Razvoj 3D video igre za utrkivanje automobilima

Markovica, Tomislav

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij

RAZVOJ 3D VIDEO IGRE ZA UTRKIVANJE AUTOMOBILIMA

Diplomski rad

Tomislav Markovica

Osijek, 2023.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

Obrazac D1: Obrazac za imenovanje Povjerenstva za diplomski ispit

Osijek, 12.09.2023.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za diplomski ispit

Ime i prezime Pristupnika:	Tomislav Markovica
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. Pristupnika, godina	D-1227R, 07.10.2021.
OIB studenta:	45676577963
Mentor:	doc. dr. sc. Tomislav Galba
Sumentor:	,
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	izv. prof. dr. sc. Časlav Livada
Član Povjerenstva 1:	doc. dr. sc. Tomislav Galba
Član Povjerenstva 2:	izv. prof. dr. sc. Alfonzo Baumgartner
Naslov diplomskog rada:	Razvoj 3D video igre za utrkivanje automobilima
Znanstvena grana diplomskog rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	Potrebno je izraditi 3D video igru za utrkivanje automobilima koristeći Unity programski okvir. Igra treba podržavati barem jedan tip igre (npr. single player) koji onda uključuje i razvoj jednostavnog računalnog protivnika. Napomena: tema rezervirana za Tomislav Markovica
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Izvrstan (5)

Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene od strane mentora:	12.09.2023.

Potvrda mentora o predaji konačne verzije rada:	*Mentor elektronički potpisao predaju konačne verzije.*
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

IZJAVA O ORIGINALNOSTI RADA

Osijek, 25.09.2023.

Ime i prezime studenta:	Tomislav Markovica
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-1227R, 07.10.2021.
Turnitin podudaranje [%]:	2

Ovom izjavom izjavljujem da je rad pod nazivom**: Razvoj 3D video igre za utrkivanje automobilima**

izrađen pod vodstvom mentora doc. dr. sc. Tomislav Galba

i sumentora ,

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SADRŽAJ

1. UVOD 1

1.1. Zadatak diplomskog rada 2

2. PREGLED PODRUČJA I TEME 3

2.1. Što je Game Engine 3

2.2. Unity 3

2.3. Kulturalni softver 4

2.4. Igre utrkivanja 6

3. ARHITEKTURA GAME ENGINE-A 7

3.1. Arhitektura 7

3.2. Hardver, pogonski programi i operacijski sustav 7

3.3. Vanjske biblioteke 8

3.4. Sloj neovisnosti o platformi 8

3.4.1. Datotečni sustavi 9

3.5. Core Systems 10

3.6. Upravljanje resursima 10

3.7. Funkcionalnosti više razine 11

3.8. Game Loop 11

4. IMPLEMENTACIJA IGRE 13

4.1. Dizajn igre 13

4.1.1. Scena izbornika vozila 13

4.1.2. Scena utrke 14

4.2. Cinemachine – paket kamera 16

4.2.1. Cinemachine Virtual Camera 16
4.2.2. Cinemachine Brain 19
4.3. Model vožnje 20
4.3.1. Upravljanje vozilom 21
4.3.2. Utvrđivanje orijentacije skalarnim umnoškom 29
4.4. Kretanje protivnika 30
4.4.1. Implementacija protivnika 32
4.5. Prilagodba brzine prema teksturi terena 36
5. ZAKLJUČAK 40
LITERATURA 41
POPIS SLIKA 43
SAŽETAK 45
ABSTRACT 46
ŽIVOTOPIS 47
PRILOG 48

1. UVOD

Industrija video igara je jedna od glavnih stimulatora za razvoj računalne tehnologije, što proizlazi iz želje za unaprjeđenjem korisničkog iskustva, ugođaja igranja i grafike. Sredinom 1990-ih godina počinje se upotrebljavati termin *game engine* koji označava softverski okvir za razvoj video igara [1]. *Game engine*-i su revolucionirali industriju, postali su temelj za razvoj igara pružajući programerima potrebne resurse i alate te značajno umanjili vrijeme i troškove razvoja. Isto tako svaki razvojni okvir dolazi sa svojim prednostima i nedostacima iz čijih razlika proizlaze razni žanrovi i stilovi igara. Mnogobrojni razvojni okviri optimizirani su za stvaranje različitih tipova sadržaja i imaju nešto drugačije procese razvoja. Odabir odgovarajućeg G*ame engine*-a obavlja se prema značajkama i učinkovitosti njegovih pojedinih komponenata: okvira fizike (engl. *Physics Engine*), okvira iscrtavanja (engl. *Rendering Engine*), jezika za pisanje skripti (engl. *Scripting Language*), biblioteka imovina (engl. *Asset Libraries*) - (modeli, teksture, animacije) i korisničkog sučelja (engl. *User Interface*). Okviri su zaduženi da brinu o tim zadacima na nižem nivou: iscrtavanjem slike, fizikom objekata i umjetnom inteligencijom. To je glavni cilj namjene razvojnih okvira. Viša razina okvira tada postaje platforma koja korisnicima okvira, tj. developerima omogućuje veći fokus na dizajn i pruža im hrpu alata za ostvarivanje njihovih ideja.

Praktični dio diplomskog rada je izrada 3D video igre u kojoj se utrkuje automobilima, a u narednim poglavljima ovoga rada biti će prezentirani razvoj te igre i općenito kako se odvija unutarnji rad i arhitektura *Game Engine*-a. Za implementaciju igre utrkivanja koristit će se Unity Game Engine i C# programski jezik za pisanje koda. Unity se pokazao kao vrlo zanimljiv alat u kojem je moguće stvaranja „novih svjetova“, a uz to je besplatan za korištenje te se na internetu može pronaći hrpa gotovih besplatnih modela. To su neki od razloga odabira Unity-a za implementaciju igre.

Drugo poglavlje ukratko objašnjava što je *game engine*, iznosi njegove značajke i neka područja primjene specifično za Unity Game Engine koji je korišten za implementaciju igre ovog diplomskog rada. Zatim će biti pojašnjen pojam kulturalni softver, kako softverski alat može značajno oblikovati kulturu industrije ili grupe ljudi, postaviti trendove ponašanja, percepcije i razmišljanja. Softverski alat može prouzročiti usmjeravanje ideja isto kao što se to naviklo viđati u likovnoj umjetnosti, glazbi, filmovima, itd.

Treće poglavlje će navesti i ukratko opisati sastavne dijelove *game engine*-a. Objasnit će kako je arhitektura načinjena od slojeva i koje komponente se nalaze na pojedinim slojevima.

Četvrto poglavlje će detaljnije opisati dizajn igre, mehanike i implementaciju. Pokazati će izgled igre, stazu te model vozila. Opisivat će se kako su izvedeni model vožnje i računalni protivnik te će biti priložen programski kod. Dio o računalnom protivniku pojašnjava način na koji su kontrolne točke iskorištene kako bi se postiglo da protivnik ima sposobnost prilagođavati svoju putanju prema stazi. Na kraju je opisano kako je postignuta mehanika prilagođavanja brzine gdje se prema teksturi terena određuje na kojoj vrsti terena se vozilo nalazi.

1.1. Zadatak diplomskog rada

Potrebno je izraditi 3D video igru za utrkivanje automobilima koristeći Unity programski okvir. Igra je zamišljena da podržava igranje samo jednog igrača, to onda uključuje i razvoj jednostavnog računalnog protivnika.

2. PREGLED PODRUČJA I TEME

2.1. Što je Game Engine

Game engine prema je softverski alat koji omogućuje stvaranje digitalnog sadržaja i okvir za kodiranje sadržaja kojeg je moguće pokretati na različitim platformama [1, 2, 3]. To su alati na kojima se gradi sadržaj i logika igre. *Game engine* odrađuje komunikaciju s operacijskim sustavom, brine se o alociranju memorije, rukuje s korisničkim unosima i iscrtava piksele na ekran. Ovakvi alati se uglavnom povezuju sa industrijom video igara, no granica primjene ovakvog softvera je samo kreativnost korisnika. Ostale primjene: vizualizacija, 3D animacije, arhitektonski modeli, složene simulacije, edukacija. Neki popularni game *engine*-i koji su javno dostupni: Unreal Engine (vlasnik Epic Games), Unity (vlasnik Unity Technologies), GameMaker Studio (vlasnik Yoyo Games), Godot (besplatan i otvorenog koda). Neki popularni vlasnički (engl. *Proprietary*) *game engine*-i – to znači da su u vlasništvu kompanija samo za njihovo korištenje te nisu dostupni ostalima: Frostbite (vlasnik Electronic Arts), Anvil (vlasnik Ubisoft), id tech (vlasnik id Software), Source (vlasnik Valve), Fox engine (vlasnik Konami Digital Entertainment).

2.2. Unity

Za izradu praktičnog dijela ovoga rada kao razvojni okvir je izabran Unity Game Engine. To je okvir koji je zadobio veliku popularnost time što nudi podršku razvoja za mnoštvo platformi: iOS, Android, Windows, Universal Windows Platform, Linux, WebGL, PS4, PS5, XBOX ONE, Oculus Rift, AndroidTV, tvOS, Nintendo Switch, ARCore, Microsoft HoloLens, Magic Leap. [4] Nudi mogućnost razvoja 2D, 3D, VR i AR aplikacija. Dodatno, ima vlastitu trgovinu imovinama (engl. *Asset Store*) što doprinosi jednostavnosti razvoja i čuvanju budžeta prilikom razvoja. Unity je jeftin (besplatna licenca za sve korisnike sa prihodom ili financijama do 100 000 USD u posljednjih 12 mjeseci) i relativno jednostavan za korištenje, ali istovremeno itekako sposoban za profesionalnu primjenu. Direktor kompanije Unity Technologies, John Riccitiello, iznosi da je polovica svih video igara i projekata virtualne stvarnosti na suvremenim uređajima razvijena u Unity-u [5].

Kao primjeri stvarne primjere korištenja Unity-a izvan samih video igara mogu se navesti:

- NASA: *Eyes on the Solar System* – alat za vizualizaciju Sunčevog sustava. [6]

- BMW: *Efficient development of simulated environments for autonomous vehicle training* – BMW grupa koristi Unity da razvije grafički uređivač scenarija za pojednostavljenje procesa testiranja i validacije značajki autonomne vožnje. Developerima pruža sučelje, vizualizaciju i postavljanje tisuća scenarija kojima se povećava zrelost i spremnost značajki autonomne vožnje. Skoro 95% svih BMW-ovih testnih kilometara autonomne vožnje jesu odvoženi virtualnim vozilima u virtualnim svjetovima. [7]

2.3. Kulturalni softver

Kulturalni softver (engl. *Cultural Software*) je izraz koji opisuje zajednička vjerovanja, vrijednosti i prakse koje oblikuju razmišljanja grupe ljudi [1,8]. Može se na to gledati kao programski jezik kojim se utječe na ljudski um - utjecaj na način percepcije svijeta, razmišljanja i ponašanja. Kulturalni softver postavlja strukturu shvaćanja i razmišljanja određenih koncepata prema kojima se oblikuju ideje. Može se dogoditi da samo pojedini dijelovi softvera ostave toliko značajan utjecaj na ljude i oblikuju čitavu kulturu ponašanja, razmišljanja ili navika. Utjecaj može otići toliko daleko da dio softvera ili pak samo neke prakse postanu sastavni dio tehnološke pismenosti – što bi se odnosilo na sposobnost korištenja, shvaćanja i upravljanja tehnologijom na siguran, efektivan i odgovoran način [8]. Utjecaj kulture može obuhvatiti sve od dizajna sučelja, korisničkog iskustva, procesa razvoja do favoriziranja određenih softverskih rješenja ili softverskih alata prije nekih drugih alata za istu svrhu korištenja. Unity je jedan takav softver koji je uspio oblikovati cijelu kulturu razvoja igara i ostaviti utjecaj na milione ljudi. Kao još neki primjeri za kulturalni softver mogu se navesti tzv. medijska sučelja: ikonice, mape, zvukovi, animacije, ekrani osjetljivi na dodir. Spomenuta sučelja vrše interakciju s medijima gdje ona ukazuju na namjenu aplikacija i usmjeravaju kretanje po aplikacijama, pobliže tumače programski kod koji će se izvoditi.

Igre se često promoviraju sa svojim *engine*-om na kojem se pokreću. Igrači su tako stvorili asocijacije prema kompanijama i njihovim *game engine*-ima, asocijacije kao npr. dobra grafika, stabilan *frame rate*. Nekad je to bilo još bolje isticano nego danas. Danas, u novije doba sve se više developera odlučuje na korištenje komercijalnih *game engine*-a, stoga isticanje *engine*-a u kojem se igra pokreće nosi manji značaj jer potencijalno već postoje stotine igara razvijenih u tome istome. Proces razvoja alata specifično za namjene kompanije je postao neisplativ, to je jedino isplativo kompanijama sa AAA naslovima radi specifičnih zahtjeva i performansi. Razvoj takvih alata može trajati i godinama, uz dodatne potrebe održavanja da bi radili i sa novim

sklopovljem. Razvoj okvira je izrazito spor i zahtjeva gomilu resursa, iz tog razloga kompanije su vrlo rijetko dijelile alate u kojima su razvijale svoje igre. Ipak, početkom 2000-ih počeli su se polako pojavljivati jeftini razvojni okviri koji su pružili prilike još većoj skupini kreatora da se uključe u proizvodnju. Takva komercijalna rješenja *game engine*-a sebi su stvorila poslovni model i način licenciranja kojim kreator može zaobići programiranje svojih alata za razvoj i sav fokus razvoja može usmjeriti na sadržaj igre. Sadržaj se odnosi na prezentacijske elemente kao što su: imovina igre, animacije, zvuk, scenarij, dizajn nivoa. Kako je razvoj postao usmjeren na sadržaj, *content-centric development*, glavnina razvoja se preusmjerila na dizajn i umjetnost te se uz programere pojavila još veća potražnja za dizajnerima, piscima, skladateljima, producentima, itd. Unity je ovdje odigrao vrlo važnu ulogu u prelasku na razvoj usmjeren prema sadržaju te je time postao istaknut kulturalni softver u razvoju video igara. Unity je proizvod na kojem svakodnevno rade tisuće inženjera, proizvod koji ciljano želi postići: „demokratizaciju razvoja video igara“, što je i slogan kompanije Unity Technologies. Uspjeli su time stvoriti široku zajednicu korisnika koji održavaju alat pristupačnim na način da objavljuju svoje modele u trgovini imovina te dijele znanja i savjete na internetu. Unity podržava razvoj igara za stvarno veliki broj platformi, a među vodećim je alatima u industriji po broju različitih medijskih tipova datoteka koje podržava. Kompatibilan je s velikim brojem trenutnih softvera za izradu modela, tj. *game assets*-a.

Uz sve prednosti vezane za inovacije *game engine*-a, profesionalci iz industrije iznose opažanja i sa suprotne strane. Iznose kako studenti ne uče „pravi“ razvoj igara već uče kako koristiti Unity. Ova izjava zapravo nije nikakva osuda, a ni preporuka, izjava ukazuje na to kako je današnji razvoj igara i okvira usredotočen na dizajn te je oslobođen briga nižih razina razvoja softvera. Drugo, dostupnost alata kao što je Unity stvara prezasićenost tržišta nekvalitetnim igrama i manjak programerskih vještina. Mali broj jako dominantnih kompanija ili platformi kontrolira kulturu proizvodnje to nazivaju homogenizacija i racionalizacija proizvodnje video igara. To znači da unatoč tome što se radi o igrama s drugačije zamišljenim mehanikama i ugođaju mogu se pojaviti sličnosti među njima jer se grade na istim temeljima. Homogenizacija se odnosi na raznolikost i kreativnost u industriji jer se na igrama koje su izgrađene u istom okviru mogu pojaviti slična korisnička sučelja i mehanike igranja. Pojam racionalizacije proizvodnje video igara se odnosi na takvu proizvodnju gdje je želja postići trenutne tržišne i industrijske standarde umjesto toga da se izrazi kreativnost i originalnost. To su izazovi malih timova, pitanja financija i samog profita.

2.4. Igre utrkivanja

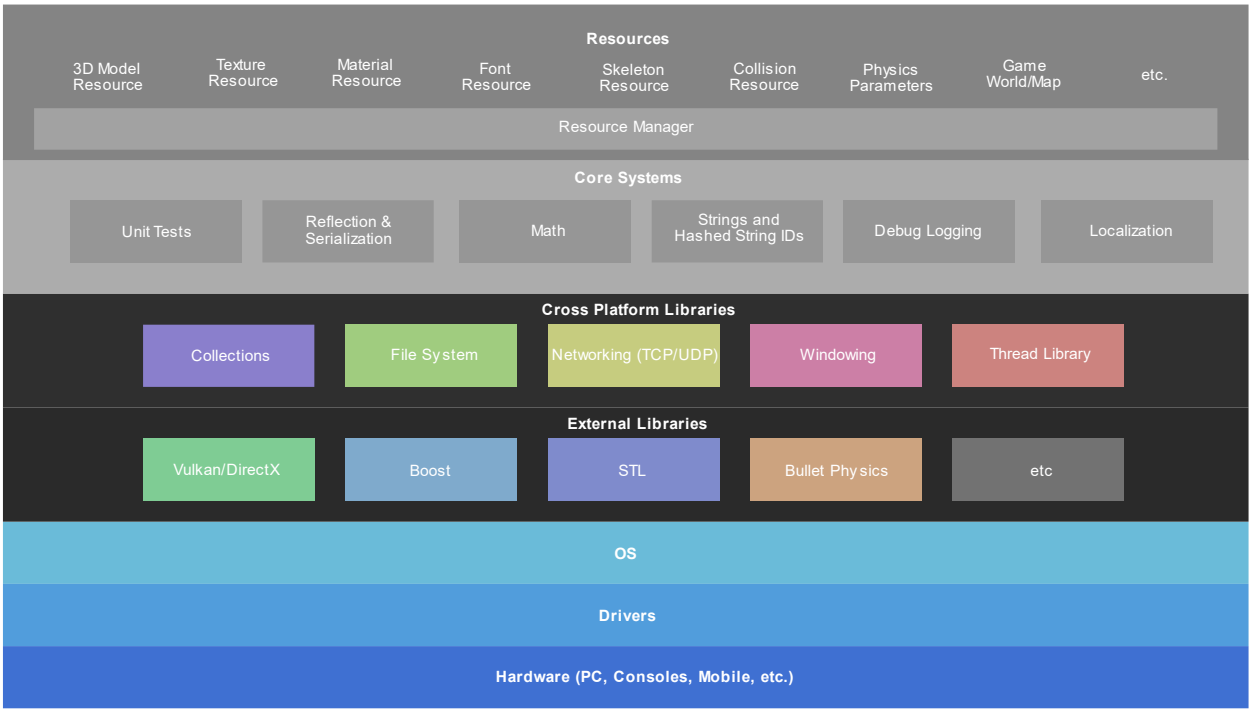
Jason Gregory odgovorio je na pitanje što je igra [3]. Piše kako se u računarstvu video igre može zvati sustavima stvarnog vremena, obrazlaže da su video igre preciznije i opširno rečeno, *soft real-time interactive agent-based computer simulations*. Simulacije u izrazu znači da se radi o matematičkom modelu kojim se želi predstaviti neki zamišljeni ili dio stvarnog svijeta. Zatim, sustav s ublaženim vremenskim zahtjevima znači da nema sigurnosnih posljedica za korisnika. Cilj je ispuniti zahtjeve obrade kako bi se slika stigla iscrtavati i postigla fluidnost igranja. Agenti, u *agent-based* izrazu odnose se na entitete u simulaciji koji međusobno surađuju, a to bi jednostavno bili objekti u igri: vozila, likovi, metci, itd. Interaktivni dio simulacije jesu interakcije s agentima, a te se interakcije događaju u stvarnome vremenu. Želja je virtualne svjetove napraviti dovoljno dinamičnima kako bi uspijevale savladavati nepredvidivost korisnika. Zadaća igre je stalno očekivati i reagirati na korisnikov unos naredbi. Manje dinamične simulacija popu šaha ili strateških igara mogu se isto tako svrstati pod simulacije stvarnog vremena prema tome što se i dalje očekuje da korisničko sučelje reagira u stvarnome vremenu.

Igre utrkivanja se dijele na još dvije potkategorije, a to su simulacije i arkade [9]. Simulacije pokušavaju pružiti što je moguće realističniji model vožnje, neki primjeri takvih igara su: Gran Turismo, Dirt, Project Cars… Karakteristično, simulacijske igre utrkivanja zahtijevaju pravilnu tehniku vožnje, prilagođavanje brzine i precizne manevre, a isto tako i zvuk vozila može odigrati bitnu ulogu gdje zvuk signalizira igraču ponašanje automobila. Dok, arkadne igre pokušavaju postići što zabavniji model igranja i ne obaziru se na realizam vožnje. U arkadama se često može dogoditi da se pravovremenim sudaranjem u zidove ili druge igrače dobije na brzini umjesto očekivanog usporavanja ili kažnjavanja igrača na neki način. Ovdje se ističe Need For Speed serijal kao možda najznačajniji serijal igara arkadnog utrkivanja, a još neki primjeri arkada su: Driveclub, TrackMania, Burnout, The Crew…

3. ARHITEKTURA GAME ENGINE-A

3.1. Arhitektura

Game engine je kompleksan softver koji je sačinjen od mnogo međusobno povezanih podsustava. Kao i svi softverski sustavi *engine*-i se grade u slojevima gdje viši slojevi ovise o nižim slojevima pa se time uklanjaju ovisnosti između podsustava. Arhitektura *game engine*-a je pokazana na slici 3.1.



Slika 3.1. Arhitektura *game engine*-a [10]

3.2. Hardver, pogonski programi i operacijski sustav

Sloj hardvera je temelj računalnog sustava. Hardver je obično različit za svaku platformu - osobna računala, igrače konzole, mobilne uređaje, itd. Hardverom upravljaju pogonski ili upravljački programi (engl. *Device Drivers*). Oni upravljaju specifičnim značajkama pojedinog hardvera tako da pružaju softversko sučelje kojim operacijski sustav prenosi instrukcije hardveru. Operacijski sustav se zatim brine o procesima i memoriji računala.

3.3. Vanjske biblioteke

Kako bi se stvorila neovisnost o platformi, zadaća ovog sloja je pružiti sučelje višim slojevima okvira. Sloj se sastoji od dva segmenta. Prvi segment je briga o poslovima niske razine okvira kao što su iscrtavanje grafike i računanje fizike. Ovo se odrađuje implementacijom gotovih paketa trećih strana koji rješavaju takve poslove – paket se odnosi na SDK (engl. *Software Development Kit*). Zadaća drugog segmenta je postići apstrakciju i pružiti sučelje koje će rukovati operacijskim sustavom zbog specifičnih funkcionalnosti okvira.

Pri razvoju *game engine*-a uobičajeno je koristi se nizom gotovih paketa trećih strana koji su već testirani i pružaju naprednije funkcionalnosti koje bi bilo prezahtjevno razvijati od nule. Ovime se pažnja može preusmjeriti na višu razinu dizajna okvira i inovacije. Korištenje gotovih alata pruža stabilnost i može osigurati bolju kompatibilnost s raznim platformama. SDK-ovi se koriste za:

- Grafiku – služi za renderiranje grafike, pruža API koji se koristi za interakciju s GPU-om. Primjeri API-ja za grafiku: Glide, OpenGL, DirectX, libgcm, Edge.
- Kolizije i fiziku – postoje paketi koji pružaju gotove okvire za izračune fizike, realističnog kretanja i kolizija. Primjeri: Havok, PhysX, Open Dynamics Engine.
- Animacije – ovo su paketi s alatima za upravljanje animacijama. Primjeri: Granny, Havok Animation, Edge.

3.4. Sloj neovisnosti o platformi

Često se događa da kompanije prilikom izdavanja igre ciljaju obuhvatiti što veće tržište, napraviti igru da je podržana na što više platformi. U ovom segmentu se također događa da ovisi o softverskim rješenjima treće strane, pa se prema tome ovaj sloj u ilustraciji arhitekture nalazi povrh segmenta vanjskih biblioteka (slika 3.1.). Ovaj segment okviru stvara razinu apstrakcije nad sustavnim pozivima operacijskog sustava i ima zadaću osigurati ispravnost tih poziva za različite platforme.

Moduli koji se pojavljuju u implementacijama okvira potrebni za postizanje neovisnosti o platformi:

- Detekcija platforme.

- Kolekcije i iteratori – okviri često imaju vlastite implementacije uobičajenih struktura podataka, ova praksa prevladava kod konzola. Razlozi korištenja: kontrola nad memorijom, prilika za optimizacijom, mogu se ugraditi specifično željeni algoritmi, eliminacija vanjskih ovisnosti.
- Networking biblioteke – rješenja komunikaciju multiplayer igre, modeli komunikacije koji se korite: klijent-server i peer-to-peer.
- Thread Library – pruža podršku za višenitnost, klase za stvaranje, upravljanje i sinkronizaciju nitima. Ova biblioteka apstrahira mehanizme operacijskog sustava koji upravljaju nitima.
- Graphics Wrapper – služi za renderiranje grafike, upravljanje teksturama, sjenama i modelima bez znanja o detaljima grafičkih API-ja. Ovime se stvara razina apstrakcije čime se ostvaruje mogućnost jednostavnog prebacivanja projekta na drugu platformu koja koristi neki drugi API za iscrtavanje slike.
- Physics Wrapper – mogu se pojaviti i omotači za fiziku.
- Datotečni sustavi.

3.4.1. Datotečni sustavi

Game engine mora biti sposoban upravljati raznim vrstama i formatima medijskih datoteka – bitmape, 3D mesh podaci, animacije, zvučne datoteke, itd. Glavna svrha je omogućiti učitavanje i pristup tim resursima, a isto tako okvir mora biti efikasan pri rukovanju memorijom. Zbog čestog korištenja ili možda nedostatka alata datotečnog sustava, obično je upravljanje datotekama riješeno načinom da se API izvornog datotečnog sustava omota u API *game engine*-a.

Datotečni sustav *game engine*-a tipično rješava slijedeće:

- Upravljanje imenima datoteka i putanjama jer obično se pravila mogu razlikovati između operacijskih sustav.
- Otvaranje, zatvaranje, čitanje i pisanje datotekama.
- Pretraživanje sadržaja direktorija.

- Rukovanje asinkronim I/O zahtjevima, ovo se odnosi na mogućnost učitavanja podataka u pozadini dok se glavni program i dalje izvodi.

3.5. Core Systems

Na ovom sloju se nalaze alati za razvoj, testiranje i konfiguriranje aplikacije. Nekoliko modula jezgrinih sustava:

- Upravljanje memorijom – svaki okvir implementira svoj vlastiti sustav alokacije memorije, da se osigura brzina i ograniče negativni učinci fragmentacije memorije.
- Biblioteku matematike – ovakve biblioteke pružaju matematičke alate koje bi programeri mogli trebati - za igre su to vektori, matrice, kvaternioni, geometrija, trigonometrija, rješavanje sustava jednadžbi…
- Prilagođene podatkovne strukture i algoritmi – skup alata za upravljanje temeljnim strukturama podataka (povezani popisi, dinamička polja, binarna stabla, hash mape, …) i algoritmima (pretraživanje, sortiranje, …). Za ovo se mogu koristiti i paketi treće strane, a mogu se izrađivati ručno da bi se smanjilo ili eliminiralo dinamičku alokaciju memorije i osiguralo optimalne performanse za ciljanu platformu.
- Raščlanjivači različitih formata (engl. *Parsers*) - JSON, CSV, XML, …
- Jedinično (Unit) testiranje.
- Otklanjanje grešaka i ispis.

3.6. Upravljanje resursima

Upravitelj resursa (engl. *Resource Manager*) je sloj u arhitekturi zadužen za efikasno učitavanje i korištenje različitih resursa igre. Upravljanje resursima podrazumijeva da ima sljedeće mogućnosti: alate koji se koriste da se resurse može stvarati i upravljanje resursima za vrijeme dok se igra izvodi. Upravitelj mora pružiti unificirano sučelje za pristup različitim vrstama podataka, no okvir ne mora te sve formate striktno koristiti u njihovom izvornom obliku. Obično postoji implementacija nekog oblika cjevovoda kojim se resursi mogu konvertirati.

Zadaci koje upravitelj resursima treba obavljati u tijeku izvođenja igre:

- U memoriji osigurati postojanje samo jedne kopije od svakog jedinstvenog resursa.
- Brinuti se o životnom ciklusu svih učitanih resursa.
- Upravlja korištenjem memorije, učitavanjem i uklanjanjem podataka iz memorije.
- Rukovoditi kompozitnim resursima – resursi sastavljeni od drugih resursa, primjerice 3D model koji može imati kao dodatke: mesh, materijal, teksturu, skeleton, animaciju, itd.

Dizajn podsustava za alokaciju memorije je osmišljen da vodi računa i o lokaciji gdje će se resursi smjestiti u memoriju nakon učitavanja. Resursi poput tekstura i shader-a će vjerojatno biti smješteni u video RAM, dok će se ostali resursi smjestiti u RAM. Primarni problem s kojim se suočava sustav upravljanja resursima je fragmentacija memorije, a neke tehnike izbjegavanja fragmentacije su heap-based alokacija i pool-based alokacija.

3.7. Funkcionalnosti više razine

Viši slojevi *game engine*-a obuhvaćaju iscrtavanje, animacije, uređaje za unos, zvuk, sustav skriptiranja. Iscrtavanje predstavlja završni proces stvaranja slika ili animacija iz modela. Ovo zahtijeva izgradnju sučelja prema grafičkim uređajima korištenjem grafičkih SDK-ova poput DirectX-a ili OpenGL-a. Nakon toga se polazi od stvaranja primitivnih oblika do stvaranja kompleksnih modela na što efikasniji i vizualno ljepši način moguće. Povrh sloja iscrtavanja nalazi se sloj za vizualne efekte koji se brine o svijetlu, sjenama, vatri, dimu, vodi, itd.

3.8. Game Loop

U središtu *game engine*-a postoji glavna petlja iz koje se pozivaju svi ostali podsustavi. U petlji je nakon hvatanja unosa naredbi potrebno ažurirati entitete na sceni i potom iscrtati sliku na ekran. Iscrtavanje i animacije trebale bi se pozivati s učestalošću 30 – 60 Hz. Dok sustavi više razine kao što je sustav umjetne inteligencije mogu zahtijevati i nešto manju učestalost pozivanja, ako nije potrebno da bude sinkroniziran s iscrtavanjem. Temeljna načela središnjih petlji *game engine*-a ostaju ne promijenjena. Središnje petlje modernih razvojnih okvira jesu vjerojatno mnogo sofisticiranije, ali temeljni principi korištenja i dalje ostaju isti (slika 3.2.). Razlike postoje u trenutcima i načinima pozivanja specifičnih podsustava igre, što može usmjeriti i namjenu okvira za određeni žanr igara.

```
while (true)
{
    Capture();
    Update();
    Render();
}
```

Slika 3.2. Središnja petlja *game engine*-a

Implementacija središnje petlje Unity-a je skrivena od korisnika. Kontrola vremena pozivanja pružena je preko API-ja za skriptanje nasljeđivanjem MonoBehaviour klase, a to su metode: Update, FixedUpdate, LateUpdate.

4. IMPLEMENTACIJA IGRE

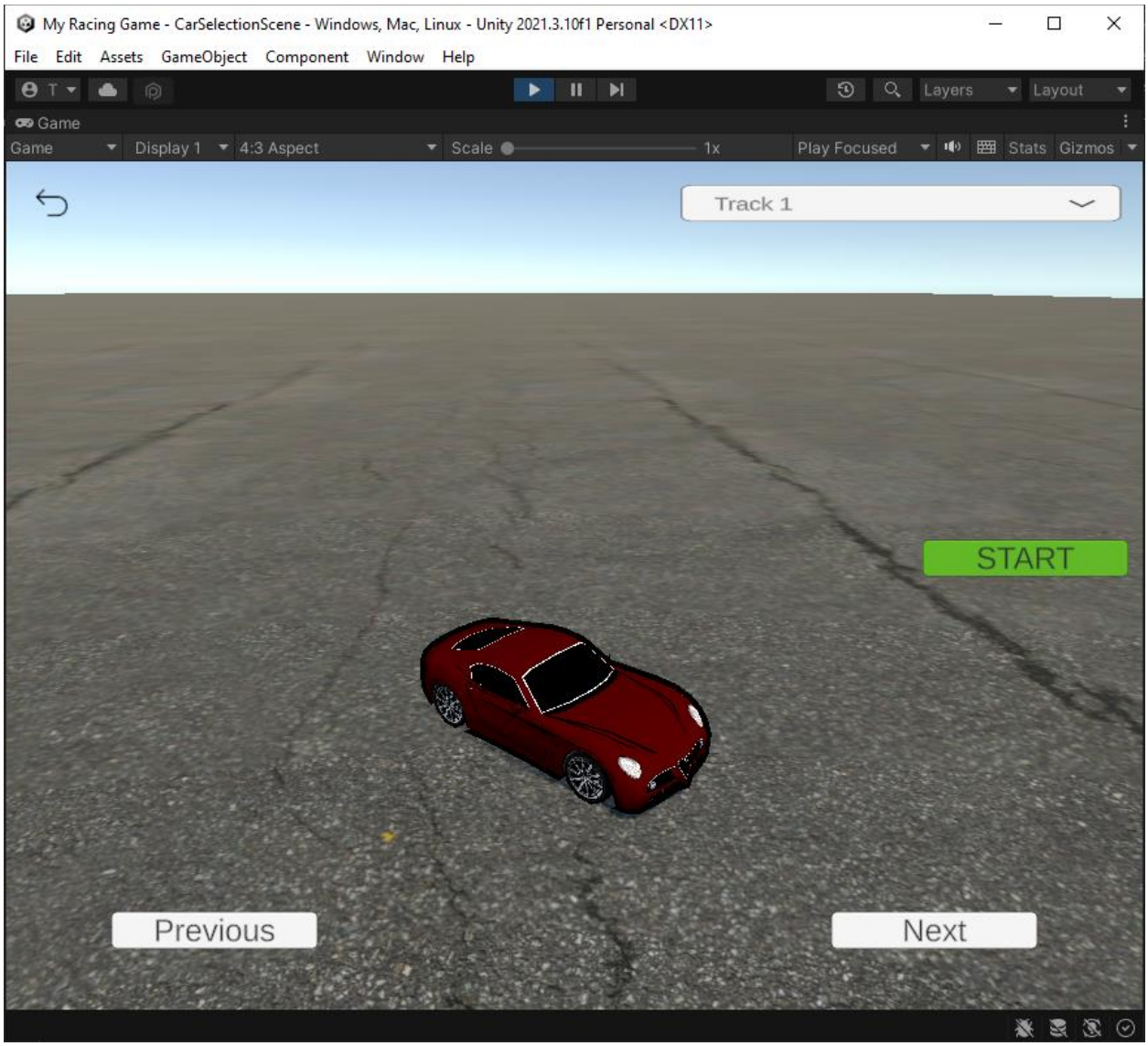
4.1. Dizajn igre

Za razvoj igre korišten je Unity Engine (verzija 2021.3.10f1) [12, 13]. Igra je pisana u C# programskom jeziku i VS Code je korišten za uređivanje programskog koda.

Za dizajn igre iskorišteni su gotovi besplatni modeli dostupni sa službene trgovine Unity Asset Store. U prilogu dokumenta nalaze se sve poveznice na pakete koji su uključeni u projekt: modeli automobila, teksture, zvučni efekti, te svi ostali objekti i predmeti na stazi. Nadalje, Unity Editor ima ugrađen set alata za uređivanje terena koji pruža mogućnosti oblikovanja reljefa, bojanje terena, dodavanje drveća, trave i kamenja. Dizajn ceste na stazi u principu se svodi na bojanje površine terena željenom teksturom, a zapravo je svaka tekstura zasebni sloj terena. Slojevi terena se mogu koristiti i za stvaranje dinamičnijeg iskustva vožnje gdje se svakoj vrsti podloge mogu se dodijeliti drugačije karakteristike vožnje tom podlogom. U ovom projektu konkretno, koriste se dvije vrste terena - sloj ceste i sloj zemlje. Sloj zemlje još služi kako bi se igrača kaznilo za vožnju izvan zamišljene ceste.

4.1.1. Scena izbornika vozila

Prva scena igre zamišljena je kao zanimljiviji pristup sučelju izbornika u kojem se odabire automobil i staza, prikazano na slici 4.1. Staza se bira u padajućem izborniku, a automobili se listaju jedan po jedan. Za rotaciju voznog parka postoji lista referenci na objekte automobila. Prebacivanje između vozila izvedeno je na način da se trenutni automobil na sceni uništava i instancira se idući u nizu. Nakon listanja automobila klikom na start zaključava se odabir vozila te igra prelazi na odabranu stazu. Svaka staza je zasebna scena. Prelaskom na novu scenu potrebno je prenijeti i saznanje o odabranom automobilu iz prošle scene kako bi se znalo taj isti instancirati za utrku. Kada igrač odabere automobil, indeks tog objekta iz liste se sprema u preferencijske postavke igrača (PlayerPrefs). Pohrana funkcionira u obliku ključ-vrijednost.



Slika 4.1. Slika zaslona scene za odabir auta

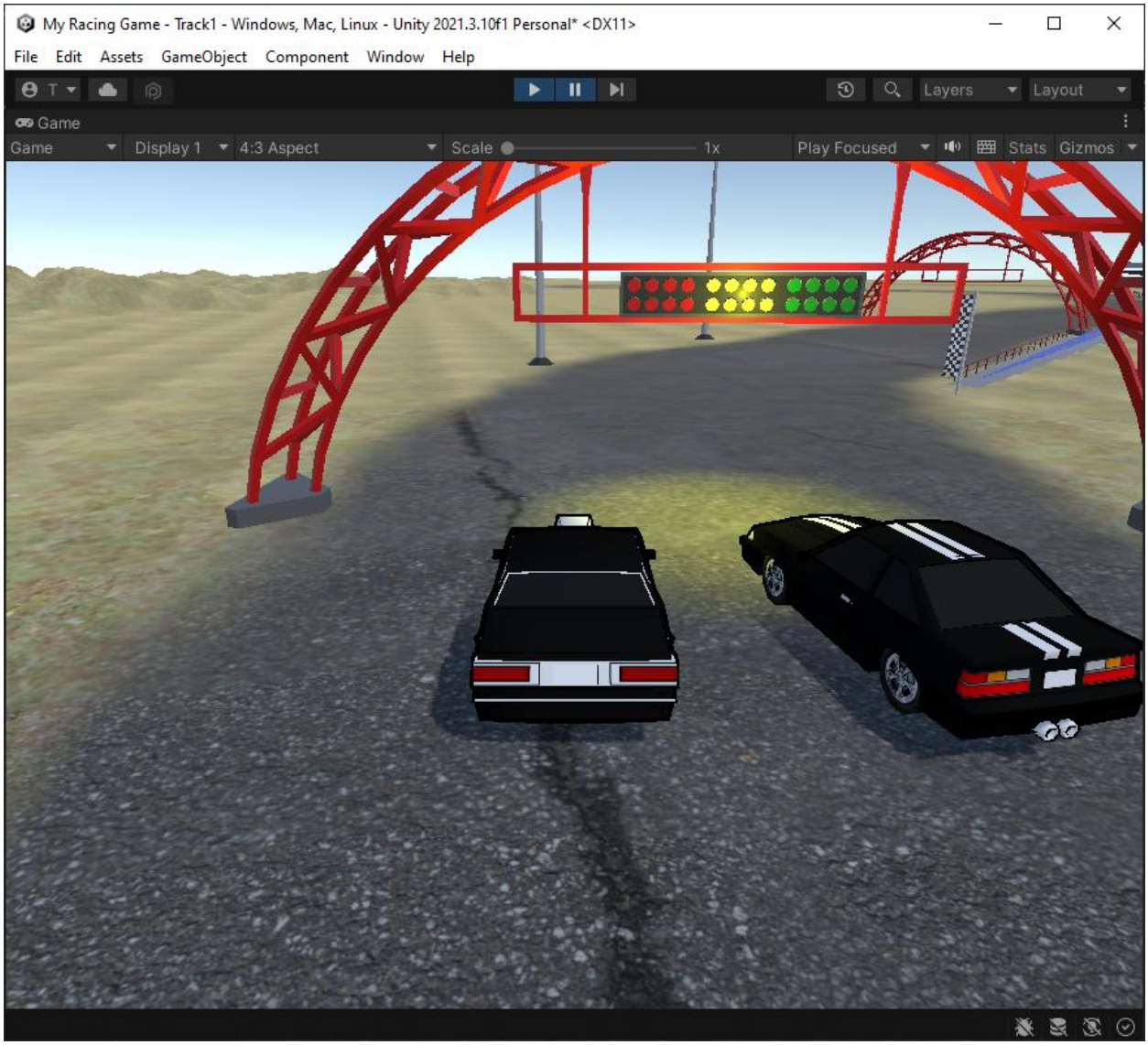
4.1.2. Scena utrke

Na slici 4.2. se vidi slika zaslona igre na startnoj liniji. Automobile se na ovu scenu stvara dinamički iz razloga što se odabir vozila prenosi iz prošle scene, a to je s ciljem da bi se postigla raznolikost protivnika (samo vizualno, vožnja je ista za sve). Model automobila za računalnog protivnika odabire se nasumično, svaki puta kada se pokreće utrka.

U igri je implementiran semafor da bi se pratilo odbrojavanje za početak utrke. Služi da vizualno i zvučno signalizira početak utrke. Pozadinska misija toga je da obavijesti skripte sudionika o početku utrke. Skripte za vožnju inicijalno su deaktivirane kako bi vožnja bila

onemogućena za vrijeme odbrojavanja. Tek nakon što se odbroji za početak utrke pristupa se objektima vozila i skripte se aktiviraju.

U igri je još implementiran jednostavni računalni protivnik protiv kojeg se utrkuje. Implementiran je na način da su duž cijele staze postavljene kontrolne točke. Na taj način protivnik redom prolazi svim točkama, a smjer iduće doznaje tek nakon što je prošao kroz prethodnu točku. Detaljnije objašnjenje implementacije protivnika biti će u kasnijem potpoglavlju.



Slika 4.2. Snimka zaslona igre

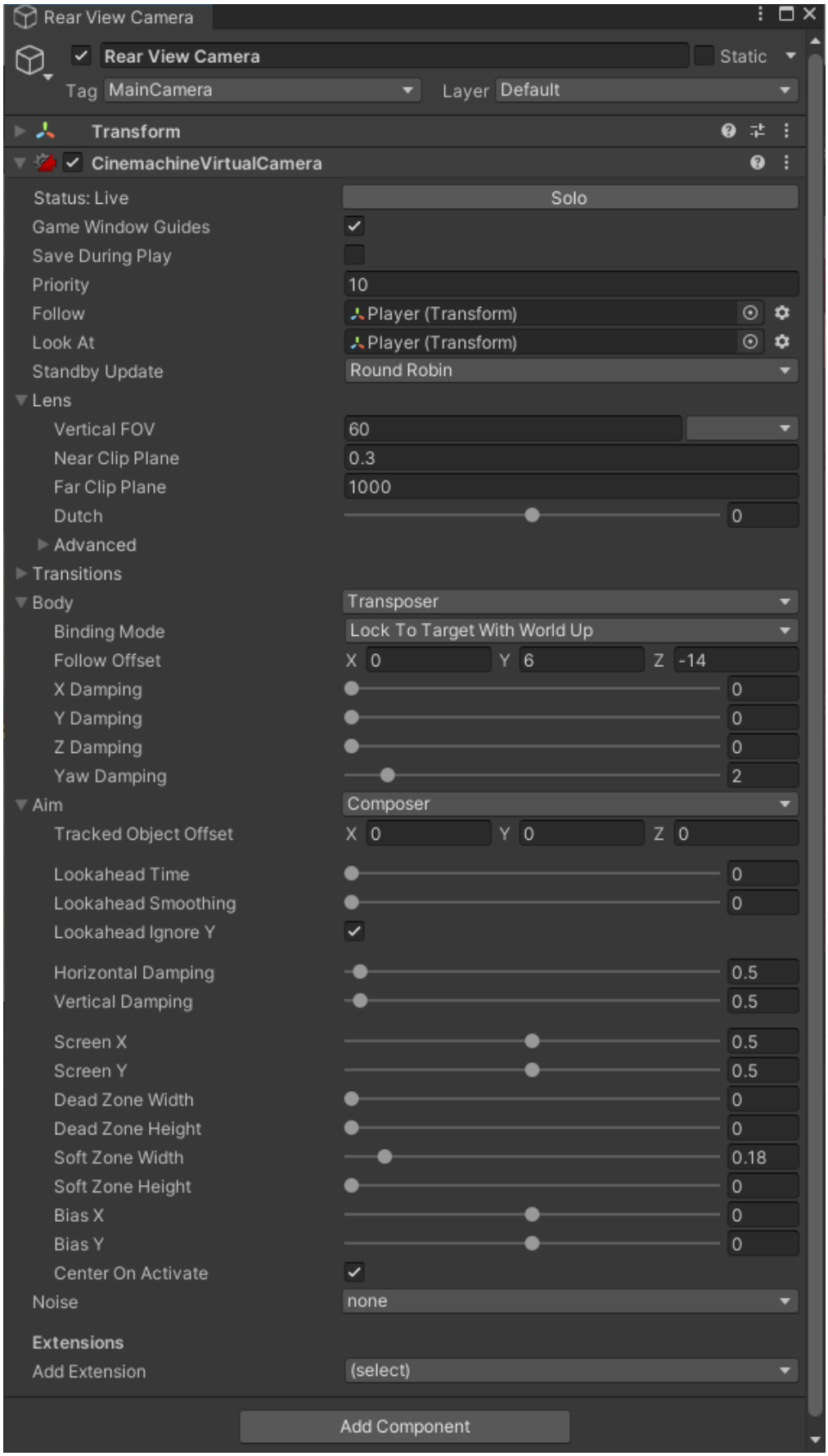
4.2. Cinemachine – paket kamera

Cinemachine je skup modula za upravljanje kamerom u Unity-u [14]. Cinemachine paket se može besplatno preuzeti u Unity Package Manager-u. Paket je temeljito osmišljen i brine o kompleksnoj logici ponašanja kamera. Obuhvaća veliki broj pozicija kamera i stilova snimanja za sve vrste žanrova igara. Paket sadrži hrpu mogućnosti, ali najvažnije je što oslobađa projektni tim od razvoja opširne logike za upravljanje kamerom i time značajno ubrzava razvoj. Brojni načini snimanja već su gotovi za korištenje, a neka specifičnija ponašanja mogu se postići podešavanjem parametara. Cinemachine kamera radi u stvarnome vremenu i ima mogućnosti dinamički se prilagođavati, pratiti objekt pri promjenama brzine, promjenama pozicija i scena.

4.2.1. Cinemachine Virtual Camera

Cinemachine je osmišljen da se ne bi stvarale nove kamere, nego da usmjerava običnu Unity kameru na više kadrova. Za postavljanje tih kadrova koriste se virtualne kamere iz Cinemachine paketa. Virtualne kamere zasebni su objekti igre i stoje odvojeno od obične Unity kamere. Svaka virtualna kamera upravlja svojim kadrom i tada ima kontrolu nad običnom Unity kamerom.

Svojstva komponente virtualne kamere pokazana su na slici 4.3. Virtualna kamera sadrži svojstva koja referenciraju Transform komponente objekata kako bi kamera mogla pratiti njihovo kretanje. Svojstvima se zadaje koji objekt kamera treba pratiti (Follow) i prema kojem objektu treba gledati (LookAt).



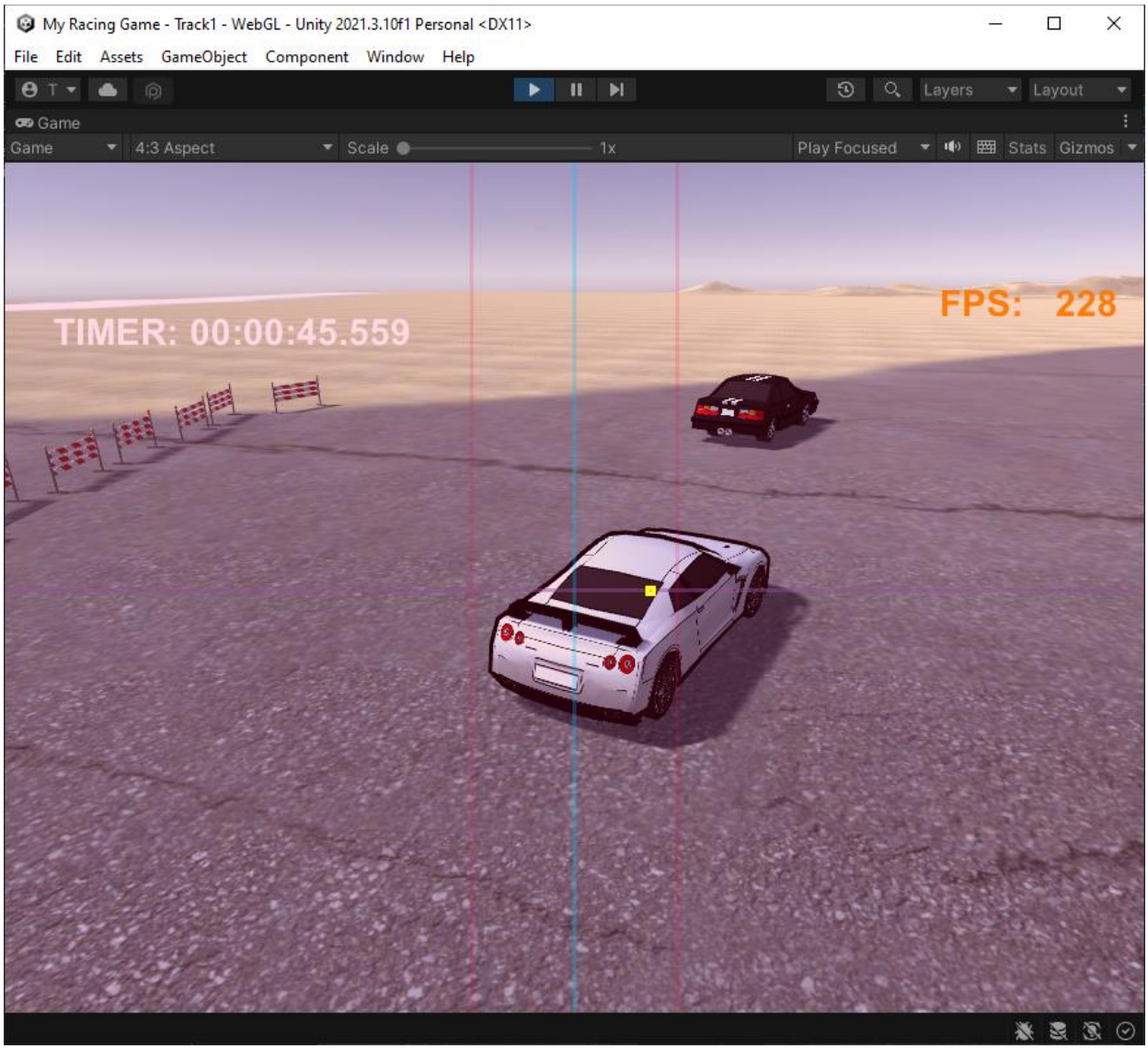
Slika 4.3. Svojstva komponente Cinemachine Virtual Camera

U projektu kamera prati vozilo sa stražnje strane i namješteno je da prilikom skretanja dopušta određeno odstupanje u poravnanju s vozilom pa tako i malo kašnjenje prije ponovnog poravnanja. Slikom 4.4. pojasnit će se svojstva kojima se postiglo željeno ponašanje kamere u ovoj igri. Svojstva koja su bitna za podesiti:

- Dead Zone – je područje u kojem se kamera ne treba vraćati u poravnanje, a kada je objekt u tom području kamera se neće vraćati. U ovoj igri kao i u većini slučajeva je potrebno da se kamera uvijek vrača u točno poravnanje s objektom, stoga je Dead Zone područje uklonjeno.

- Soft Zone – je područje u kojem je dozvoljeno odstupanje kamere od orijentacije objekta kojeg prati, pri čemu se kamera uvijek automatski vraća u poravnanje s tim objektom. To je područje između dvije vertikalne crvene linije, prikazano na slici 4.4. Ovime se postiže blagi efekt kašnjenja pri kojem kamera lagano zaostaje pri skretanju vozila prije nego se ponovno poravna sa smjerom kretanja.

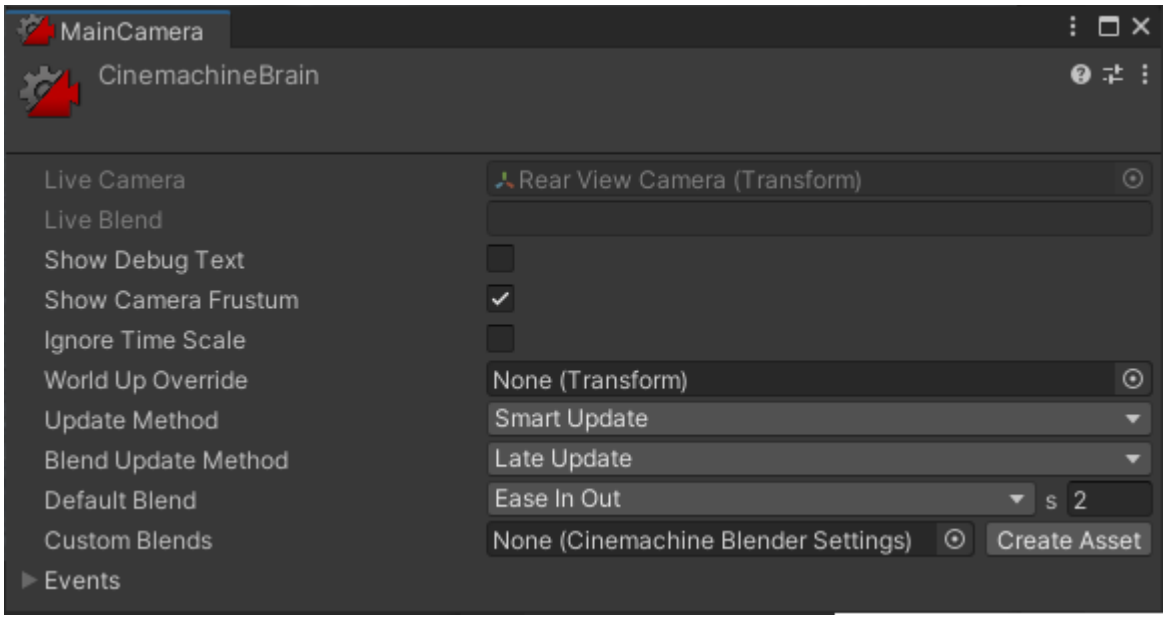
- Damping – je dopušteno odstupanje objekta iz fokusa kamere. Žuti kvadratić na slici 4.4. prikazuje koliko se objekt pomaknuo od sredine kamere.



Slika 4.4. Kalibriranje Cinemachine kamere

4.2.2. Cinemachine Brain

Cinemachine Brain je komponenta koja se dodaje na glavnu Unity kameru. Ta komponenta nadzire sve aktivne virtualne kamere u sceni. Cinemachine brain bira posljednje aktiviranu virtualnu kameru te je postavlja kao aktivnu (Live) kameru. Pri tome, odabir aktivirane kamere može ovisiti i o prioritetu. Izmjene između kamera odrađuju se na način da se aktivira ili deaktivira objekt virtualne kamere. Na slici 4.5. se vide svojstva Cinemachine Brain komponente.



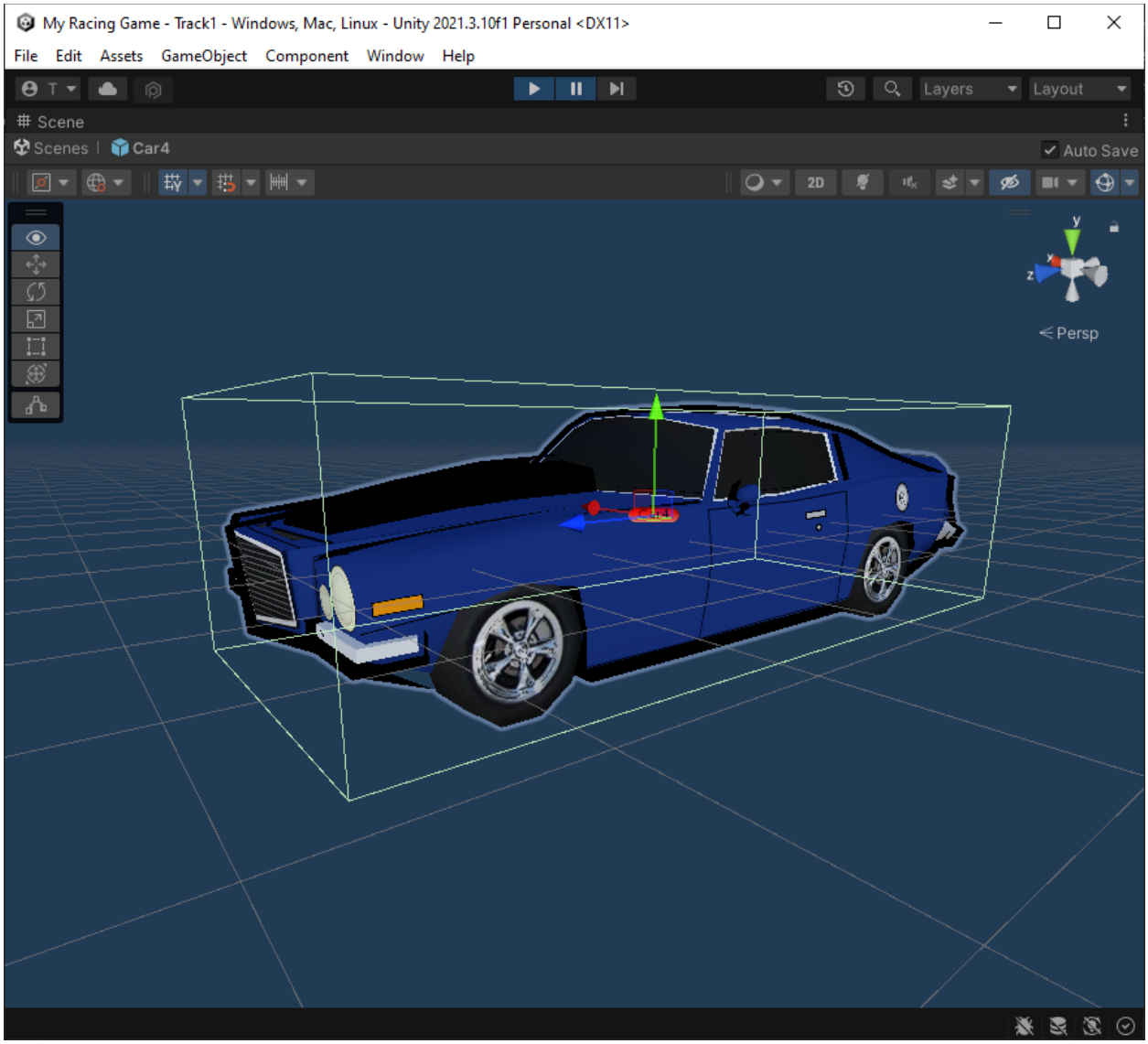
Slika 4.5. Svojstva Cinemachine Brain komponente

4.3. Model vožnje

3D model automobila samo se iscrtava da je vidljiv na ekranu, a zapravo on ne sudjeluje u interakciji s okolinom. Interakcija automobila s okolinom odvija se preko njegovog Collider-a. Collider je oblika kvadra iz BoxCollider komponente i podešen je da obuhvati cijeli 3D model kao što je prikazano na slici 4.6. Model vožnje jednostavno je izveden kao kvadar koji se kreće po podu. U ovom segmentu moguća su poboljšanja jer u ovakvoj izvedbi y-os nije potrebna za vožnju te se vožnja odvija samo po x i z-osi.

Rigidbody komponenta reprezentira fizičko tijelo, a dimenzije i oblik tog tijela definirane su BoxCollider komponentom. Rigidbody je glavna komponenta koja omogućuje fizičko ponašanje objekata na način da se simulacijom fizike upravlja pozicijom objekata. Ovom komponentom omogućeno je korištenje gravitacije i gotovih funkcija za dodavanje sila. Ugrađene funkcije Rigidbody komponente omogućile su da se objekte u igri pomiče djelovanjem sile umjesto da se ispočetka gradi cijeli sustav fizike korištenjem translacije i rotacije. Korištenjem fizike i dodavanjem sile odmah se postiže očekivano ponašanje pri pomicanju objekata. Inače, svaki objekt ima svoju Transform komponentu koja definira položaj, rotaciju i veličinu objekta u sceni. Time se objekti mogu translatirati i rotirati, ali to nije isto kao korištenje fizike. Dodavanjem sile ili zakretnog momenta sav posao rotiranja i translatiranja odrađuje se u Rigidbody komponenti. Naravno da će se pri tome mijenjati i vrijednosti u Transform komponenti, ali ne direktno već

kao posljedica djelovanja sila. Mijenjanje položaja preko Transform komponente u isto vrijeme pri korištenju fizike može uzrokovati probleme sa izračunima i kolizijama. Stoga je potrebno koristiti ili jedan ili drugi način kontrole kretanja objekata.



Slika 4.6. BoxCollider komponenta

4.3.1. Upravljanje vozilom

Za upravljanje vozilom poslužilo je korištenje ulaza preko virtualnih osi. Unity API nudi metodu koja vraća vrijednost u rasponu od -1 do 1 za pojedinu os. Svrha unosa preko virtualnih osi je što pružaju mogućnost doziranja ulaza putem tipkovnice ili češće gljivica kontrolera. Osi se identificiraju svojim imenima, pa je tako smjer kretanja po vertikalnoj ili horizontalnoj osi određen time je li povratna vrijednost ulaza za promatranu os veća ili manja od nule.

U igri je namjerno napravljeno da svaki automobil vozi na isti način kako bi utrkivanje bilo ravnopravno. Zbog toga svi automobili imaju iste specifikacije - istu maksimalnu brzinu vožnje, vrijeme ubrzanja, snagu kočenja i skretanja. Iako su specifikacije identične za sve automobile, u projektu je ostvarena struktura koja omogućuje jednostavnu dodjelu jedinstvenih specifikacija svakom automobilu. Izvedeno je na način da skripta koja upravlja vozilom koristi podatke iz Scriptable objekta koji joj je predan, to je objekt koji služi kao spremnik podataka. Definicija Scriptable objekta nalazi se na slici 4.7. Ovakvi objekti omogućuju jednostavne i brze izmjene specifikacija za individualna ponašanja automobila.

```
[CreateAssetMenu(fileName = "CarSpecs", menuName = "CarSpecs Scriptable Object")]
public class CarSpecifications : ScriptableObject
{
    public float maxSpeed = 700f;
    public float accelerationTime = 5f;
    public float breakingStrength = 2.5f;
    public float turnStrength = 2f;
    public float maxWheelTurn_deg = 50f;
}
```

Slika 4.7. Scriptable objekt za specifikacije automobila

Idući dio skripte (slika 4.8) pokazuje varijable koje se koriste, pokazuje dohvaćanje referenci za Rigidbody komponentu vozila i dohvaćanje Transform komponenata prednjih kotača.

```
private Rigidbody carRigidBody;
public CarSpecifications specs;
private float maxSpeed;
private float speed = 0;
private float accelerationRate;

[SerializeField] Transform leftFrontWheel, rightFrontWheel;

void Awake()
{
    carRigidBody = this.GetComponent<Rigidbody>();

    leftFrontWheel = this.transform.Find("Front_Left_Wheel");
    rightFrontWheel = this.transform.Find("Front_Right_Wheel");
}

void Start()
{
    maxSpeed = specs.maxSpeed;
    accelerationRate = maxSpeed / specs.accelerationTime;
}
```

Slika 4.8. Inicijalizacijski dio skripte za upravljanje vozilom

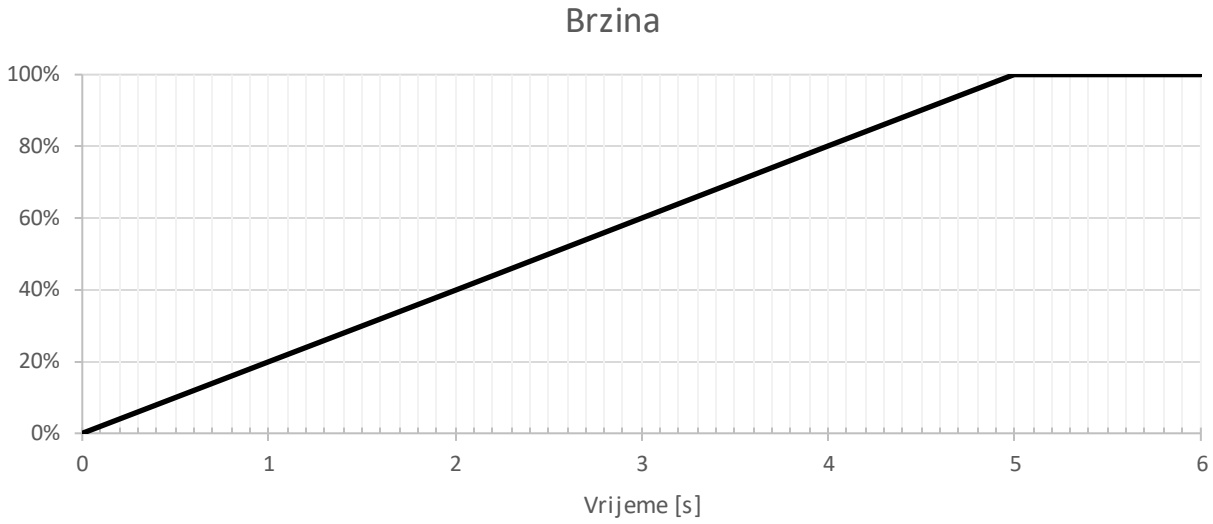
Update metoda se poziva svakim iscrtavanjem slike na ekran. Za ovu metodu nema garancije da će držati konstantni razmak između poziva, ali pošto se poziva svakim iscrtavanjem slike u njoj se provjeravaju ulazi kako bi se postigao najbolji odziv naredbi. Za razliku od Update, metoda FixedUpdate održava konstanti vremenski razmak između svakog poziva, stoga se u nju obično smještaju izračuni fizike. Prema tome, djelovanje sile za kretanje automobila je smješteno u FixedUpdate. Automobil vozi na način da se dodaje sila na Rigidbody, a iznos i smjer djelovanja sile računaju se na način da se vektor smjera objekta pomnoži sa vrijednosti varijable brzine. Uz to je potrebno postaviti ForceMode parametar tako da ta sila dodaje konstanto ubrzanje na Rigidbody i zanemaruje masu. U FixedUpdate se još nalaze i pozivi metoda koje brinu o skretanju kotača i prilagodbu brzine prema terenu po kojem se vozi. Slijedi programski kod koji služi za upravljanje vozilom.

```
void Update()
{
    OtherControls();
    DrivingControls();
}

void FixedUpdate()
{
    carRigidBody.AddForce(
        carRigidBody.transform.forward * speed,
        ForceMode.Acceleration
    );

    AdjustSpeedForTerrain();
    TurningWheels();
}
```

Slika 4.9. Metode Update i FixedUpdate u PlayerController skripti

Na slici 4.9. može se vidjeti kako su naredbe i izračun fizike odvojeni u pripadajuće metode po učestalosti pozivanja. Kontrole se smještaju u Update metodu kako bi se unos testirao svakim iscrtavanjem slike, a unutar FixedUpdate metoda odrađuje dodavanje sile.

Brzina vozila je ograničena na neki maksimum i postiže se konstantnim ubrzanjem. Brzina se povećava pri svakom pozivu Update metode, a iznos za koji se brzina poveća pri svakom pozivu je mali isječak ubrzanja. Brzina koja se postigne u tom kratkom vremenu je dio ubrzanja za to vrijeme između dva poziva Update metode, a to vrijeme se dohvaća kao Time.deltaTime. To je isto što i vrijeme između iscrtavanja pojedine slike ili period promjene slike. Ubrzanje do maksimalne brzine postavljeno je da traje 5 sekundi (graf na slici 4.10.). Slika ilustrira promjenu brzine prema periodu promjene slike, taj period je na grafu prikazan vertikalnim linijama. Kako bi graf bio pregledniji ovdje je za primjer frekvencija promjena slika (Frames Per Second) samo 10 Hz, od uobičajenih 30 Hz na više. U stvarnoj primjeni pozivi iscrtavanja slika bi bili još češći i ne bi se pojavljivali u tako pravilnim razmacima.



Slika 4.10. v – t graf ubrzanja

Za vožnju naprijed – nazad provjeravamo ulaz sa vertikalne osi. Prema tome se provjeravaju tri slučaja ulaza: naprijed, nazad ili nema ulaza. Zatim se prema tim slučajevima prilagođava brzina.

U izvedbi ovog projekta skretanje vozilom i skretanje kotača je odvojeno. Kotači skreću na način da se rotiraju oko y-osi za najviše 50°, dok vozilo skreće tako da se rotira objekt automobila. Skretanje vozila mora biti omogućeno samo kada je vozilo u pokretu. Kada taj uvjet bude zadovoljen poziva se metoda koja dalje obavlja posao skretanja vozila. Kako bi se utvrdilo da li je vozilo u pokretu računa se skalarni umnožak.

```
private void DrivingControls()
{
    accelerationRate = maxSpeed / specs.accelerationTime;
    float dotProduct
        = Vector3.Dot(carRigidBody.velocity, carRigidBody.transform.forward);

    if (Input.GetAxis("Vertical") > 0)
    {
        HandlePositiveVerticalInput();
    }
    else if (Input.GetAxis("Vertical") < 0)
    {
        HandleNegativeVerticalInput();
    }
    else
    {
        HandleNoVerticalInput(dotProduct);
    }
    speed = Mathf.Clamp(speed, -maxSpeed, maxSpeed);

    if (IsMovingForwards(dotProduct) || IsMovingBackwards(dotProduct))
    {
        HandleSteering();
    }
}
```

Slika 4.11. Upravljanje vozilom

Metoda na slici 4.11. rješava kontrole vožnje. Za ulaze na vertikalnoj osi pozitivna vrijednost ulaza znači da je pritisnuta naredba naprijed, a za negativan ulaz naredba nazad. Nakon toga potrebno je odraditi odgovarajuću promjenu brzine prema unesenoj naredbi i prema onome kako se vozilo već kreće. Smanjivanje i povećavanje brzine odnosi se na promjenu varijable brzine prema apsolutnoj vrijednosti, pa su pozitivan ili negativan iznos samo smjerovi kretanja. Cijela mehanika vožnje obavlja se ulazom po osima. Varijabla brzine je množitelj vektora smjera kod dodavanja sile na objekt u igri (dodavanje sile se nalazi u FixedUpdate metodi). Negativan iznos brzine znači da se vozi unazad, odnosno da se dodaje sila u suprotnom smjeru orijentacije objekta. Za kočenje automobila je potrebno samo pritisnuti ulaz suprotnog smjera od trenutnog smjera kretanja vozila. Brzina će se tada smanjivati do zaustavljanja, a zatim će nastaviti rasti u suprotnom smjeru i vozilo će se nastaviti kretati tim smjerom. Ovako se brzina ponaša po apsolutnoj vrijednosti. Vozilo se zapravo kreće brzinom između $[-v_{MAX}, v_{MAX}]$ gdje se predznak iskoristio za smjer kretanja.

Potrebno je bilo pokriti slijedeće slučajeve prema ulazima koji se mogu pojaviti (slika 4.12). Prvi slučaj je povećavati brzinu u smjeru prema pritisnutoj naredbi. Drugi slučaj je kada se vozilo već kreće nekom brzinom, ali suprotnog smjera od smjera naredbe koja se pritišče, to znači da je potrebno kočiti, tj. mijenjati brzinu u suprotnom smjeru. Treći slučaj je kada nema unosa naredbe, a vozilo je još uvijek u pokretu. Tada je potrebno smanjivati brzinu i zaustaviti vozilo.

```
private void HandlePositiveVerticalInput()
{
    if (speed < 0)
    {
        speed += accelerationRate * Time.deltaTime * specs.breakingStrength;
    }
    else
    {
        speed += accelerationRate * Time.deltaTime;
    }
}

private void HandleNegativeVerticalInput()
{
    if (speed > 0)
    {
        speed -= accelerationRate * Time.deltaTime * specs.breakingStrength;
    }
    else
    {
        speed -= accelerationRate * Time.deltaTime;
    }
}

private void HandleNoVerticalInput(float dotProduct)
{
    if(IsMovingForwards(dotProduct))
    {
        speed -= accelerationRate * Time.deltaTime;
    }
    else if (IsMovingBackwards(dotProduct))
    {
        speed += accelerationRate * Time.deltaTime;
    }
    else
    {
        speed = 0;
    }
}
```

Slika 4.12. Metode za upravljanje ubrzavanjem i usporavanjem

Sljedeća metoda obavlja zadaću skretanja vozilom (slika 4.13.). Prvo je potrebno vektor smjera automobila pretvoriti u vektor smjera skretanje. Vektor smjera skretanja dobije se na način da se obični vektor smjera rotira po y-osi za kut skretanja kotača. Za taj novi vektor smjera potrebno je pronaći njegovu rotaciju, njegov Quaternion (Unity rotacije sprema kao

kvaternione). Zatim se postepeno obavlja skretanje do tog novog vektora smjera. To se obavlja korištenjem metode za sferičnu interpolaciju čime je postignuto da se vozilo postepeno i glatko rotira svakom iteracijom metode. Iznos rotacije je definiran u specifikacijama vozila kao snaga skretanja.

```
private void HandleSteering()
{
    Vector3 originalForward = transform.forward;
    Vector3 directionSteer =
        Quaternion.AngleAxis(
            specs.maxWheelTurn_deg * Input.GetAxis("Horizontal"),
            Vector3.up
        ) * originalForward;
    Quaternion rotationSteer = Quaternion.LookRotation(directionSteer);
    transform.rotation = Quaternion.Slerp(
        transform.rotation,
        rotationSteer,
        specs.turnStrength * Time.deltaTime
    );
}
```

Slika 4.13. Metoda za upravljanje skretanjem vozila

4.3.2. Utvrđivanje orijentacije skalarnim umnoškom

Za utvrđivanje orijentacije smjera kretanja objekta računa se skalarni umnožak kako bi se utvrdilo kreće li se objekt naprijed ili nazad. Skalarni umnožak: vektora brzine (velocity) i vektora smjera promatranog objekta. Vektori za objekt vozila, nalaze se u komponentama [11]:

- Rigidbody.velocity – reprezentira stopu promjene položaja Rigidbody komponente.
- Transform.forward – vraća normalizirani vektor z-osi, uzima u obzir orijentaciju objekta.

Skalarni umnožak vektora je realan broj $\vec{a} \cdot \vec{b}$ gdje je $\emptyset$ kut između $\vec{a}$ i $\vec{b}$ definiran kao:

$$\vec{a} \cdot \vec{b} = |\vec{a}| \cdot |\vec{b}| \cdot \cos \emptyset$$

Skalarni umnožak u Kartezijevom koordinatnom sustavu:

$$\vec{a} \cdot \vec{b} = \overrightarrow{a_x} \cdot \overrightarrow{b_x} + \overrightarrow{a_y} \cdot \overrightarrow{b_y} + \overrightarrow{a_z} \cdot \overrightarrow{b_z}$$

Skalarni umnožak dva vektora mjeri sličnost njihovih smjerova. Ako su dva vektora paralelna skalarni umnožak će im biti pozitivan, a ako su anti-paralelni (suprotnih orijentacija), skalarni umnožak će biti negativan. Ako su vektori paralelni skalarni umnožak im je nula. Po ovom principu uspoređivanjem predznaka skalarnog umnoška može se utvrditi da li se objekt kreće: naprijed, nazad ili je nepomičan. Pri tome da kod mirovanja objekta skalarni umnožak neće ispasti točno nula nego će se nalaziti unutar graničnih vrijednosti.

Za uspoređivanje skalarnog umnoška potrebno je bilo zadati neku graničnu vrijednost (granična vrijednost je konstanta dotProductTreshold). Granična vrijednost postoji zbog greške pri računanju s realnim brojevima, tj. načina na koji su realni brojevi prikazani na računalu. Radi ovog ograničenja događa se da rezultat skalarnog umnoška ne ispadne točno nula čak i kad nema kretanja. Iako bi u slučaju kada objekt miruje vektor brzine trebao biti nula. U takvom slučaju kada se objekt ne kreće, a u izračunu se pojavljuje neka vrijednost skalarnog umnoška različita od nule ona bude zanemarivo mala, ali nije nula. Iz tog razloga bilo je potrebno dodati graničnu vrijednost kao toleranciju na odstupanje u računanju s realnim brojevima.

Slika 4.14. prikazuje metode za provjeru nalazi li se vozilo u pokretu.

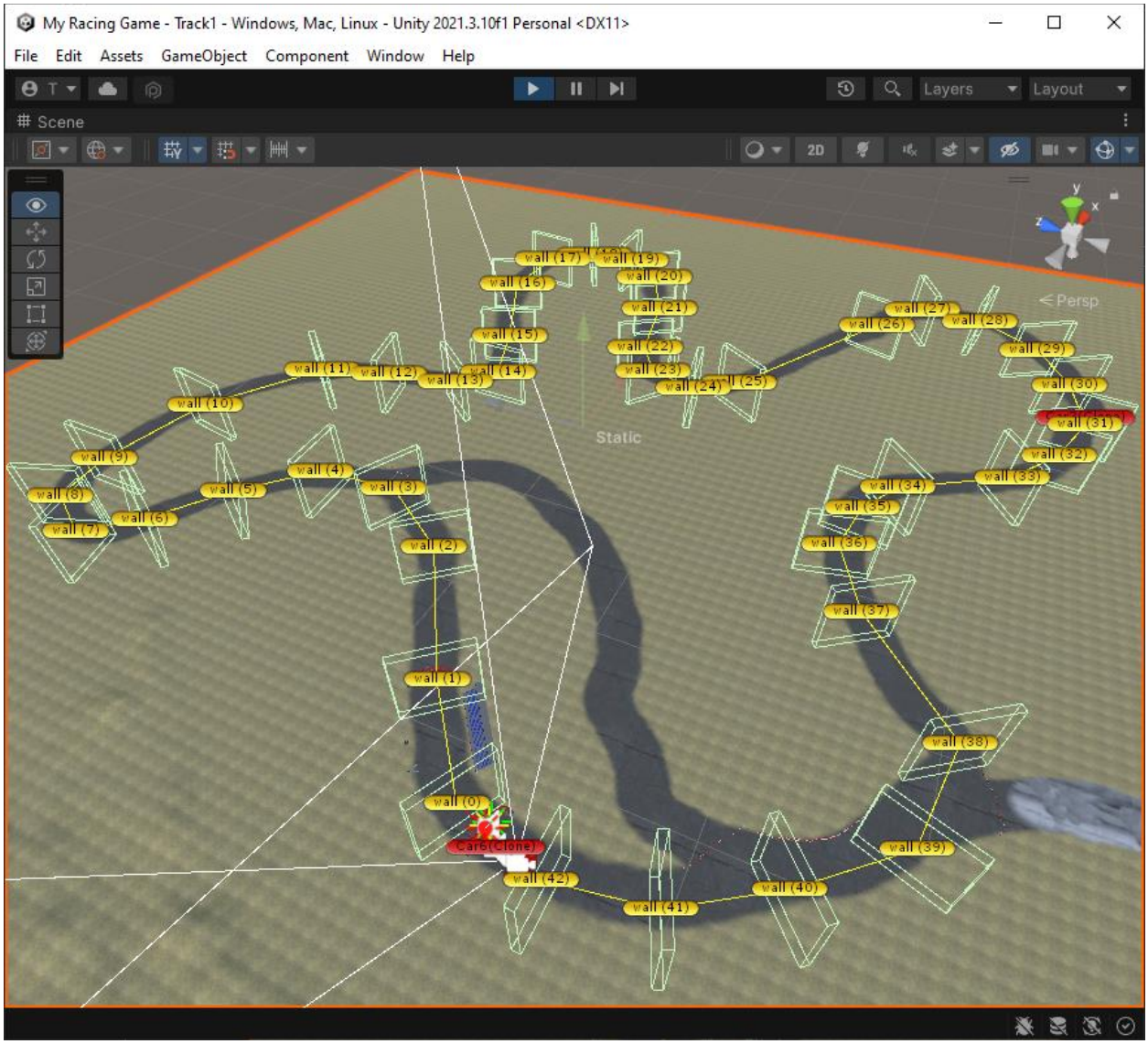
```
private const float dotProductTreshold = 0.6f;
private bool IsMovingForwards(float dotProduct)
{
    return dotProduct > dotProductTreshold;
}
private bool IsMovingBackwards(float dotProduct)
{
    return dotProduct < -dotProductTreshold;
}
```

Slika 4.14. Utvrđivanje smjera kretanja skalarnim umnoškom

4.4. Kretanje protivnika

Navigacija protivnika po stazi odvija se putem kontrolnih točaka koje su postavljene duž cijele staze. Cijela staza i sve kontrolne točke mogu se vidjeti na slici 4.15. Kontrolne točke koriste se kako bi računalni protivnik mogao prilagođavati svoju putanju. Kretanje protivnika izvedeno je na način da protivnik osvaja jednu po jednu kontrolnu točku i uvijek ima samo jednu kontrolnu točku za postavljeni cilj kretanja. Tek nakon prolaska kroz ciljanu kontrolnu točku

postavlja se smjer prema idućoj u nizu. Zatim će se putanja prilagođavati prema toj novo zadanoj kontrolnoj točki. Na ovaj način računalni protivnik ima sposobnost voziti po stazi i prilagoditi putanju nazad na stazu u slučaju skretanja sa staze.

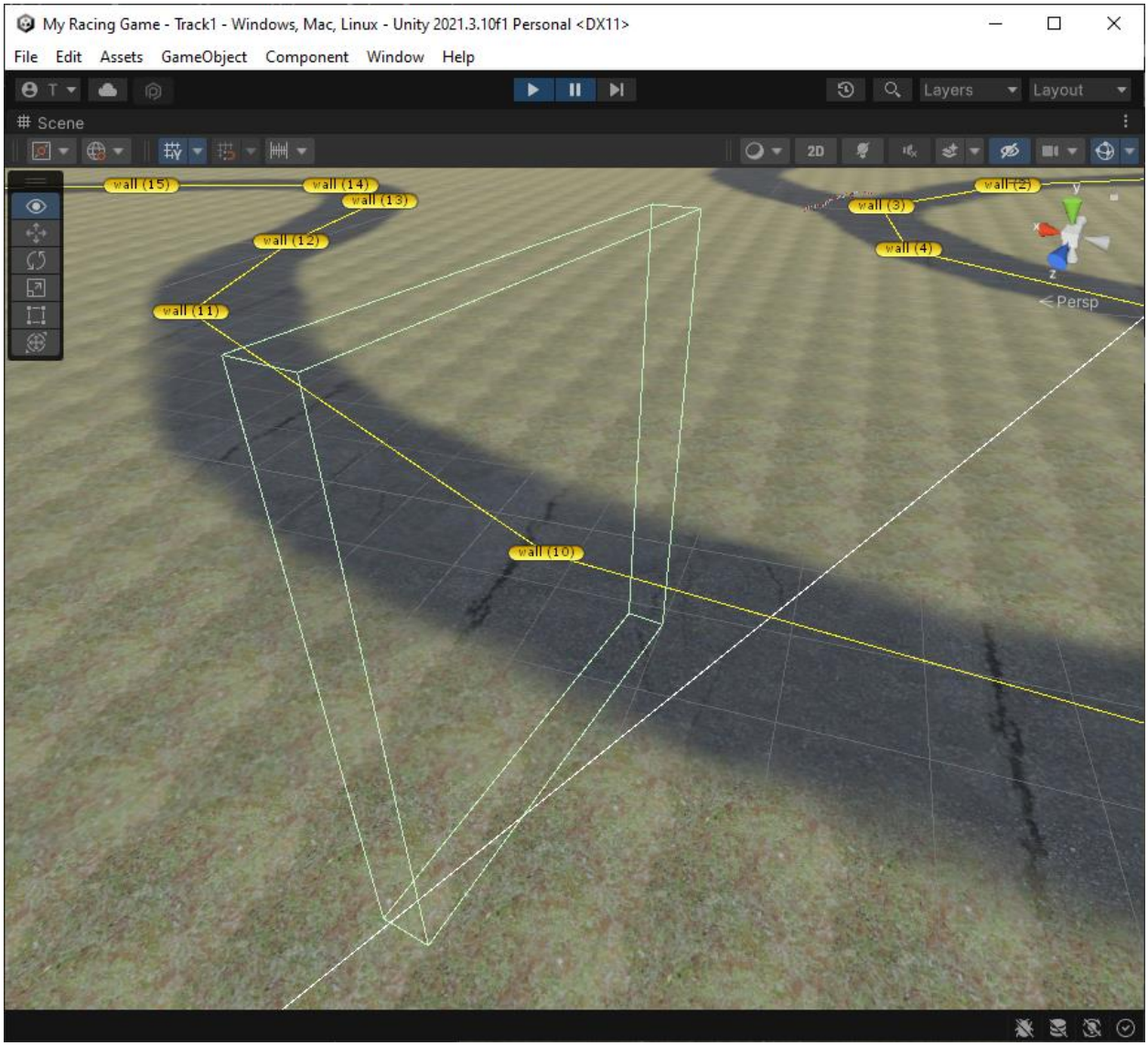


Slika 4.15. Cijela staza i zidovi kontrolnih točaka

Kontrolna točka predstavlja zid koji mora zauzeti širinu ceste da prepozna prolazak vozila (slika 4.16.). Nije potrebno da zid u igri bude vidljiv pa se za taj objekt može isključiti njegova komponenta za iscrtavanje.

Svi objekti za koje se želi da budu interaktivni u igri moraju imati Collider. To je komponenta kojom se definira oblik tijela koji će biti interaktivan u igri. Collider komponentu kontrolne točke potrebno je bilo konfigurirati da bude okidač (engl. *Trigger*), tj. u komponenti

uključiti *isTrigger* svojstvo. To znači da taj Collider neće fizički djelovati s drugim objektima, nego će okidati događaje dok drugi objekti uđu u njegov prostor. Kada drugi objekt (Collider) uđe u prostor okidača tada okidač poziva *OnTriggerEnter* metodu koju je potrebno implementirati u skripti tog nadolazećeg objekta. Metoda kao argument prima referencu na Collider komponentu objekta iz kojeg je okidač pozvao taj događaj. Zatim se u toj metodi postavlja idući cilj, tj. iduća kontrolna točka prema kojoj će se prilagođavati putanja vozila.



Slika 4.16. Zid kontrolne točke

4.4.1. Implementacija protivnika

Cilj je postići da vozilo ne prati fiksnu liniju vožnje, već da može prilagođavati svoju putanju [9]. Putanja se prilagođava prema kontrolnim točkama, no one su fiksno postavljene, stoga se na

prvi pogled može činiti da protivnik vozi po fiksnoj putanji. Kada se igrač sudari sa suparničkim vozilom i pomakne ga u njegovoj putanji protivnik će prilagoditi svoju putanju kako bi i dalje prošao kroz ciljanu kontrolnu točku. Ovo je vrlo jednostavna izvedba kretanja protivnika koja rješava zadaću vožnje po stazi i prilagođava putanju. Međutim ova implementacija ne rješava sve nepredvidive situacije, prilagodbu brzine i sudare. Stoga bi za poboljšanja bilo potrebno koristiti i još neke druge tehnike umjetne inteligencije.

Kontrolne točke se nalaze u cikličkom nizu, pristupa im se kao djeci jednog roditeljskog objekta. Za izvedbu računalnog protivnika definirana je posebna struktura podataka (slika 4.17). U toj strukturi nalazi se: referenca na taj roditeljski objekt unutar kojeg se nalaze sve kontrolne točke, indeks kontrolne točke prema kojoj se usmjerava kretanje, vektor koji usmjerava kretanje i kvaternion za upravljanje rotacijom.

```
private struct structAI
{
    public Transform checkpoints;
    public int checkpointWallIndex;
    public Vector3 directionSteer;
    public Quaternion rotationSteer;
}
```

Slika 4.17. Struktura podataka za ostvarenje kretanja protivnika

Ispod se nalazi programski kod skripte za upravljanje računalnim protivnikom (slika 4.18.). Sve varijable koje se tiču same vožnje vozila u ovoj skripti iste su kao i kod skripte za ljudskog igrača. Ova skripta dodatno sadrži referencu na BoxCollider za detekciju prolaska kontrolnim točkama i strukturu podataka koja služi da održava putanju.

```
private Rigidbody carRigidBody = null;
private BoxCollider boxCollider;
private structAI ai;
public CarSpecifications specs;

private float maxSpeed = 700;
private float speed = 0;
private float accelerationRate;

void Awake()
{
    carRigidBody = this.GetComponent<Rigidbody>();
    boxCollider = this.GetComponent<BoxCollider>();
    ai.checkpoints = GameObject.FindWithTag("Checkpoints").transform;
    ai.checkpointWallIndex = 0;
}

void Start()
{
    maxSpeed = specs.maxSpeed;
    accelerationRate = maxSpeed / specs.accelerationTime;
}
```

Slika 4.18. Inicijalizacijski dio skripte za upravljanje protivnikom

Programski kod koji služi za vožnju protivnikovog vozila pokazan je na slici 4.19. Putanja kretanja se prilagođava pri svakom pozivu Update metode. Od trenutne pozicije postavlja se smjer prema zadanoj kontrolnoj točki. Zatim se postepeno obavlja rotacija kako bi se vozilo usmjerilo na putanju koja prolazi kontrolnom točkom. Kako bi rotacija vozila prema tom novom smjeru bila glatka koristi se sferična interpolacija, a iznos rotacije je zadan kao snaga skretanja u specifikacijama vozila. Unutar FixedUpdate metode obavlja se dodavanje sile ubrzanja i prilagodba brzine za teren po kojem se vozi.

```
void Update()
{
    speed += accelerationRate * Time.deltaTime;
    speed = Mathf.Clamp(speed, 0, maxSpeed);
    HandleSteering();
}

private void HandleSteering()
{
    ai.directionSteer =
        ai.checkpoints.GetChild(ai.checkpointWallIndex).position -
        this.transform.position;

    ai.rotationSteer = Quaternion.LookRotation(ai.directionSteer);

    this.transform.rotation = Quaternion.Slerp(
        this.transform.rotation,
        ai.rotationSteer,
        specs.turnStrength * Time.deltaTime
    );
}

private void FixedUpdate()
{
    carRigidBody.AddForce(
        carRigidBody.transform.forward * speed,
        ForceMode.Acceleration
    );
    AdjustSpeedForTerrain();
}
```

Slika 4.19. Metode za upravljanje vozilom protivnika

Programski kod na slici 4.20. služi za okidanje kolizije i postavljanje iduće kontrolne točke. Na događaj kolizije, ako se radi o kontrolnoj točki (objektu koji nosi oznaku Wall), tada je potrebno postaviti novi cilj - iduću kontrolnu točku. Kontrolne točke se nalaze u cikličkom nizu.

```
private int CalcNextCheckpoint()
{
    int current = ai.checkpointWallIndex;
    int next = ai.checkpointWallIndex + 1;
    if (next > ai.checkpoints.childCount - 1)
        next = 0;

    return next;
}

private void OnTriggerEnter(Collider collider)
{
    if (collider.CompareTag("Wall") == true)
    {
        ai.checkpointWallIndex = CalcNextCheckpoint();
    }
}
```

Slika 4.20. Okidanje kolizije i postavljanje iduće kontrolne točke

4.5. Prilagodba brzine prema teksturi terena

Prilagodba brzine prema teksturi terena koristi se kao mehanika igre. Ova mehanika je namijenjena kako bi se kaznilo igrača za vožnju izvan ceste. Prema tome je postavljeno ograničenje da se brzina vožnje izvan ceste može razviti najviše do 40 % maksimalne brzine.

Kod slika na računalu uz uobičajene crvene (red), plave (blue) i zelene (green) kanale, koji definiraju boje obično postoje još i alpha kanali. Alpha kanali su dodatni kanali koji sadrže informacije o transparentnosti pojedinog piksela [11, 12]. Kod video igara Alphamap izraz predstavlja preklapanje ili snagu pojavljivanja različitih tekstura na terenu. U igrama se može pojaviti da na terenu postoji više tekstura. Svaki piksel odgovara jednoj lokaciji na terenu, a svaki kanal u alphamapu reprezentira snagu određene teksture na toj lokaciji.

Cilj je utvrditi po kojoj teksturi se vozilo kreće. Za prepoznavanje teksture potrebno je imati referencu na objekt terena po kojem se vozi. Zatim se predaje vektor položaja igrača te se za tu lokaciju na terenu traži tekstura. Radi se provjera da li se predani položaja nalazi unutar terena te je taj vektor potrebno normalizirati unutar položaja terena. Zatim se dohvaćaju alphamap podaci pozivanjem metode GetAlphamaps. Metoda izvlači alphamap podatke za igračevu lokaciju na terenu. Metoda za parametre prima x i z koordinate te širinu i visinu objekta za kojeg se traži alphamap na njegovoj lokaciji. Kao povrat metoda vraća 3-dimenzionalno polje o raspodjeli

tekstura po terenu – prve dvije dimenzije su x i y koordinate na mapi, a treća dimenzija označava teksturu na koju se alphamap primjenjuje [11].

Parametri za širinu i visinu se odnose na područje u pikselima koje se promatra na zatraženoj lokaciji. U ovom slučaju dovoljno je za širinu i visinu predati broj 1 jer ovi parametri definiraju veličine prvih dviju dimenzija alphamap polja. Postavljanjem područja na dimenzije 1x1 promatra se samo jedan piksel. Treća dimenzija alphamap polja podataka bit će velika isto koliko teren ima tekstura. Prema tome se najdominantnija tekstura za predani položaj može saznati kao najveća vrijednost alphamap polja po trećoj dimenziji.

Primjer za dvije teksture i područje 2 x 2 piksela:

$$\left[\begin{bmatrix}0 & 0\\0 & 0\end{bmatrix},\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}\right]$$

$x = 0\,, y = 0 \qquad \text{snaga teksture} = 1$

$x = 0\,, y = 1 \qquad \text{snaga teksture} = 1$

$x = 1\,, y = 0 \qquad \text{snaga teksture} = 1$

$x = 1\,, y = 1 \qquad \text{snaga teksture} = 1$

Može se zaključiti da dominira druga tekstura. Treća dimenzija za drugu po redu teksturu ima indeks 1 i dohvaća se kao alphamapData[i,j,1]. Vrijednosti prve dvije dimenzije polja govore kolika je snaga teksture u postotcima za svaki pojedini piksel tog područja koje se promatra. Indeksi ovog 2-dimenzionalnog polja koriste se kao koordinate piksela. Indeksi se koriste kao koordinate položaja u odnosu na x i z koordinate koje su bile predane metodi kako bi se dohvatili alphamap podaci.

Na slikama 4.21. i 4.22. nalazi se programski kod skripte koja se koristi kako bi se odredilo po kojoj vrsti terena vozilo vozi, tj. na kojoj teksturi se nalazi objekt vozila. Prema tome se postavlja maksimalna brzina vožnje za odgovarajući teren.

```
[SerializeField] private LayerMask _tag;
[SerializeField] private Terrain _terrain;

public float GetPercentageOfMaxSpeedAllowedAt(Vector3 playerPosition)
{
    int textureIndex = GetDominantTextureIndex(playerPosition);

    if (textureIndex == 0) // grass
    {
        return 0.4f;
    }
    else
    {
        return 1f; // road
    }
}

private Vector3 GetNormalizedTerrainPosition(Vector3 position)
{
    // Convert world position to normalized position within the terrain.
    Vector3 normalizedPosition = new Vector3(
        (position.x - _terrain.terrainData.bounds.min.x) /
        _terrain.terrainData.size.x,
        0,
        (position.z - _terrain.terrainData.bounds.min.z) /
        _terrain.terrainData.size.z
    );

    return normalizedPosition;
}
```

Slika 4.21. Skripta za prilagodbu brzine prema vrsti terena

Za pronalazak teksture potrebno je referencirati Terrain komponentu. Metoda *GetPercentageOfMaxSpeedAllowedAt* poziva se iz skripte vozila i ona vraća postotak na koji se postavlja maksimalna brzina vožnje za vrstu terena po kojem se kreće. Metoda *GetNormalizedTerrainPosition* koristi se kako bi se pretvorilo poziciju igrača (poziciju u sceni) u normaliziranu poziciju unutar terena.

```
private int GetDominantTextureIndex(Vector3 playerPosition)
{
    // Get the player's position in world space.
    // Ensure the player's position is within the bounds of the terrain.
    if (_terrain.terrainData.bounds.Contains(playerPosition))
    {
        // Get the normalized position of the player within the terrain.
       Vector3 normalizedPosition = GetNormalizedTerrainPosition(playerPosition);

        // Get the alphamap data at the player's position.
        float[,,] alphamapData = _terrain.terrainData.GetAlphamaps(
            (int)(normalizedPosition.x * _terrain.terrainData.alphamapWidth),
            (int)(normalizedPosition.z * _terrain.terrainData.alphamapHeight),
            1, 1
        );

        // Find the index of the highest value in the alphamap data.
        int dominantIndex = 0;
        float maxAlpha = 0f;

        for (int i = 0; i < alphamapData.GetLength(2); i++)
        {
            if (alphamapData[0, 0, i] > maxAlpha)
            {
                maxAlpha = alphamapData[0, 0, i];
                dominantIndex = i;
            }
        }

        return dominantIndex;
    }
    return -1;
}
```

Slika 4.22. Metoda za traženje dominantne teksture

Metoda na slici 4.22. pronalazi indeks teksture na kojoj se igrač trenutno nalazi. Prvo provjerava nalazi li se igrač unutar terena. Zatim pronalazi igračevu normaliziranu poziciju unutar terena. Normalizirana pozicija se zatim koristi za dohvaćanje alphamap podataka terena iz kojih se može zaključiti o kojoj se teksturi radi na toj lokaciji.

5. ZAKLJUČAK

U ovom projektu model vožnje automobila zapravo je samo kvadar koji se vuče po zemlji, a skripta brine o okretanju kotača i da se brzina vrtnje poklapa s brzinom vozila. Vozilo se kreće na način da se Rigidbody komponentu rotira i na nju dodaje sila ubrzanja. Specifičan osjećaj vožnje postigao se podešavanjem vrijednosti svojstava za otpor zraka (drag i angularDrag) koja se nalaze u Rigidbody komponenti. Za napredniju i složeniju izvedbu modela vožnje u može se koristiti WheelCollider komponenta koja specifično simulira model kotača. Ova komponenta simulira ovjes, amortizer, postavlja masu kotača i gotov modela trenja gume. Komponenta posjeduje i svojstva kao što su: provjera nalazi li se kotač na zemlji, brzina rotacije, kut skretanja, itd. Može se zaključiti da je prvi način postizanja boljeg i realističnijeg modela vožnje upravo ugradnja WheelCollider komponente u svaki kotač te podešavanje brojnih parametara ponašanja takvog kotača. Zatim, iduća nadogradnja mogla bi biti promjena oblika Collidera da oblik kolizije bude oblika modela automobila umjesto naprosto kvadra, tj. korištenje MeshCollider komponente umjesto BoxCollider komponente. U slučajevima kada oblik Collider-a objekta ne odgovara obliku modela koji je iscrtan na ekranu mogu se dogoditi zapinjanja objekta za nevidljive prepreke.

Navigacija protivnika pomoću kontrolnih točki može biti prikladna u određenim situacijama, ali ima svojih ograničenja. Takva realizacija inteligencije protivnika može imati poteškoća pri nekim dinamičnim promjenama, pojavom prepreka ili kompleksnijim stazama. Za bolju umjetnu inteligenciju protivnika, tj. kao poboljšanje ili možda kompletna zamjena navigacije kontrolnim točkama mogu se koristiti tehnike strojnog i pojačanog učenja.

LITERATURA

[1] B. Nicoll i B. Keogh, The Unity Game Engine and the Circuits of Cultural Software, Palgrave Pivot Cham, Brisbane, 2019.

[2] B. Clark, Game Engine Architecture: Bill Clark (CodeLabs Tech Talk 2020), CodeDay, 2020., https://www.youtube.com/watch?v=mUeNqLcx4eI&ab_channel=CodeDay, 17.7.2023.

[3] J. Gregory, Game Engine Architecture Second Edition, Taylor & Francis Group, 2015.

[4] L. Manovich, Software Takes Command, Bloomsburry Academic, New York, 2013.

[5] MULTIPLATFORM SUPPORT CREATE ONCE, REACH BILLIONS: Unparalleled platform support, Unity Technologies, 2023., https://unity.com/solutions/multiplatform, 2.5.2023.

[6] Eyes on the Solar System, Laboratory, Visualization Technology Applications and Development Team at NASA's Jet Propulsion, NASA, 2023., https://eyes.nasa.gov/apps/solar-system/#/home, 2.5.2023.

[7] EFFICIENT DEVELOPMENT OF SIMULATED ENVIRONMENTS FOR AUTONOMOUS VEHICLE TRAINING: BMW's autonomous driving journey, Unity Technologies, 2023., https://unity.com/how-to/simulated-environments-for-autonomous-vehicle-training#bmws-autonomous-driving-journey, 2.5.2023.

[8] J. Terra, What is Technology Literacy?, Simplilearn Solutions, Nashua, New Hampshire, 2023., https://www.simplilearn.com/what-is-technology-literacy-article, 16.7.2023.

[9] B. Schwab, AI Game Engine Programming, Course Technology, Boston, 2009.

[10] A. Galvan, Game Engine Architecture, 2018., https://alain.xyz/blog/game-engine-architecture, 5.10.2023.

[11] Unity Manual, Unity Technologies, 2023., https://docs.unity3d.com/Manual/UnityManual.html, 29.7.2023.

[12] W. Goldstone, Unity Game Development Essentials, Packt Publishing, Birmingham, 2009.

[13] D. Baron, Hands-On Game Development Patterns With Unity 2019, Packt Publishing, Birmingham, 2019.

[14] Cinemachine Documentation, Unity Technologies, 2019., https://docs.unity3d.com/Packages/com.unity.cinemachine@2.3/manual/index.html, 31.7.2023.

POPIS SLIKA

Slika 3.1. Arhitektura *game engine*-a [10] 7

Slika 3.2. Središnja petlja *game engine*-a 12

Slika 4.1. Slika zaslona scene za odabir auta 14

Slika 4.2. Snimka zaslona igre 15

Slika 4.3. Svojstva komponente Cinemachine Virtual Camera 17

Slika 4.4. Kalibriranje Cinemachine kamere 19

Slika 4.5. Svojstva Cinemachine Brain komponente 20

Slika 4.6. BoxCollider komponenta 21

Slika 4.7. Scriptable objekt za specifikacije automobila 22

Slika 4.8. Inicijalizacijski dio skripte za upravljanje vozilom 23

Slika 4.9. Metode Update i FixedUpdate u PlayerController skripti 24

Slika 4.10. v – t graf ubrzanja 25

Slika 4.11. Upravljanje vozilom 26

Slika 4.12. Metode za upravljanje ubrzavanjem i usporavanjem 28

Slika 4.13. Metoda za upravljanje skretanjem vozila 29

Slika 4.14. Utvrđivanje smjera kretanja skalarnim umnoškom 30

Slika 4.15. Cijela staza i zidovi kontrolnih točaka 31

Slika 4.16. Zid kontrolne točke 32

Slika 4.17. Struktura podataka za ostvarenje kretanja protivnika 33

Slika 4.18. Inicijalizacijski dio skripte za upravljanje protivnikom 34

Slika 4.19. Metode za upravljanje vozilom protivnika 35

Slika 4.20. Okidanje kolizije i postavljanje iduće kontrolne točke 36

Slika 4.21. Skripta za prilagodbu brzine prema vrsti terena 38

Slika 4.22. Metoda za traženje dominantne teksture 39

SAŽETAK

Ovaj diplomski rad, u pregledu područja govori o tome što je *game engine*, specifično o primjeni Unity Engine-a, njegovom kulturalnom značaju i o igrama utrkivanja. Sljedeće poglavlje predstavlja osnovnu strukturu arhitekture *game engine*-a i za svaki sloj objašnjava neke njegove značajke. Završni dio objašnjava implementaciju igre utrkivanja, objašnjava odluke dizajna igre, izvedbu modela vožnje i vožnju računalnog protivnika.

Ključne riječi:

Arhitektura Game Engine-a, Igra utrkivanja, kulturalni softver, Unity Game Engine

ABSTRACT

Title: Development of a 3D car racing video game

This thesis, in the overview chapter talks about what is a game engine, about application of Unity engine specifically, it's cultural significance and about racing games. The following chapter presents the core structure of game engine architecture and explains some of the features for each layer. The last part goes into explaining implementation of a racing game, explaining game design decisions, making of driving model and computer opponent driving.

Keywords:

Cultural Software, Game Engine Architecture, Racing Game, Unity Game Engine,

ŽIVOTOPIS

Tomislav Markovica rođen je u Bjelovaru 10. kolovoza 1998. Pohađao je Osnovnu školu Ivana Lackovića Croate u Kalinovcu. Nakon osnovne škole, 2013. godine upisuje Srednju strukovnu školu Đurđevac, smjer tehničar za računalstvo. Nakon završetka srednje škole, 2017. godine upisuje preddiplomski sveučilišni studij računarstva na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija Osijek. Po završetku preddiplomskog studija 2021. na istom fakultetu upisuje diplomski studij računarstva smjer programsko inženjerstvo.

PRILOG

Racing Cars Pack 1, Cerberus Studio, https://assetstore.unity.com/packages/p/racing-cars-pack-1-2195

i6 German - FREE Engine Sound Pack, Skrill Studio, https://assetstore.unity.com/packages/audio/sound-fx/transportation/i6-german-free-engine-sound-pack-106037

Props for Track Environment Lowpoly Free, BE DRILL ENTER, https://assetstore.unity.com/packages/3d/props/props-for-track-environment-lowpoly-free-211494#publisher

PBR Texture Lib, Bauervision, https://assetstore.unity.com/packages/2d/textures-materials/pbr-texture-lib-82799#publisher