresu.



Slika 5.1: Fragmentacija memorije

5.2 Audio sustav

Iako su zadnjih godina zvukovi u igrama zapostavljeni i prednost je dana grafici, glazba i zvučni efekti u igrama imaju velik utjecaj na kompletnu atmosferu igre. Mnoge popularne igre čak je moguće prepoznati samo slušanjem njihove glazbe.

Općenito razlikujemo dva tipa zvukova: glazbene teme koje sviraju u pozadini, i zvučni efekti koji odgovaraju događajima u igri, kao što su eksplozije.

Glazbene teme sviraju u pozadini ovisno o stanju u kojem se igra nalazi. Na primjer, možemo imati različite glazbene teme za glavni izbornik i za samu igru. Glazbene teme obično su u MP3 ili OGG formatu. Glazbene teme obično su dugačke i zauzimaju puno memorije, pa se zbog toga često ne učitavaju u radnu memoriju u komadu. Umjesto toga, samo se mali dijelovi glazbene teme čuvaju u memoriji, a novi se učitavaju s vremenom, kako glazbena tema napreduje.

Zvučni efekti koriste se u trenutku nekog događaja unutar igre kojeg možemo reprezentirati zvukom. Na primjer, to može biti pucnjava iz pištolja, eksplozije, zvukovi koraka i tako dalje. Za razliku od glazbenih tema, zvučni efekti su obično jako kratki. Zbog toga moguće ih je potpuno učitati u memoriju, a obično su u WAV formatu.

Jačina zvuka i ostale karakteristike zvučnog efekta ovise o poziciji izvora zvuka i o

poziciji primatelja zvuka. Audio sustavi primjenjuju različite efekte i filtere zvuka kako bi što realističnije prikazali zvukove unutar 2D ili 3D prostora igre.

5.3 Skriptni jezik

Skriptni jezik može se definirati kao programski jezik čija je glavna uloga omogućiti korisniku kontrolu i modificiranje ponašanja softverske aplikacije. U kontekstu pokretača igre, skriptni jezik je viši programski jezik relativno lagan za korištenje koji korisnicima nudi pristup često korištenim značajkama pokretača igre. Skriptni jezik mogu koristiti programeri, ali i ljudi koji nisu programeri, u svrhu razvoja nove igre ili modificiranja postojeće. Skriptni jezik najčešće se koristi za proširenje i oblikovanje funkcionalnosti objekata unutar igre. U ovom poglavlju objasniti ćemo pojam skriptnog jezika i navesti ćemo neke popularne programske jezike koji se često koriste kao skriptni jezici u pokretačima igre.

Karakteristike skriptnog jezika

Skriptni jezici obično su interpretirani preko virtualnog stroja, dakle nisu kompilirani. Ovo im donosi fleksibilnost, prenosivost i mogućnost brze iteracije. Takav programski kod neovisan je o platformi, a jednostavno je tretiran od strane pokretača igre. Učitava se u memoriju kao svaki drugi resurs igre, bez pomoći operacijskog sustava. Upravo zbog toga što se programski kod izvršava na virtualnom stroju, umjesto direktno na procesoru, pokretač igre može kontrolirati kako i kada će pokrenuti taj programski kod.

Skriptni jezici su lagani, njihovi virtualni strojevi su jednostavni i obično koriste relativno malo memorije.

Mogućnost brze iteracije još je jedna prednost skriptnih jezika. Naime, kada god promijenimo izvorni kod pokretača igre, program se mora kompilirati ispočetka, a igra se mora ugasiti i upaliti kako bi se vidjele promjene. S druge strane, kada promijenimo skriptni kod, promjene je moguće vidjeti veoma brzo jer nema potrebe za ponovnom kompilacijom. Neki pokretači igre dozvoljavaju izmjenu skriptnog koda kada je igra već pokrenuta, a izmjene su vidljive bez ponovnog pokretanja igre.

Skriptni jezici obično su jednostavniji za korištenje od originalnog programskog jezika u kojem je pisan pokretač igre. Skriptni jezik rješavanje čestih zadataka čini laganim, intuitivnim i manja je šansa za pogreške.

Popularni skriptni jezici

Moguće je dizajnirati i implementirati skriptni jezik od nule, ali to se obično ne isplati. Razvojni programeri najčešće uzimaju gotove skriptni jezici, a zatim ih u većoj ili manjoj mjeri modificiraju prema svojim potrebama. Ovdje ćemo navesti osnovne značajke popularnih skriptnih jezika, programskog jezika Lua i programskog jezika Python.

Lua je poznati skriptni jezik koji se vrlo lako integrira unutar aplikacija, pa tako i unutar pokretača igre. Glavne prednosti skriptnog jezika Lua su dobra dokumentacija, odlične performanse izvođenja u stvarnom vremenu, prenosivost, a osim toga potpuno je besplatna i otvorenog je koda. Jedna od prednosti je sučelje prema programskom jeziku C. Naime, Lua virtualni stroj može zvati funkcije napisane u programskom jeziku C istom lakoćom kao što poziva funkcije napisane u programskom jeziku Lua. Lua je jedan od najpopularnijih izbora što se tiče skriptnih jezika unutar pokretača igre.

Python je proceduralni, objektno orijentirani, dinamički pisan skriptni jezik, također često korišten kao skriptni jezik u igrama. Glavne prednosti koje navodi su čista i čitljiva sintaksa, objektno orijentiranost, modularnost, odlična dokumentacija i upravljanje pogreškama.

Poglavlje 6

Alati korišteni za implementaciju 2D pokretača igre

U ovom poglavlju ukratko ćemo opisati svaki od alata koji se koriste za implementaciju jednostavnog 2D pokretača igre razvijenog u sklopu ovog diplomskog rada. Pokretač igre implementiran je u programskom jeziku C++, a za njegovu izradu koristili smo integrirano razvojno okruženje Visual Studio tvrtke Microsoft, odnosno njegovu besplatnu verziju pod nazivom Visual Studio Community.

Za razvoj pokretača igre koristili smo i neke dodatne biblioteke otvorenog koda koje su nam omogućile bržu i stabilniju implementaciju. Za implementaciju sustava za iscrtavanje koristili smo biblioteku SFML (Simple and Fast Multimedia Library), dok smo za implementaciju sustava za fiziku koristili biblioteku Box2D. Kao skriptni jezik odabrali smo programski jezik Lua, a za njegovu implementaciju u pokretač igre koristili smo pomoćnu biblioteku Sol. Za kontrolu verzije koristili smo web servis GitHub.

6.1 Microsoft Visual Studio i programski jezik C++

Prevođeni programski jezici, kao što je C++, zahtjevaju podršku prevoditelja (engl. *compiler*) i povezivača (engl. *linker*) kako bi se programski kod transformirao u izvršni program. Postoji mnogo dostupnih prevoditelja i povezivača za programski jezik C++, ali za Microsoft-ovu platformu Windows najčešće se koristi paket Microsoft Visual Studio.

Visual Studio je integrirana razvojna okolina, što znači da sadrži dodatne značajke osim prevoditelja i povezivača. Neke od njih su uređivač teksta za pisanje programskog koda i alat za pronalaženje pogrešaka na razini programskog koda i na razini strojnog koda.

U nastavku poglavlja navesti ćemo neke osnovne informacije o programskom jeziku C++ i načinu njegovog prevođenja u izvršni program. U nastavku ovog rada očekuje se poznavanje osnovne sintakse programskog jezika C++.

Izvorne datoteke, datoteke zaglavlja i prijevodne jedinice

Program napisan u programskom jeziku C++ sastoji se od datoteka koje nazivamo izvorne datoteke. One obično imaju ekstenziju `.cpp`, a ponekad ih nazivamo i prijevodne jedinice (engl. *translation unit*), jer prevoditelj prevodi jednu po jednu izvornu datoteku iz programskog koda C++ u strojni kod.

Posebna vrsta izvorne datoteke je datoteka zaglavlja. Ona se često koristi za dijeljenje informacija između prijevodnih jedinica, kao što su prototipovi funkcija. Preprocesor programskog jezika C++ svaku pojavu linije `#include` zamijenjuje sadržajem odgovarajuće datoteke zaglavlja, a zatim se ta prijevodna jedinica šalje prevoditelju.

Dakle, iako datoteke zaglavlja postoje kao zasebne datoteke, zahvaljujući preprocesoru sve što prevoditelj vidi su prijevodne jedinice.

Biblioteke, izvršne datoteke i dinamički povezane biblioteke

Nakon što se prijevodna jedinica prevede, rezultirajući strojni kod smješta se u objektne datoteke sa ekstenzijom `.obj` ili `.o`, ovisno o operacijskom sustavu. Strojni kod unutar objektnih datoteka je prenosiv i nepovezan. Prenosiv je jer memorijske adrese vezane za taj strojni kod još nisu određene, a nepovezan je jer vanjske reference prema funkcijama i globalnim podacima definiranim izvan prijevodne jedinice još nisu riješene.

Objektne datoteke mogu se grupirati u grupe koje nazivamo biblioteke. Glavna prednost biblioteka je jednostavnost korištenja, odnosno pakiranje velikog broja objektnih datoteka u jednu datoteku laku za korištenje.

Objektne datoteke i biblioteke zatim se povezuju u izvršnu datoteku, a za taj postupak zadužen je povezivač. Izvršna datoteka sadrži strojni kod potpuno spreman za učitavanje i pokretanje od strane operacijskog sustava. Zadaci povezivača su računanje konačnih relativnih memorijskih adresa cijelog strojnog koda, i rješavanje svih vanjskih referenci na vanjske funkcije i globalne podatke nekih drugih prijevodnih jedinica, odnosno sada već objektnih datoteka.

Bitno je napomenuti da je strojni kod unutar izvršne datoteke i dalje prenosiv, odnosno adrese svih instrukcija i podataka su i dalje relativne u odnosu na proizvoljnu baznu me-

morijsku adresu. Konačna apsolutna bazna memorijska adresa poznata je tek u trenutku učitavanja programa u memoriju, neposredno prije pokretanja programa.

Dinamički povezane biblioteke su posebna vrsta biblioteke koja ima neke značajke običnih datoteka i neke značajke izvršnih datoteka. Izvršna datoteka koja koristi dinamički povezane biblioteke sadrži nepotpuno povezan strojni kod. On se u potpunosti povezuje tek u trenutku pokretanja programa, tako što se pronalaze odgovarajuće dinamički povezane datoteke. Dinamički povezane datoteke vrlo su korisne jer se mogu ažurirati i mijenjati bez potrebe za izmjenama izvršnih datoteka koje ih koriste.

6.2 Biblioteka SFML

Biblioteka **SFML** je zapravo razvojni okvir u programskom jeziku C++. SFML preko jednostavnog programskog sučelja omogućava brz razvoj programa visokih performansi. SFML je multimedijska biblioteka, a podijeljena je u pet modula:

- *System*: Ovo je bazni modul na kojem su građeni ostali moduli. On sadrži implementacije klasa za dvodimenzionalne i trodimenzionalne vektore, satove, dretve, Unicode nizove znakova i brojne druge.
- *Window*: Ovaj modul omogućava stvaranje prozora aplikacije i obradu ulaznih podataka korisnika, kao što je pomicanje miša i pritisak tipke na tipkovnici.
- *Graphics*: Ovaj modul zadužen je za dvodimenzionalno iscrtavanje slika, teksta, oblika i boja.
- *Audio*: Rad sa zvukovima omogućen je preko ovog modula.
- *Network*: SFML također omogućuje slanje podataka preko interneta, te rad sa protokolima kao što su HTTP ili FTP.

Svaki od ovih modula nalazi se u zasebnoj biblioteci, što omogućava korištenje samo onih modula koji su nam potrebni.

Slijedi primjer minimalne aplikacije koja prikazuje neke osnovne funkcionalnosti:

```
1 #include <SFML/Graphics.hpp>
2 int main() {
3   sf::RenderWindow window(sf::VideoMode(640, 480), "SFML Application");
4   sf::CircleShape shape;
5   shape.setRadius(40.f);
6   shape.setPosition(100.f, 100.f);
7   shape.setFillColor(sf::Color::Cyan);
8   while (window.isOpen()) {
9     sf::Event event;
```

```
10      while (window.pollEvent(event)) {
11        if (event.type == sf::Event::Closed)
12        window.close();
13      }
14      window.clear();
15      window.draw(shape);
16      window.display();
17    }
18  }
```

Ovaj program otvoriti će prozor širok 640 piksela i visok 480 piksela, a naslov će mu biti `SFML Application`. Zatim će se stvoriti krug svijetlo plave boje i sve dok je prozor otvoren iscrtavati će se na ekran. Uz iscrtavanje kruga, program će u svakom koraku petlje provjeriti eventualne ulazne podatke korisnika. U ovom slučaju obraditi ćemo samo događaj tipa `sf::Event::Closed`, koji označava zahtjev za gašenjem programa.

U razvoju 2D pokretača igre koristit ćemo module *System*, *Window* i *Graphics*. Biblioteku SFML najviše ćemo koristiti za iscrtavanje objekata na ekran pomoću modula Graphics. Više o biblioteci može se vidjeti na poveznici [8] i u knjizi [3].

6.3 Biblioteka Box2D

Box2D je C++ biblioteka otvorenog koda namjenjena za otkrivanje sudara i simulaciju kretanja krutih tijela u dvodimenzionalnom prostoru. Biblioteka ne koristi spremnike iz standardne C++ biblioteke, što ju čini iznimno prenosivom. Osim toga odlikuje se dobro napisanom dokumentacijom i širokom zajednicom korisnika.

Neke od mogućnosti sustava za otkrivanje sudara su:

- kontinuirano otkrivanje sudara;
- povratne informacije o kontaktima, u trenutku nastajanja i prestajanja kontakta;
- podrška za konveksne mnogokute i krugove;
- mogućnost pridruživanja više oblika jednom krutom tijelu;
- grupe i kategorije sudarajućih objekata.

Box2D sadrži i sustav za fiziku, odnosno simulaciju kretanja krutih tijela. Neke njegove mogućnosti su:

- kontinuirana simulacija sa pronalaskom vremenskog trenutka sudara;
- sila trenja i sila restitucije;

- različiti tipovi spojeva krutih tijela;
- visoka točnost simulacije;
- visoka točnost sila koje nastaju kao reakcije na sudare.

U razvoju 2D pokretača najviše ćemo koristiti klasu `b2World` koja predstavlja svijet u kojem se simulira dinamika krutih tijela. Osim toga, često ćemo koristiti klase `b2CircleShape` i `b2PolygonShape` koje predstavljaju geometrijske oblike kojima ćemo reprezentirati objekte virtualnog svijeta igre. Ostale detalje o biblioteci Box2D mogu se vidjeti na poveznici [7].

6.4 Skriptni jezik Lua

Lua ja snažan, učinkovit, a izrazito lagan skriptni jezik. Podražava gotovo sve oblike programiranja: proceduralno programiranje, objektno orijentirano programiranje, funkcionalno programiranje i programiranje vođeno podacima.

Tipovi podataka u programskom jeziku Lua su dinamički, što znači da varijable nemaju tip, ali objekti imaju. Lua je interpretirani programski jezik, dakle pokreće se na virtualnom stroju, a osim toga sadrži i sustav za automatsko sakupljanje otpada (engl. *garbage collection*).

Lua se koristi u mnogim komercijalnim aplikacijama (npr. Adobe Photoshop Lightroom) i u mnogim video igrama (npr. World of Warcraft i Angry birds), što ju čini vodećim skriptnim jezikom u igrama.

Lua biblioteka je iznimno prenosiva, a moguće ju je ugraditi u bilo koju platformu koja sadrži standardni C prevodilac. Programski jezik Lua moguće je pokrenuti na svim vrstama Windows i Unix operacijskih sustava, na svim Android, iOS i Windows pametnim telefonima, te na raznim vrstama mikroprocesora.

Zbog ovih, ali i mnogih drugih mogućnosti, kao skriptni jezik 2D pokretača igre odabran je skriptni jezik Lua. Više o skriptnom jeziku Lua nalazi se na [10].

Pomoćna biblioteka Sol

Sol [9] je pomoćna C++ biblioteka koja omogućava vrlo jednostavnu implementaciju programskog jezika Lua u C++ projekt. Trenutno podržava sve verzije programskog jezika Lua veće od verzije 5.1. Cijela Sol biblioteka dostupna je i u obliku jedne datoteke zaglavlja, što omogućava vrlo jednostavnu integraciju. Slijedi kratak primjer korištenja biblioteke Sol.

```
#include <sol.hpp>
#include <cassert>
int main() {
  sol::state lua;
  int x = 0;
  lua.set_function("beep", [&x]{ ++x; });
  lua.script("beep()");
  assert(x == 1);
}
```

O ovom primjeru možemo vidjeti definiciju Lua funkcije `beep()` pomoću C++ funkcije `set_function()`. Funkcija `beep()` jednostavno povećava vrijednost varijable `x` za jedan. Pomoću funkcije `script()` pozivamo Lua interpreter koji pokreće funkciju `beep()` i povećava vrijednost varijable `x` za jedan.

Više o sintaksi programskog jezika Lua i samoj implementaciji reći ćemo u sljedećem poglavlju.

6.5 Upravljanje verzijama izvornog koda

Za upravljanje verzijama izvornog koda projekta koristili smo web servis GitHub i besplatan GitHub dodatak za Visual Studio koji omogućuje kontrolu nad verzijama izvornog koda preko jednostavnog korisničkog sučelja.

Poglavlje 7

Implementacija 2D pokretača igre

U ovom poglavlju opisat ćemo implementaciju jednostavnog 2D pokretača igre razvijenog u sklopu ovog diplomskog rada. Osnovna ideja je napraviti pokretač igre temeljen na komponentama (engl. *component based game engine*). Svakom objektu u igri moći će se pridružiti razne komponente pomoću kojih se opisuje njegovo ponašanje unutar virtualnog svijeta igre.

Cilj je korisniku omogućiti jednostavno oblikovanje objekta sa nekim traženim svojstvima. Osim unaprijed implementiranih komponenti, korisnik će moći objektu kao komponentu dodati i vlastiti kod kojim će kontrolirati tok igre. Osim toga, korisniku će na raspolaganju biti klase preko kojih lako može saznati ulazne podatke i druge informacije o trenutnom stanju igre (npr. pritisak tipke na tipkovnici ili ukupno vrijeme proteklo od pokretanja igre).

Prvo ćemo opisati strukturu i dijelove pokretača igre, a zatim ćemo vidjeti kako su svi ti dijelovi povezani unutar **glavne petlje igre** (engl. *game loop*). Nakon toga opisat ćemo glavne klase koje upravljaju pokretačem igre i preko kojih korisnik može doći do informacija o stanju igre. Zatim ćemo navesti implementirane **komponente**, te objasniti njihove funkcije i načine na koje utječu na objekte unutar igre.

Nakon što se upoznamo sa implementacijom pokretača igre, reći ćemo nešto više o skriptnom jeziku Lua i načinu na koji je integriran u pokretača igre. Kroz par jednostavnih primjera upoznati ćemo se sa osnovama sintakse i načinom korištenja skriptnog jezika Lua u oblikovanju logike igre. Na samom kraju vidjeti ćemo primjer jednostavne igre izrađene u ovom pokretaču igre, te ćemo navesti planove i ideje o budućem poboljšanju pokretača igre i dodavanju novih komponenti.

Izvorni programski kod i kompletan Visual Studio projekt može se naći na linku: [6].

7.1 Struktura i dijelovi pokretača

Pokretač igre sastoji se od četiri glavna dijela:

1. Sustav za obradu ulaznih podataka obrađuje sve ulazne podatke koje dobije od korisnika. To mogu biti pomicanje miša, pritisak tipke miša ili pritisak tipke na tipkovnici. U svakom vremenskom trenutku poznato je koja tipka je upravo pritisnuta ili puštena. Za obradu ulaznih podataka pomoći će nam biblioteka SFML.

2. Sustav za simulaciju fizike zapravo poziva funkcije iz biblioteke Box2D, koja je odgovorna za otkrivanje sudara i simulaciju fizike krutih tijela. U svakom vremenskom koraku svijet fizike napreduje za mali vremenski interval.

3. Sustav za upravljanje skriptama brine o pozivanju odgovarajućih funkcija unutar korisničkih skripti. Svaka instanca korisničke skripte koja je pridružena nekom objektu u igri može sadržavati funkcije koje se automatski pozivaju tijekom nekih određenih događaja. Na primjer, funkcija `Start()` pozvati će se na početku igre, a funkcija `OnCollision()` poziva se u trenutku sudara objekta kojem je pridružena skripta sa nekim drugim objektom.

4. Sustav za iscrtavanje brine o iscrtavanju igre na ekran, a za to koristi bibilioteku SFML. U svakom vremenskom trenutku ažuriraju se pozicije i rotacije objekata, a zatim se objekti iscrtavaju na ekran.

Svaka od komponenti pridruženih objektu unutar igre utječe na jedan od ova četiri sustava. Na primjer, iscrtavati će se samo objekti koji imaju pridruženu komponentu `Sprite`, a u sudarima sustava za fiziku sudjelovati će samo objekti koji imaju pridruženu komponentu `Collider`.

7.2 Klasa GameEngine i glavna petlja igre

Pokretač igre predstavljen je klasom `GameEngine`. Ova klasa implementirana je prema oblikovnom obrascu `singleton`. Pokretanje igre vrlo je jednostavno, a dovoljno je pozvati statičku funkciju `Run()` iz klase `GameEngine`. Glavni program tada izgleda ovako:

```
1 int main() {
2   GameEngine::Run();
3   return 0;
4 }
```

Glavna petlja igre nalazi se unutar funkcije `Run()`, a izgleda ovako:

```
void GameEngine::Run(){
  GameEngine::Create();
  while (gameEngine->window->isOpen()) {
    gameEngine->ProcessInput();
    gameEngine->UpdatePhysics();
    gameEngine->UpdateScripts();
    gameEngine->UpdateGraphics();
    Sleep(1000 * timeStep);    // timeStep = 1.0f / 60.0f;
  }
}
```

Funkcija `Create()` inicijalizira podsustave potrebne za izvršavanje igre, kao što su Lua virtualni stroj, svijet fizike i prozor igre. Sve dok je prozor igre otvoren, u svakom vremenskom koraku pozivamo odgovarajuće sustave da ažuriraju svoje stanje. Prvo se obrađuju ulazni podaci, zatim se ažurira sustav za fiziku, nakon toga se pozivaju korisničke skripte, a na kraju se igra iscrtava na ekran. Možemo vidjeti da se ovaj korak ponavlja otprilike 60 puta u sekundi, a broj ponavljanja određujemo preko varijable `timeStep`.

Obrada ulaznih podataka

Funkcija za obradu ulaznih podataka koristi biblioteku SFML i njezinu klasu `sf::RenderWindow`. Funkcija `ProcessInput()` izgleda ovako:

```
void GameEngine::ProcessInput() {
  Input::ClearInputDownUpFlags();
  sf::Event event;
  while (gameEngine->window->pollEvent(event)) {
    if (event.type == sf::Event::Closed) {
      gameEngine->window->close();
      return;
    } else if (event.type == sf::Event::KeyPressed) {
      Input::OnKeyDown(event.key.code);
    } else if (event.type == sf::Event::KeyReleased) {
      Input::OnKeyUp(event.key.code);
    } else if (event.type == sf::Event::MouseButtonPressed) {
      Input::OnMouseButtonDown(event.mouseButton.button);
    } else if (event.type == sf::Event::MouseButtonReleased) {
      Input::OnMouseButtonUp(event.mouseButton.button);
    }
  }
}
```

Prilikom svakog događaja pozivaju se statičke funkcije klase `Input`, koja zatim obrađuje te podatke i pruža sučelje za dohvaćanje tih podataka ostalim sustavima pokretača igre. Više o klasi `Input` reći ćemo kasnije.

Ažuriranje sustava za fiziku

Funkcija `UpdatePhysics()` vrlo je jednostavna:

```
1 void GameEngine::UpdatePhysics(){
2   gameEngine->physicsWorld->Step(timeStep, velocityIterations,
      positionIterations);
3 }
```

Svijet fizike predstavljen je objektom `physicsWorld` tipa `b2World` iz biblioteke Box2D. Prilikom svakog koraka glavne petlje igre pozivamo funkciju `Step(...)` na objektu `physicsWorld`, čime svijet fizike "napreduje" za određeni vremenski interval. Vremenski interval određen je već spomenutom varijablom `timeStep`.

Pozivanje korisničkih skripti

Funkcija `UpdateScripts` također je jednostavna, a izgleda ovako:

```
1 void GameEngine::UpdateScripts() {
2   Script::Update();
3 }
```

Pozivanje korisničkih skripti prepušta se klasi `Script` i njezinoj statičkoj funkciji `Update()`. Više o toj klasi, koja je ujedno i vrsta komponente, reći ćemo kasnije.

Iscrtavanje igre

Iscrtavanje igre izvršava se na samom kraju, preko funkcije `UpdateGraphics`. Ova funkcija izgleda ovako

```
1  void GameEngine::UpdateGraphics(){
2    window->clear();
3
4    ...
5
6    // Render sprites
7    for (Sprite* s : Sprite::sprites) {
8      b2Vec2 position = s->gameObject.transform->GetPosition();
9      float32 angle = -1.0f * s->gameObject.transform->GetRotation();
10     s->sprite.setPosition(position.x, windowSizeY - position.y);
11     s->sprite.setRotation(angle);
12     window->draw(s->sprite);
13   }
14   window->display();
15 }
```

Prvi korak je obrisati trenutnu sliku prozora igre. Nakon toga, u `for` petlji prolazimo kroz listu svih objekata koje iscrtavamo. Objekti koje iscrtavamo predstavljeni su

klasom `Sprite`, a više o njoj reći ćemo kasnije. Za svaki objekt koji iscrtavamo, prvo ažuriramo njegovu poziciju i rotaciju, a zatim ga iscrtavamo u prozor igre pomoću funkcije `draw(...)`.

7.3 Implementacija ostalih klasa

U ovom poglavlju opisat ćemo ostale klasa koje su bitan dio pokretača igre. Osim prethodno opisane klase `GameEngine` koja predstavlja pokretača igre, najbitnije klase su `GameObject`, `Component` i `Input`.

Klasa `GameObject`

Klasa `GameObject` predstavlja jedan objekt unutar igre. Svaki objekt sadrži varijablu `name` koja predstavlja ime objekta, i sadrži listu komponenti koje su mu pridružene. Ova klasa također sadrži i statičku listu svih objekata tipa `GameObject`, i na taj način omogućava dohvaćanje određenog objekta preko njegovog imena. Varijable članice klase su sljedeće:

```
static std::list<GameObject*> gameObjects;
std::list<Component*> components;
std::string name;
Transform *transform;
```

Više o klasi `Component` reći ćemo u sljedećem paragrafu. Komponente se mogu dodati i ukloniti preko funkcija `AddComponent()` i `RemoveComponent()`. Preko funkcije `GetComponent()` moguće je dohvatiti željenu komponentu. Navedene funkcije su parametrizirane i izgledaju ovako:

```
template <typename T>
T* AddComponent() {
  T* newComponent = new T(*this);
  components.push_back(newComponent);
  return newComponent;
}
template <typename T>
T* GetComponent() {
  for each (Component* comp in components) {
    if (dynamic_cast<T*>(comp)) {
      return dynamic_cast<T*>(comp);
    }
  }
  return nullptr;
}
```

Klasa `Component`

Klasa `Component` je bazna klasa za svaku komponentu. Na ovaj način svi tipovi komponenti mogu se lagano spremati u jednu listu čiji elementi su tipa `Component`. Ova klasa sadrži samo funkcije koje skriptnom jezuku Lua omogućavaju dinamičku konverziju tipova komponente.

Klasa `Input`

Klasa `Input` sastoji se od nekoliko statičkih objekata koji sadrže informacije o trenutnom stanju ulaznih uređaja. Unutar glavne petlje igre pozivaju se funkcije koje ažuriraju stanje tih objekata. Te funkcije izgledaju ovako:

```
1  void Input::ClearInputDownUpFlags() {
2    for (sf::Keyboard::Key key : keyboardKeys) {
3      keyboardKeyDown[key] = false;
4      keyboardKeyUp[key] = false;
5    }
6    for (sf::Mouse::Button button : mouseButtons) {
7      mouseButtonDown[button] = false;
8      mouseButtonUp[button] = false;
9    }
10 }
11 void Input::OnKeyDown(sf::Keyboard::Key key) {
12   keyboardKeyDown[key] = true;
13 }
14 void Input::OnKeyUp(sf::Keyboard::Key key) {
15   keyboardKeyUp[key] = true;
16 }
17 void Input::OnMouseButtonDown(sf::Mouse::Button button) {
18   mouseButtonDown[button] = true;
19 }
20 void Input::OnMouseButtonUp(sf::Mouse::Button button) {
21   mouseButtonUp[button] = true;
22 }
```

Klasa sadrži i nekoliko funkcija preko kojih se dohvaća trenutno stanje ulaznih uređaja. Funkcije `GetKeyDown(...)` i `GetKeyUp(...)` vraćaju vrijednost `true` ako je tražena tipka pritisnuta ili puštena upravo u tom vremenskom trenutku, a inače vraća `false`. Funkcija `GetKey(...)` vraća vrijednost `true` ukoliko je tražena tipka pritisnuta.

7.4 Implementacija komponenti

Svi tipovi komponenti implementirani su preko klasa koje nasljeđuju baznu klasu `Component`. Komponente se mogu dodati ili ukloniti sa objekta u igri, a pomoću njih oblikujemo

ponašanje svakog objekta igre. U ovom poglavlju ćemo navesti i objasniti tipove komponenti implementirane u pokretaču igre.

Komponenta `Transform`

Komponenta `Transform` uvijek je dodana objektu igre, od trenutka njegove inicijalizacije. Ova komponenta sadrži poziciju i rotaciju objekta, te veličinu objekta. Ti podaci čuvaju se u objektu tipa `b2Body` biblioteke Box2D.

Osim toga, komponenta `Transform` je zadužena za čuvanje podataka o hijerarhiji objekata u sceni. Svaki objekt ima točno jednog objekta roditelja, a može imati nula ili više objekata djece. Ovo svojstvo omogućava kretanje više objekata kao da su cjelina. Na primjer, ako se lik unutar igre nalazi u vozilu, tada tom liku za roditelja možemo postaviti vozilo u kojem se nalazi. Lik će se tada kretati zajedno sa vozilom, na način na koji to očekujemo. Konstruktor komponente `Transform` izgleda ovako:

```
1 Transform::Transform(GameObject& parentGO) : Component(parentGO){
2   b2BodyDef bodyDef;
3   bodyDef.type = b2_staticBody;
4   bodyDef.position.Set(0.0f, 0.0f);
5   body = GameEngine::GetInstance()->physicsWorld->CreateBody(&bodyDef);
6 }
```

Objekt body tipa `b2Body` stvaramo preko funkcije `CreateBody(...)` objekta `physicsWorld`. Ta funkcija kao parametar prima malu strukturu podataka preko koje fizički definiramo objekt. Poziciju mu postavljamo u ishodište, a tip objekta postavljamo na `b2_staticBody`. Ovaj tip objekta označava da je objekt statičan, odnosno da ne sudjeluje u sudarima svijeta fizike. Objekt će postati dinamičan i sudjelovati u sudarima tek ako mu pridružimo komponente `Collider` i `Rigidbody`. O njima ćemo govoriti kasnije.

Pozicija i rotacija objekta mogu se dohvatiti i modificirati preko članskih funkcija `GetPosition()`, `GetRotation()`, `SetPosition(float x, float y)` i `SetRotation(float angle)`.

Komponenta `Sprite`

Komponenta `Sprite` predstavlja grafički izgled objekta u obliku slike. Ova komponenta čuva sliku (teksturu) koja se iscrtava na poziciji objekta. Ta slika čuva se u objektu tipa `sf::Texture` biblioteke SFML, a samo iscrtavanje ostvaruje se preko objekta tipa `sf::Sprite` iste biblioteke. Osim ova dva objekta, komponenta `Sprite` sadrži statičku listu `sprites` koja čuva sve komponente tipa `Sprite` koje se koriste u sceni:

```
1 static std::list<Sprite*> sprites;
```

Ova lista koristi se kod iscrtavanja unutar glavne petlje igre.

Komponenta `Collider`

Komponenta `Collider` je bazna klasa za posebne tipove oblika koji sudjeluju u sudarima. To su `BoxCollider`, `SphereCollider` i `PolygonCollider`.

Komponenta sadrži samo jedan objekt kao člana, a to je objekt fixture tipa `b2Fixture` iz biblioteke Box2D. Ovaj objekt povezuje geometrijski oblik koji sudjeluje u sudarima sa objektom `body` tipa `b2Body` iz komponente `Transform`. Objekt `fixture` je potreban jer je jednom objektu `body` moguće pridružiti više geometrijskih oblika koji sudjeluju u sudarima ako je objekt kompleksan. To ostvarujemo dodavanjem više komponenata `Collider` istom objektu.

Komponenta `SphereCollider`

Komponenta `SphereCollider` prestavlja objekt u obliku kružnice koji sudjeluje u sudarima. Konstruktor ove komponente izgleda ovako:

```
1 SphereCollider::SphereCollider(GameObject& parentGO) : Collider(parentGO
    ) {
2   if (gameObject.transform->body->GetType() == b2BodyType::b2_staticBody
    )
3     gameObject.transform->body->SetType(b2BodyType::b2_kinematicBody);
4 }
```

Tip objekta `body` komponente `Transform` postavlja se na `b2BodyType::b2_kinematicBody`. Ovaj objekt sada sudjeluje u sudarima. Bitno je napomenuti da ovaj objekt i dalje nije dinamičan i na njega ne djeluju sile, ako mu ne dodamo komponentu `RigidBody`.

Pomoću funkcije `SetSize(float x)` postavljamo radijus kružnice koja sudjeluje u sudarima. U slučaju promjene radijusa, prethodan objekt `fixture` potrebno je uništiti, te stvoriti novi objekt sa novim radijusom. Funkcija za postavljanje radijusa izgleda ovako:

```
1 void SphereCollider::SetSize(int x) {
2   if (fixture != nullptr)
3     gameObject.transform->body->DestroyFixture(fixture);
4
5   b2CircleShape circleShape;
6   circleShape.m_radius = 1.0f * x / GameEngine::physicsToGraphicsRatio;
7   fixture = gameObject.transform->body->CreateFixture(&circleShape, 0.0f
    );
8 }
```

Preko strukture podataka `2CircleShape` definiramo izgled geometrijskog oblika, a zatim ga stvaramo metodom `CreateFixture(...)`.

Ovdje možemo uočiti statičku konstantu `GameEngine::physicsToGraphicsRatio`. Ta konstanta potrebna je zbog omjera veličina u svijetu fizike i u svijetu igre. Na primjer,

udaljenost od jedne mjerne jedinice u svijetu fizike može odgovarati udaljenosti od sto mjernih jedinica u svijetu igre.

Komponenta `BoxCollider`

Komponenta `BoxCollider` slična je komponenti `SphereCollider`, samo što ona u sudarima predstavlja objekt u obliku pravokutnika. U ovom slučaju funkcija `SetSize(float x, float y)` prima dva parametra koja odgovaraju širini i visini pravokutnika.

Komponenta `Rigidbody`

Komponenta `RigidBody` objektu daje svojstva krutog tijela. Objekt kojem je pridružena komponenta `RigidBody` ima određenu masu i reagira na sile. Prilikom stvaranja komponente `Rigidbody`, tip objekta body komponente `Transform` postavlja se na `b2BodyType::b2_dynamicBody`. Ovo govori biblioteci Box2D kako ovaj objekt treba sudjelovati u simulacijama dinamike krutog tijela.

Komponenta `Camera`

Svijet igre iscrtava se u prozor igre preko komponente `Camera`. Objekt kojem je pridružena ova komponenta predstavlja virtualnu kameru unutar scene. Unutar scene uvijek mora postojati točno jedna glavna, trenutno aktivna kamera. Pokazivač na glavnu kameru čuvamo u statičkoj varijabli `mainCamera`, dok popis svih kamera čuvamo u listi `cameras`:

```
1 static Camera* mainCamera;
2 static std::list<Camera*> cameras;
```

Podaci o kameri potrebni biblioteci SFML za iscrtavanje scene čuvaju se u objektu view tipa `sf::View`.

Komponenta `Script`

Komponenta `Script` služi za povezivanje pokretača igre sa skriptom napisanom u skriptnom jeziku Lua. Klasa sadrži sljedeće varijable članice:

```
1 static std::list<Script*> scripts;
2 sol::state lua;
3 std::string scriptName;
```

Statička lista `scripts` služi nam za čuvanje svih komponenti tipa `Script` na jednom mjestu. Preko te liste možemo lagano iterirati po svim skriptama i zvati odgovarajuće funkcije. Objekt `lua` predstavlja skriptu koju želimo koristiti, dok varijabla `scriptName` predstavlja njezino ime.

Funkcija `Init(std::string name)` služi nam za inicijalizaciju skripte. Kao parametar prima samo ime datoteke u koju je zapisan programski kod, a sama funkcija izgleda ovako:

```
1  void Script::Init(std::string name){
2    if (scriptName != "")
3      return;
4
5    scriptName = name;
6    lua.open_libraries();
7    Script::AddUserTypes(lua);
8    lua["gameObject"] = gameObject;
9    lua.script_file(scriptName, [](lua_State* L, sol::
       protected_function_result pfr) {
10     sol::error err = pfr;
11     std::cout << err.what() << std::endl;
12     return pfr;
13   });
14   sol::protected_function luaStart = lua["Start"];
15   auto luaStartResult = luaStart();
16   if (!luaStartResult.valid()) {
17     sol::error err = luaStartResult;
18     std::cout << err.what() << std::endl;
19   }
20 }
```

Na početku možemo uočiti poziv funkciji `open_libaries()` koja učitava potrebne biblioteke programskog jezika Lua. Zatim pozivamo statičku funkciju `Script::AddUserTypes(lua)` koja će se pobrinuti da je Lua virtualni stroj upoznat sa tipovima podataka koje ćemo koristiti. Više o mostu između programskog jezika C++ i skriptnog jezika Lua reći ćemo u sljedećem poglavlju.

Naredbom `lua["gameObject"] = gameObject;` stvaramo novi Lua objekt i pridružujemo mu C++ objekt `gameObject`. Preko njega moći ćemo iz skripte pristupiti C++ objektu kojem je skripta pridružena. Nakon toga inicijaliziramo skriptu pomoću njezinog imena preko funkcije `script_file(...)`. Na kraju pozivamo funkciju `Start()` koja služi za inicijalizaciju vrijednosti varijabli unutar skripte. Primjer korištenja skripte vidjeti ćemo u sljedećem poglavlju.

Komponenta `Script` sadrži i jednu statičku funkciju `Update()` koja se poziva unutar glavne petlje igre. Ta statička funkcija prelazi po listi svih skripti i poziva funkciju `Update` unutar svake skripte.

```
1  void Script::Update() {
2    for (Script* script : scripts) {
3      sol::protected_function luaUpdate = script->lua["Update"];
4      auto luaUpdateResult = luaUpdate();
```

```
5       if (!luaUpdateResult.valid()) {
6         sol::error err = luaUpdateResult;
7         std::cout << err.what() << std::endl;
8       }
9     }
10 }
```

7.5 Skriptni jezik Lua

Skriptni jezik Lua omogućava nam brz razvoj igre bez potrebe za stalnim prevođenjem pokretača igre. Pokretač igre prevodi se samo jednom, a igra se zatim u potpunosti razvija preko skriptnog jezika. U ovom poglavlju objasnit ćemo način na koji je skriptni jezik Lua implementiran u pokretač igre. Kao pomoćna biblioteka koristi se biblioteka Sol, koja olakšava korištenje skriptnog jezika Lua. Na kraju ćemo navesti jedan primjer korištenja skripte.

Most između programskog jezika C++ i skriptnog jezika Lua

Jednostavan most između programskog jezika C++ i skriptnog jezika Lua pruža nam bibiloteka Sol. Svaka instanca skripte napisane u skriptnom jeziku Lua predstavljana je objektom tipa `sol::state`. Taj objekt sadrži funkcije preko kojih možemo učitati skriptu ili pozvati neke funkcije unutar skripte.

Lua virtualni stroj ne zna automatski raditi sa svim tipovima podataka koje smo definirali. Zato koristimo funkciju `new_usertype<typename T>(...)` objekta tipa `sol::state` kako bismo definirali novi tip podatka koji će Lua znati koristiti. U 2D pokretaču igre svaka klasa koji smo definirali implementira funkciju `AddUserType(sol::state &lua)` preko koje se objektu `lua` dodaje novi tip podatka koji će znati koristiti. Na primjer, klasa `GameObject` implementira funkciju `AddUserType` na sljedeći način:

```
void GameObject::AddUserType(sol::state & lua){
  lua.new_usertype<GameObject>("GameObject",
    "name", &GameObject::name,
    "AddComponent", &GameObject::AddComponentLua,
    "GetComponent", &GameObject::GetComponentLua,
    "GetComponents", &GameObject::GetComponentsLua,
    "RemoveComponent", &GameObject::RemoveComponentLua
    );
}
```

Funkcija `new_usertype` je parametrizirana, a u ovom slučaju šaljemo joj tip `GameObject`. Prvi parametar funkcije predstavlja ime tipa preko kojeg će Lua pristupati ovoj klasi.

Ovo ime ne mora nužno biti isto kao ime klase u programskom jeziku C++, ali ovdje i u većini drugih klasa ćemo zadržati isto ime.

Nakon toga navodimo funkcije klase `GameObject` koje želimo učiniti dostupnima skriptnom jeziku Lua. Prvo navodimo ime funkcije preko kojega će Lua pristupati toj funkciji, a zatim navodimo adresu te funkcije. U ovom slučaju imena funkcija preko kojeg će Lua pristupati funkcijama klase `GameObject` ne odgovaraju u potpunosti imenima tih funkcija u programskog jeziku C++. Razlog tome je što Lua ne podržava parametrizirane funkcije, pa funkcije kao što su `GameObject::AddComponent<typename T>()` ne možemo koristiti direktno. Zato koristimo funkciju `GameObject::AddComponentLua-(std::string componentType)` gdje tip komponente koju želimo dodati šaljemo preko parametra `componentType`.

Prilikom instanciranja novog objekta tipa `Script`, pozivaju se funkcije `AddUserType` na svim klasama i komponentama pokretača igre, kako bi im se moglo pristupiti iz te konkretne skripte. Primjer poziva funkcije skriptnog jezika Lua iz programskog jezika C++ vidjeli smo i ranije, a izgleda ovako:

```
1 sol::protected_function luaStart = lua["Start"];
2 auto luaStartResult = luaStart();
3 if (!luaStartResult.valid()) {
4   sol::error err = luaStartResult;
5   std::cout << err.what() << std::endl;
6 }
```

Ovdje koristimo objekt tipa `sol::protected_function` kojem pridružujemo funkciju `luaStart`. Pomoću indeksa `Start` dohvaćamo skriptnu funkciju imena `Start`, iz skripte pridružene objektu `lua` koji je tipa `sol::state`. Nakon poziva funkcije lako možemo provjeriti da li je došlo do eventualnih grešaka kao što je, na primjer, nepostojanje funkcije tog imena.

Primjer korištenja skripte

Scenu i objekte unutar nje oblikujemo pomoću skripte `init.lua`, koja se automatski poziva prilikom pokretanja igre. Skripta tog imena mora postojati, a možemo ju usporediti sa funkcijom `main()` koju svaki program napisan u programskom jeziku C mora sadržavati. Pomoću skripte `init` stvaramo objekte koje ćemo koristiti i pridružujemo im komponente koje će upravljati njihovim ponašanjem.

U sljedećem primjeru možemo vidjeti jednostavnu skriptu koja omogućava pomicanje objekta korištenjem tipkovnice.

```
1 function Start()
2   transform = gameObject:GetComponent("Transform"):ToTransform()
3   positionDelta = Vector2.new()
```

```
4   positionDelta.x = 1
5   positionDelta.y = 1
6 end
7 function Update()
8   position = transform:GetPosition()
9   if(Input.GetKey("W")) then
10    position.y = position.y + positionDelta.y
11  end
12  if(Input.GetKey("A")) then
13    position.x = position.x - positionDelta.x
14  end
15  if(Input.GetKey("S")) then
16    position.y = position.y - positionDelta.y
17  end
18  if(Input.GetKey("D")) then
19    position.x = position.x + positionDelta.x
20  end
21  transform:SetPosition(position.x, position.y)
22 end
```

Funkcija `Start()` poziva se na samom početku, prilikom inicijalizacije skripte, tj. komponente tipa `Script`, koju smo dodali nekom objektu. Ova funkcija prvo varijabli `transform` pridružuje komponentu tipa `Transform` objekta na kojem se nalazi. Podsjetimo, komponenta `Transform` čuva podatke o poziciji i rotaciji objekta. Zatim deklariramo novi objekt tipa `Vector2`, koji jednostavno sadrži dva broja. Vrijednosti tog vektora koristiti ćemo kasnije za pomicanje objekta, a što su vrijednosti vektora veće, objekt će se brže kretati.

Funkcija `Update()` automatski se poziva u svakom koraku glavne petlje igre. Na početku se dohvaća trenutna pozicija objekta i posprema u objekt `position`. Zatim provjeravamo da li je neka od tipki W, A, S ili D pritisnuta. Ako je, mijenjamo odgovarajuću varijablu vektora `position`. Na kraju objektu postavljamo novu poziciju.

7.6 Primjer korištenja pokretača igre

U ovom poglavlju opisat ćemo jednostavnu igru razvijenu pomoću implementiranog 2D pokretača igre. Igra primarno služi za prikaz osnovnih funkcionalnosti pokretača igre, te za prezentaciju korištenja skriptnog jezika Lua.

Igrač u ovoj igri može upravljati lopticom, a cilj igre je popesti se što je više moguće po platformama koje su nasumično generirane. Tijekom cijelog vremena izvođenja igre, sa dna scene dolazi voda koja dostiže lopticu. Kada voda dostigne lopticu, igra je gotova.

Igru postavljamo koristeći skriptu `init.lua`. Ona izgleda ovako:

```
function Init()
  ball = GameObject.new()
  ball.name = "ball"
  ball:AddComponent("Sprite"):ToSprite():SetTexture("blueBall.png")
  ball:AddComponent("SphereCollider"):ToSphereCollider():SetSize(40 / 2)
  ball:AddComponent("Rigidbody"):ToRigidbody():SetMass(10)  ball:
    GetComponent("Transform"):ToTransform():SetPosition(0, 40 / 2) ball:
    AddComponent("Script"):ToScript():Init("ball.lua")

  water = GameObject.new()
  water:AddComponent("Sprite"):ToSprite():SetTexture("water.png")
  water:AddComponent("Script"):ToScript():Init("water.lua")
  water:GetComponent("Transform"):ToTransform():SetPosition(0, -450)
  platformGenerator = GameObject.new()
  platformGenerator:AddComponent("Script"):ToScript():Init("generator.
    lua")
  camera = GameObject.new()
  camera:AddComponent("Camera")
  camera:AddComponent("Script"):ToScript():Init("camera.lua")
end
```

Možemo uočiti četiri glavna objekta igre. Prvi je objekt `ball` koji predstavlja lopticu kojom igrač može upravljati. Ovom objektu dodajemo skriptu `ball.lua` koja će upravljati kretnjama loptice koristeći ulazne podatke igrača.

Drugi objekt, `water`, predstavlja vodu koja dostiže lopticu. Njoj pridružujemo skriptu `water.lua` koja upravlja kretnjama vode.

Objekt `platformGenerator` brine o nasumično generiranim platformama po kojima se igrač može penjati.

Zadnji objekt predstavlja kameru, a pridružujemo joj skriptu `camera.lua` koja prati igrača.

Skripta `water.lua` izgleda ovako:

```
function Start()
  ballTransform = GameObject.Find("ball"):GetComponent("Transform"):
    ToTransform()
  transform = gameObject:GetComponent("Transform"):ToTransform()
end
function Update()
  position = transform:GetPosition()
  if(position.y < ballTransform:GetPosition().y - 450) then
    transform:SetPosition(position.x, position.y + 3)
  elseif(position.y < ballTransform:GetPosition().y - 350) then
    transform:SetPosition(position.x, position.y + 2)
  elseif(position.y < ballTransform:GetPosition().y - 300) then
    transform:SetPosition(position.x, position.y + 1)
  else
```

```
14      os.exit()
15    end
16  end
```

U svakom koraku glavne petlje igre provjeravamo međusobne odnose pozicija loptice i igre. Što je loptica udaljenija od vode, to će se voda brže dizati. Ako voda dostigne lopticu, igra je gotova.

Skripta `camera.lua` gotovo je identična skripti `water.lua`. Jedina razlika je u zadnje tri linije funkcije `Update`. Skripta `generator.lua` nasumično stvara platforme. Dodatne platforme se stvaraju samo kada se loptica dovoljno približi zadnje generiranoj platformi. Funkcija `Update()` skripte `generator.lua` izgleda ovako:

```
1  function Update()
2    if(ballTransform:GetPosition().y + 500 > nextY) then
3      platform= GameObject.new()
4      platform:AddComponent("Sprite"):ToSprite():SetTexture("platform.png
       ")
5      platform:AddComponent("BoxCollider"):ToBoxCollider():SetSize(100 /
       2, 50 / 2)
6      platform:GetComponent("Transform"):ToTransform():SetPosition(nextX,
       nextY)
7      nextY = nextY + 100
8      if(nextX == 0) then
9        if(math.random(0, 1) == 0) then
10         nextX = 200;
11       else
12         nextX = -200;
13       end
14     else
15       nextX = 0;
16     end
17   end
18 end
```

Svaki puta kada stvorimo novu platformu, ažuriramo varijable `nextX` i `nextY` koje nam označavaju poziciju sljedeće platforme koje će eventualno biti stvorena. Ovdje možemo vidjeti i korištenje funkcije `math.random(...)` koja vraća nasumičan broj u zadanom intervalu. Što se tiče osi x, platforme stvaramo na koordinatama -200, 0 ili 200.

7.7 Planovi i ideje za budućnost

Implementacija ovog 2D pokretača igre i dalje nije potpuna, a neke komponente su neoptimizirane. U budućnosti plan je dodati nove komponente koje će proširiti mogućnosti pokretača igre, te optimizirati postojeće komponente.

Na primjer, svakako su potrebne komponente za izradu izbornika kao što su `Button` i `Text`. `Button` predstavlja objekt na koji se može kliknuti mišem, i tako odabrati odgovarajuću opciju izbornika. `Text` komponenta predstavlja niz znakova koji se će se iscrtavati na ekranu.

Osim njih, potrebno je dodati komponente koje će omogućiti korištenje zvuka unutar igre. Komponenta `AudioSource` predstavlja izvor zvuka unutar scene, a pridružuje joj se jedan zvučni isječak koji se može pokrenuti, zaustaviti ili pauzirati. Ova komponenta može se pridružiti proizvoljnom broju objekata. Osim nje, potrebna je i komponenta `AudioListener`, a ona predstavlja mikrofon unutar scene virtualnog svijeta. Ona brine o tome da se bliži zvukovi čuju jače, a dalji slabije. Ova komponenta obično se pridružuje samo jednom objektu unutar scene, najčešće glavnom liku u igri.

Opisivanje scena pomoću skripte `init.lua` je pomalo komplicirano za veće scene. Zbog toga je jedna od ideja za budućnost dodati podršku za opisivanje scena igre pomoću jezika XML. Naime, jezik XML je lako čitljiv i pruža mnoge mogućnosti koje olakšavaju stvaranje scene. Na primjer, moguće je hijerarhijski posložiti objekte, tako da se odmah vidi tko je čiji objekt roditelj i tko su čiji objekti djeca.

Bibliografija

[1] ESA - istraživanje o video igrama, http://www.theesa.com/about-esa/industry-facts/, posjećena srpanj 2017.

[2] J. Gregory, *Game Engine Architecture*, CEC Press, 2009.

[3] J. Haller, H. V. Hansson i A. Moreira, *SFML Game Development*, Packt publishing, 2013.

[4] J. S. Harbour, *Advanced 2D Game Development*, Course Technology, 2009.

[5] Članak o Teoremu o osi razdvajanja, https://gamedevelopment.tutsplus.com/tutorials/collision-detection-using-the-separating-axis-theorem--gamedev-169, posjećena srpanj 2017.

[6] Web stranica implementiranog 2D pokretača igre, https://github.com/matejorlic/2DGameEngine, posjećena srpanj 2017.

[7] Web stranice biblioteke Box2D, https://github.com/erincatto/Box2D, posjećena srpanj 2017.

[8] Web stranice biblioteke SFML, https://www.sfml-dev.org/, posjećena srpanj 2017.

[9] Web stranice biblioteke Sol, https://github.com/ThePhD/sol2, posjećena srpanj 2017.

[10] Web stranice programskog jezika Lua, https://www.lua.org/, posjećena srpanj 2017.

[11] Web stranice wikipedije o industriji video igara, https://en.wikipedia.org/wiki/Video_game_industry, posjećena srpanj 2017.

[12] Web stranice wikipedije o pokretaču igre, https://en.wikipedia.org/wiki/Game_engine, posjećena srpanj 2017.

[13] Web stranice wikipedije o povijesti video igara, https://en.wikipedia.org/wiki/History_of_video_games, posjećena srpanj 2017.

Sažetak

Kroz ovaj diplomski rad upoznali smo se sa osnovnim dijelovima pokretača igre, procesom razvoja igre i ostalim temama neposredno vezanim uz razvoj video igara. Nakon što smo se upoznali sa ulogom i arhitekturom pokretača igre, opisali smo dva veća sustava pokretača igre. Sustav za iscrtavanje koji je zadužen za iscrtavanje virtualnog svijeta igre na ekranu računala, te sustav za fiziku koji je zadužen za otkrivanje sudara između objekata unutar igre i za simulaciju njihove dinamike. Nakon toga, naveli smo glavne zadaće nekih manjih, ali i dalje bitnih podsustava pokretača igre, kao što su sustav za upravljanje memorijom, sustav za zvuk i skriptni jezik.

Na kraju smo predstavili primjer implementacije jednostavnog 2D pokretača igre napisanog u programskom jeziku C++. Kroz njegov razvoj upoznali smo se sa bibliotekama SFML i Box2D, kao i sa skriptnim programskim jezikom Lua koji omogućava brži i lakši razvoj igara.

Summary

Through this thesis we introduced the basic parts of a game engine, the game development process and some other topics related to the development of video games. After getting acquainted with the architecture of a game engine and its role, we have described two major game engine subsystems. Rendering system is used for rendering the virtual game world onto a computer screen, while the physics system is responsible for detecting collisions between objects within the game and for simulating their dynamics. After that, we highlighted the main tasks of some smaller, but stil essential game engine subsystems, such as memory management system, sound system, and scripting language.

At the end we presented an example of the implementation of a simple 2D game engine written in C++ programming language. Through its development we learned about SFML and Box2D libaries, as well as about the Lua scripting language that allows faster and easier game development.

Životopis

Osobni podaci:

- Prezime i ime: Orlić Matej
- Datum rođenja: 06. rujna 1993.
- Mjesto rođenja: Zagreb, Republika Hrvatska

Obrazovanje:

- 2015.-2017. Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet, Matematički odsjek, diplomski studij Računarstvo i matematika
- 2012.-2015. Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet, Matematički odsjek, preddiplomski studij Matematika, smjer nastavnički
- 2008.-2012. Gimnazija Velika Gorica, Velika Gorica
- 2000.-2008. Osnovna škola Vukovina, Vukovina

Zvanje:

- srpanj 2012. "Sveučilišni prvostupnik edukacije matematike", Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet, Matematički odsjek, preddiplomski studij Matematika, smjer nastavnički