Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Rok Pajnič

Izdelava realističnih prostorov za vizualizacijo v navidezni resničnosti

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM
PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: izr. prof. dr. Matija Marolt
Somentor: as. dr. Ciril Bohak

Ljubljana, 2019

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

V okviru diplomske naloge preučite področje navidezne resničnosti in izdelave čimbolj realistično izgledajočih prostorov za vizualizacijo v navidezni resničnosti. Pri tem se posvetite tako modeliranju geometrije kot izdelavi materialov in osvetlitvi prostorov. Za primer izdelajte vizualizacijo dela stavbe Fakultete za računalništvo in informatike. V vizualizacijo dodajte tudi interaktivne elemente risanja po površinah in prostoru in upravljanja s predmeti v prostoru.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
VR	virtual reality	navidezna resničnost
3D	three-dimensional	tridimenzionalno
UE	Unreal Engine	Unreal Engine
BP	Blueprint	Blueprint
FRI	Faculty of Computer and Information Science	Fakulteta za računalništvo in informatiko
UMG	Unreal Motion Graphics	Unreal Motion Graphics
AR	augmented reality	obogatena resničnost

Povzetek

Naslov: Izdelava realističnih prostorov za vizualizacijo v navidezni resničnosti

Avtor: Rok Pajnič

Razvoj računalniške grafike je vseskozi spremljala želja po realističnih vizualizacijah. V zadnjih letih že dosegamo rezultate, ki jih težko ločimo od resničnega. Nenehno pa iščemo tudi nove načine, kako te izkušnje realističnih vizualizacij približati uporabnikom. Navidezna resničnost je primer tehnologije, ki skuša uporabnika potopiti v virtualni svet. Namen diplomskega dela je bil zgraditi realističen prostor, v katerega uporabnik lahko vstopi s pomočjo sistema za navidezno resničnost. Kot prostor smo vzeli nam znano okolje, Fakulteto za računalništvo in informatiko. Delo smo začeli z modeliranjem objekta v programu Maya, nato pa smo ga prenesli v pogon Unreal Engine, kjer smo implementirali vso logiko programa. Ciljni sistem za navidezno resničnost je bil HTC Vive. Uporabnik ima na razpolago možnosti risanja po površinah in v 3D prostoru ter postavitve in upravljanja s predmeti v prostoru. Poleg implementacije omenjenih funkcij delo opisuje uporabljene tehnologije in orodja, potek dela in kaj vse moramo upoštevati pri razvoju tovrstne izkušnje.

Ključne besede: navidezna resničnost, VR, Unreal Engine, vizualizacija.

Abstract

Title: Creation of realistic spaces for visualization in virtual reality

Author: Rok Pajnič

The development of computer graphics has always been coupled with the desire to create realistic visualizations. In recent years, we have already been achieving results that are hard to distinguish from reality. However, we are constantly searching for ways to immerse the user even further. Virtual reality is an example of such technology. The purpose of this thesis was to create a realistic environment that a user can experience with a virtual reality system. We took a familiar setting for our scene, the Faculty of Computer and Information Science in Ljubljana. We began by modeling in Maya, and from there transferred the environment into Unreal Engine, where the implementation of the logic took place. Our target virtual reality system was HTC Vive. The user has the options of painting on surfaces and in 3D space, as well as arranging and interacting with objects in the scene. In addition to the implementations of the said functions, the thesis describes the tools, technologies and workflow, along with the main guidelines to be followed when developing these types of experiences.

Keywords: virtual reality, VR, Unreal Engine, visualization.

Poglavje 1

Uvod

Ljudje svet zaznavamo s čutili, ki jih ponavadi delimo na vid, tip, voh, okus in sluh. Pri uporabi računalniških naprav pa se običajno zanašamo zgolj na vid in sluh. Od pojavitve prvih osebnih računalnikov v drugi polovici 20. stoletja pa vse do danes z računalniki komuniciramo preko vhodno-izhodnih naprav, kot so tipkovnica, miška ali zasloni na dotik. Bližamo se tudi že pogovoru. Vse povratne informacije in multimedijske vsebine dojemamo z očmi in ušesi.

Z napredkom tehnologije in zmogljivosti računalnikov je porasla tudi količina in zahtevnost opravil, ki jih izvršujemo na računalniških napravah. Ena od smeri, ki je bila v zadnjih letih deležna močnega napredka, je tudi računalniška grafika. Ta razvijalcem omogoča izdelavo realističnih 3D vsebin, ki se jih uporablja v mnogih različnih industrijah. Pogost cilj teh je, da se vizualno čim bolj približajo realnemu svetu in uporabnika pritegnejo v svoje okolje in zgodbo. Za primer panog, ki gradijo na visokorealistični grafiki, lahko vzamemo filmsko in igričarsko industrijo, ki skušata z vizualnimi učinki gledalca oz. igralca čim bolj potopiti v svojo okolje, ali pa industrijo 3D vizualizacij, kjer modelirajo visokodetajlne modele izdelkov ali prostorov.

Naša ambicija je čim bolj zabrisati mejo med realnim in digitalnim, pri čemer nam pomaga tehnologija navidezne resničnosti (angl. *virtual reality* – VR). Ta za razliko od konvencionalnih tehnologij, ki komunicirajo le z

našim vidom in sluhom, zaposli tudi naš ravnotežni organ. Uporabniki te tehnologije sami postanejo del navideznega sveta. To odpira veliko vrat, saj uporabnikom vzbudi bolj poglobljen občutek resničnosti in vpletenosti v okolje.

V diplomskem delu smo skušali ustvariti vizualno realističen prostor in ga s pomočjo tehnologije VR skupaj z uporabnikom umestiti v svet navidezne resničnosti. Hkrati smo prostor dodatno obogatili z interaktivnimi vsebinami. Na poti do tega se najprej osredotočimo na orodja in tehnologije, ki smo jih uporabili za realizacijo cilja, postopek priprave prostora oz. modelov in na potek dela. Sledi razčlemba implementacije zadanega cilja. Zaključek zajema sklepe o izdelku, predlagane možnosti za izboljšavo in splošen vtis o izdelku.

1.1 Navidezna resničnost

Ideja navidezne resničnosti se zdi kot moderen koncept, ki je le pred kratkim prišel na trg do uporabnikov. Kljub temu pa ambicija, da tehnologija ljudi popelje v neko simulirano okolje, ni nova.

Predhodnike sodobne navidezne resničnosti lahko najdemo že v 19. stoletju. Leta 1838 je angleški znanstvenik in izumitelj Charles Wheatstone zasnoval prvi *stereoskop*. Stereoskop je naprava, skozi katero uporabnik gleda na dve sliki, ki ju možgani združijo v eno. To gledalcu da vtis globine in perspektive. Skozi stoletje so se koncepti stereoskopa nadgrajevali. Pojavili so se novi poizkusi stereoskopov, najbolj prepoznaven pa je morda *View Master* (slika 1.1).

Na področju simulacij lahko omenimo *Link Trainer*, zgodnji simulator letenja, ki je bil ustvarjen leta 1929. Namenjen pa ni bil ravno zabavi, temveč za urjenje pilotov v vojski. Domišljijo o navidezni resničnosti je pisatelj znanstvenofantastičnih knjig Stanley G. Weinbaum ubesedil v knjigi *Pygmalion's Spectacles*. Zamislil si je očala, z uporabo katerih bi uporabnik lahko simuliran svet fizično občutil, zavohal in okusil. Nosilec naprave naj ne bi

Slika 1.1: View Master Model G (vir: Wikipedia [40])

razločeval med navideznim in resničnim.

Sredi 20. stoletja pa se je pojavila nova ideja, ko je kinematograf Morton Heilig zasnoval napravo *Sensorama*. To je bila cela kabina, kjer si je gledalec lahko ogledal nabor šestih filmov. Sensorama je bila opremljena s stereoskopskim zaslonom, zvočniki, ventilatorji, tresočim stolom, proizvajala pa je celo vonjave. Heilig je leta 1960 patentiral tudi prvi naglavni prikazovalnik (angl. *head-mounted display* – HMD). Njegova funkcionalnost je bilo zgolj predvajanje vsebin, brez zaznavanja gibanja. Leto pozneje so očala *Headsight* ta izziv rešila z zaznavanjem gibanja s pomočjo magnetov. Tudi očala Headsight so bila razvita za uporabo v vojski, in sicer za opazovanje nevarnih situacij, saj so lahko z očali nadzorovali oddaljeno kamero in opazovali zajeto sliko.

Leta 1965 je računalniški inženir Ivan Sutherland predstavil svojo vizijo za navidezno resničnost v članku *The Ultimate Display* [26]. Njegov koncept je predstavil tako navidezno resničnost, da uporabnik ne bi razločil simulacije od resničnosti. Njegova vizija je bil prostor, kjer bi računalniki lahko nadzirali obstoj materije. Uporabniki in generirani objekti bi vzajemno vplivali eden na drugega, zadetek naboja v takem prostoru pa bi bil, kot je

Slika 1.2: Damoklejev meč (vir: Vrroom [38])

zapisal Sutherland, lahko smrtonosen. Članek je postal temelj tega, kar si prizadevamo doseči na področju navidezne resničnosti.

Tri leta pozneje sta Sutherland in njegov študent Bob Sproull izdelala prva očala za VR. Imenovala sta jih *Damoklejev meč* (angl. *The Sword of Damokles*). Podobno kot v istoimenski prispodobi iz starogrške zgodovine, kjer je meč visel nad Damoklejevo glavo, so bila tudi očala del konstrukcije, ki je visela nad uporabnikom, saj je bil sistem pretežak in preneroden, da bi jih bilo možno nositi samostojno (slika 1.2). Naprava je bila priklopljena na računalnik in je lahko prikazovala le enostavne like, ob premiku prikazovalnika pa se je spreminjala tudi perspektiva v navideznem prostoru. Sistem lahko označimo tudi kot začetnika obogatene resničnosti (angl. *augmented reality* – AR), saj je izrisovanje likov potekalo skozi prosojne leče. Zaradi nepraktičnosti uporabe se projekt ni razširil izven prostorov raziskave.

Računalniški inženir in umetnik Myron Krueger je do konca sedemdesetih let prejšnjega stoletja razvil tudi platformo *Videoplace*, kjer se je računalniško simulirano okolje odzivalo na ljudi v okolici. Sistem, tokrat brez očal, je na podlagi kombinacije računalniške grafike, kamer, projektorjev in prostorskih senzorjev simuliral virtualno okolje s pomočjo senc.

V teh letih so se pojavili tudi različni idejni nasledniki že omenjenega simulatorja letenja Link Trainer. Glavni pobudnik je bila vojska, saj je želela

uriti svoje vojake v varnem okolju. V ta namen so bile razvite več milijon dolarjev vredne naprave. Od očal za AR, ki lahko sledijo očem uporabnika, do velikih čelad za VR, ki so poizkušale simulirati letenje.

Pojem „navidezna resničnost“ kot tak se je uveljavil šele sredi osemdesetih let. Jaron Lanier in Thomas Zimmerman sta leta 1987 ustanovila podjetje *VPL (Visual Programming Lab)*, ki je znano kot prvo podjetje, ki je začelo s komercialno prodajo VR očal (EyePhone) in rokavic (Dataglove).

V začetku devetdesetih let prejšnjega stoletja so se na trgu začele pojavljati virtualne naprave in vsebine za širšo javnost. Tematika navidezne resničnosti je dosegla filmsko industrijo (npr. filmi Kosec, Tron in Matrica), tehnologija pa se je pojavila tudi na arkadnih igrah. Samostojen razvoj naprav za VR so začeli tudi izdelovalci igralih konzol, kot sta SEGA in Nintendo. SEGA je leta 1993 naznanila VR očala za svoje arkadne igre in konzolo Genesis. Na tržišče pa je prispela zgolj različica za arkadne igre. En zanimiv argument podjetja za to je bil, da je izkušnja ob uporabi preveč realistična in bi bila konzola prenevarna za domačo uporabo. Glede na omejeno zmogljivost strojne opreme v tistem času temu seveda težko verjamemo.

Nintendo je leta 1995 na tržišču ponudil svojo prenosno napravo za VR *Virtual Boy* (slika 1.3). Delovala je na principu stereoskopov in imitiranja 3D slike. Kljub imenu konzola ni ponujala navdušujoče navidezne resničnosti, kar je bil glavni razlog poleg enobarvne grafike, skromne izbire iger, visoke cene, neudobja pri igranju in ostalih razlogov, da naprava ni bila deležna uspeha. V manj kot letu po izdaji so konzolo umaknili iz prodaje. V svetu VR so se preizkusila tudi podjetja, kot so Atari in Victormaxx, nobeno od teh pa se ne more pohvaliti z dobrim prodajnim uspehom.

Skočimo v manj oddaljeno preteklost. VR se je znova aktualiziral leta 2010, ko je Palmer Luckey izdelal prvi prototip očal za VR *Oculus Rift*. Leta 2012 je Luckey projekt objavil na platformi Kickstarter, kjer je do konca kampanje zbral okoli 2,4 milijona dolarjev. Priložnost v tehnologiji navidezne resničnosti so prepoznala tudi druga podjetja. Facebook je leta 2014 kupil podjetje Oculus VR za 2,3 milijarde dolarjev, Samsung in Google

Slika 1.3: Nintendo Virtual Boy (vir: Wikipedia [41])

sta skušala VR popularizirati na mobilnih napravah (Google Cardboard in Samsung Gear VR). Sony je začel razvoj VR očal za svojo konzolo Playstation, HTC pa je leta 2015 predstavil svoj sistem VR, ki še vedno konkurira Oculusu. HTC Vive omogoča prosto gibanje po prostoru in vključuje dva nadzorna ploščka.

Razvoj naprav za navidezno in obogateno resničnost je v polnem teku, vodijo pa ga velika in mala podjetja. Tehnologiji imata namreč velik potencial v filmski in igričarski industriji, uporabimo ju lahko v zdravstvene namene, pri zdravljenju psihičnih težav, pa tudi za urjenje v zahtevnih strokah (npr. kirurgi, vojaki in piloti) v simuliranem okolju.

Gonilna sila na področju VR je tudi Kitajska. Navidezna resničnost je namreč ena od tehnologij v Kitajskem petletnem načrtu (2016–2020) državnega razvoja. Vključena področja so v teh petih letih v fokusu financiranja in intenzivnejšega razvoja. Za tem stojijo tudi ogromna podjetja, kot so Baidu, Alibaba in Tencent, ki imajo moč narekovati potek celotne industrije. Do konca leta 2022 je vrednost trga tehnologij AR in VR ocenjena na okoli

209,2 milijarde dolarjev [36].

Primere najsodobnejše uporabe tehnologij AR in VR smo si imeli priložnost ogledati v Autodeskovem tehnološkem centru odličnosti na področju VR v Münchnu v Nemčiji. Po besedah enega od predstavnikov največji potencial navidezne resničnosti vidijo v sodelovanju večjega števila ljudi na oddaljenih lokacijah v virtualnem okolju. Preizkusili smo njihov nabor projektov VR in AR, ki združujejo več različnih tehnologij.

Prostor, v katerem smo se nahajali, je bil poustvarjen in prenesen v navidezno okolje, ki smo ga lahko videli na zaslonih in s pomočjo očal za VR. Opremljen je bil tudi s kamerami in monitorji. To omogočala, da uporabniki z ali brez očal za VR vidijo, kar se dogaja v 3D prostoru in z njim tudi upravljajo. Na voljo smo imeli tudi nabor predmetov, ki jih kamere v prostoru zaznavajo. Z njimi smo lahko v simuliranem okolju vplivali na predmete, kot so kamera, luči ali druga orodja, specifična za predstavitev. Projekti so bili torej kombinacija navidezne in obogatene resničnosti, kjer smo lahko vsi prisotni v prostoru vplivali na to, kar se dogaja v sceni, obenem pa je med nami lahko tekla diskusija. Govorili smo tudi o pomembnih faktorjih, na katere je treba paziti pri izdelavi okolja za navidezno resničnost, ki jih bomo razčlenili v nadaljevanju tega dela. Center je opremljen tudi s sodobno opremo za čim boljšo izkušnjo VR:

- zmogljivi računalniki, ki poganjajo celoten sistem;
- brezžični adapter za VR očala, s katerim ne potrebujemo povezave na računalnik;
- kamere za sledenje predmetom.

Glavni igralci na področju navidezne resničnosti so trenutno: HTC s sistemom Vive (Pro), Oculus VR s sitemom Rift in Sony s svojimi očali za konzolo Playstation. To pa ne pomeni, da se stvari ne bodo kmalu spremenile. Na pohodu so že novi sistemi, ki obljubljajo vse boljšo izkušnjo in funkcije.

Slika 1.4: Prostor v Autodeskovem centru (vir: Youtube [43])

1.2 Cilji in potek dela

Glavni cilj naloge je bil ustvariti realistično 3D sceno notranjosti Fakultete za računalništvo in informatiko, v katero uporabnik lahko vstopi v navidezni resničnosti. Da je izkušnja bolj interaktivna in zanimiva, se uporabnik v njej lahko premika, komunicira z okolico in jo spreminja. Zadane funkcionalnosti so bile:

- risanje po površinah;
- risanje v prostoru;
- upravljanje s predmeti.

Čeprav so ciljne platforme naprave za VR, je bil cilj program narediti tako fleksibilen, da bo deloval tudi na računalnikih brez namenske naprave, torej na navadnem zaslonu ter z miško in tipkovnico.

Delo na tovrstnem projektu je združevalo več disciplin, zato je bilo potrebno cilje in postopke dela dobro zastaviti vnaprej. To nam je pomagalo,

da v fazi razvoja nismo zabredli pregloboko v napačno smer in da je delo potekalo bolj tekoče. Delo smo razdelil na tri sklope:

1. razvoj funkcionalnosti;
2. gradnja in delo na okolju scene;
3. testiranje in evalvacija obeh predhodnih faz.

To je bil cikel razvoja, ki se je skozi iteracije testiranja ponavljal do ustrezne mere zadovoljstva. Ponovitve testiranja so zahtevale tudi menjavo okolij razvoja, zato je smiselna dobra urejenost datotek.

1.3 Struktura dela

Delo je zajemalo kombinacijo različnih tehnologij, razvoj pa je temu ustrezno potekal v več fazah. V vsaki fazi je potekalo delo na specifični funkciji ali segmentu končnega izdelka. Diplomsko delo je razdeljeno na poglavja, ki skušajo odražati te segmente in jih razčleniti.

V delu najprej omenimo in na kratko opišemo glavna orodja in tehnologije, ki smo jih uporabili. Nadaljujemo z implementacijo, ki je jedro našega dela. V poglavju opišemo vse faze in rezultate dela. Začetek poglavja zajema modeliranje in pripravo scene. Sledi podpoglavje implementacije, ki je bolj programersko naravnano. Zadnja poglavja opisujejo bolj vizualne dele naloge. Mednje sodijo izdelava uporabniškega vmesnika, osvetljevanje scene in izdelava materialov. Zadnji del zajema testiranje izdelka in možne ukrepe za izboljšavo izkušnje. Sledi povzetek implementacije, ki bolj jedrnato povzame delo na izdelku in predstavi primer uporabe izdelanega programa. V zaključku delo še ovrednotimo in podamo možne izboljšave.

Poglavje 2

Uporabljena orodja in tehnologije

2.1 Unreal Engine

Unreal Engine (UE) je orodje podjetja Epic Games [5]. Prva različica igralnega pogona je bila uporabljena v prvoosebni strelski igri Unreal, ki je izšla leta 1998. Najsodobnejša različica je bila izdana leta 2014, razvijalcem pa je bila na voljo preko naročniškega modela. Velik mejnik za UE je bilo leto po izdaji, ko je orodje postalo odprtokodno in prosto dostopno vsem uporabnikom. Epic Games je s tem začel tudi z novim poslovnim modelom, in sicer ob zaslužku, večjem od 3000 $, gre 5 % delež podjetju Epic Games. Zanimivo je omeniti tudi, da je Oculus VR naznanil kritje teh deležev na vse izdane izdelke, ki so objavljeni v njihovi trgovini Oculus Store, vse do vsote 5 milijonov dolarjev [16]. Razvoj v UE poteka bodisi v programskem jeziku C++ bodisi v njihovem sistemu vizualnega programiranja *Blueprint* (BP). Unreal Engine je skozi svoj obstoj prejel že veliko nagrad in priznanj, večkrat tudi za najboljši igralni pogon [32, 8].

Čeprav je Unreal Engine v prvi vrsti orodje za razvoj iger, ga razvijalci vse pogosteje uporabljajo tudi za bolj industrijsko usmerjene projekte. Interaktivni projekti, kot so na primer konfiguratorji izdelkov, vizualizacija prosto-

rov in animacije, so priljubljeni predvsem na področjih avtomobilizma [31], arhitekture [30] in filmske industrije [29]. Epic Games je s tem v mislih zasnoval novo različico pogona Unreal Engine, ki je namenjena predvsem industrijski uporabi. Poimenovali so ga *Unreal Studio* [34]. Najpomembnejši del je zagotovo nabor orodij, ki ga imenujejo *Datasmith*. Je pravzaprav skupek vtičnikov za pogon, ki omogočajo hitro uvažanje podatkov CAD ter na splošno geometrije in topologije v najpogosteje uporabljenih formatih iz programov, ki veljajo za industrijske standarde. To so na primer 3ds Max, SolidWorks, Inventor in podobni. Dodali so tudi programiranje skriptov v Pythonu, kar še dodatno razširi dostopnost in uporabnost programa. Unreal Studio je v času pisanja v beta fazi razvoja, licenca pa je brez finančnih obveznosti do podjetja Epic Games.

Ker je arhitektura prostorov in sama geometrija modela pomemben del tega diplomskega dela, smo tudi sami preizkusili in uporabili Unreal Studio. To je omogočilo hitro uvažanje modela ob vsaki iteraciji popravkov na modelu.

V nadaljevanju sem ob omembi pogona Unreal Engine še vedno uporabil oznako UE*, čeprav sem uporabljal omenjeno različico Unreal Studio. Kljub drugačnemu poimenovanju gre še vedno za modificirano verzijo prvotne različice.*

Pri izdelavi je bila uporabljena različica pogona 4.20.

2.2 Maya

Program Maya [13] podjetja Autodesk je orodje za modeliranje, animacijo, simulacijo in upodabljanje 3D modelov in scen. Maya je prvič izšla leta 1998 v okviru podjetja Wavefront Technologies [42]. Po več menjavah lastništva je omenjeno tehnologijo leta 2005 kupilo podjetje Autodesk [3]. Maya je na voljo za operacijske sisteme Windows, Linux in iOS. Maya je bila v našem primeru uporabljena za modeliranje večine predmetov, uporabljenih v 3D sceni.

2.3 HTC Vive

HTC Vive je eden izmed kompletov za navidezno resničnost. Razvilo ga je podjetje HTC [9] v sodelovanju z Valve Corporation [35]. Set zajema očala Vive, dva krmilnika in dve postaji za zaznavanje (t.i. Lighthouse). HTC Vive je na tržišču pristal leta 2016. Očala so opremljena z dvema zaslonoma AMOLED, vsak ima ločljivost 1080 x 1200. V začetku leta 2018 je HTC izdal tudi naprednejšo verzijo svojih očal Vive, ki je dobila pripono Pro. Najpomembnejša izboljšava pri tej različici je višja ločljivost zaslonov (1400 x 1600). Opremo Vive poganja Valvov sistem SteamVR [24], ki ga podpirata tudi pogona Unreal Engine in Unity.

2.4 Unreal C++

Primarni programski jezik v pogonu UE je C++. Uporabljen je zato, ker predstavlja dobro razmerje med zmogljivostjo in enostavnostjo uporabe. Večina iger in programov, ki so narejeni v UE in sorodnih pogonih, je običajno zelo kompleksnih in zahtevnih. Nizkonivojska narava jezika dovoljuje doseganje višjih zmogljivosti in nasploh večjo fleksibilnost, ki ju pri teh programih potrebujemo. Za razliko od UE pogon Unity uporablja za programiranje logike jezik C#, ki je višjenivojski jezik. Kljub temu pa je tudi pogon Unity pod površjem programiran v C++ ravno zaradi omenjenih potreb po zmogljivosti. C++ je tudi objektno usmerjen programski jezik, kar olajšuje implementacijo kompleksnejše logike v sceni.

UE programerjem ponuja svojo razširitev standardnega jezika C++, ki so jo poimenovali *Unreal C++*. Pomemben del teh razširitev so nekatere nove implementacije obstoječih podatkovnih tipov, ki skušajo poenotiti obnašanje podatkov na različnih platformah zavoljo boljše prenosljivosti. Praktični so tudi t. i. *makri*, ki omogočajo hitre implementacije logike, ki se navezujejo na sam pogon. Za prevajanje kode moramo imeti nameščena orodja Visual Studio (oziroma XCode za MacOS uporabnike).

Poglavje 3

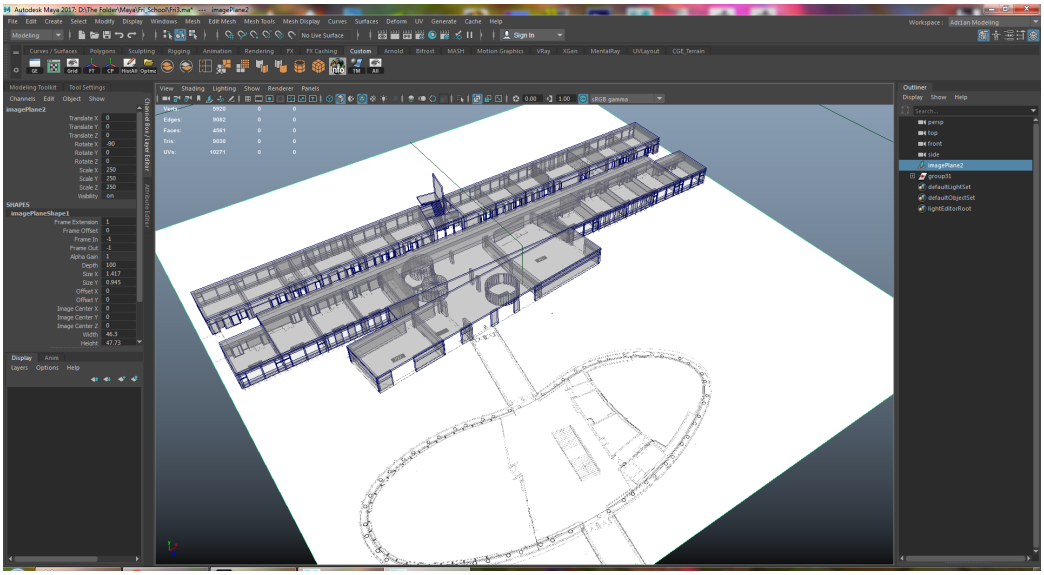
Implementacija

3.1 Modeliranje in priprava modela

Modeliranje in urejanje geometrije prostorov fakultete je potekalo v programu Maya. Ključni del so bili arhitekturni načrti celotnega objekta in razporeditev prostorov. S pomočjo teh smo lahko razbrali vse potrebne mere in razmerja predmetov. Za pomoč pri modeliranju, natančneje pri postavitvi geometrije, smo sliko tlorisa poslopja kar uvozili v program Maya in jo projicirali na ravnino. Tako smo lahko direktno modelirali čez sam načrt. Postopek smo ponovili za vsa tri nadstropja Fakultete za računalništvo in informatiko. Stranske perspektive smo razbirali posebej iz načrtov.

Ker modeliranje ni bilo bistvo naloge in je za manj izkušene uporabnike tovrstnih programov zahteven in dolgotrajen proces, smo se odločili natančneje poustvariti le prvo nadstropje. Preostali dve sta modelirani bolj površinsko, in sta za uporabnike ostali nedostopni.

Pomemben del v fazi modeliranja je bila tudi pravilna priprava tako imenovanih UV-jev. Črki U in V označujeta obe osi, ki sta uporabljeni pri 2D teksturah. UV zemljevid je projekcija geometrije v tridimenzionalnem prostoru na ravnino. Ti podatki povedo, kako se bodo morebitne teksture razporedile po modelu. To je pomembno v procesu dodeljevanja materialov po predmetih v modelu, pa tudi pri osvetljevanju. Večji del izdelave materialov



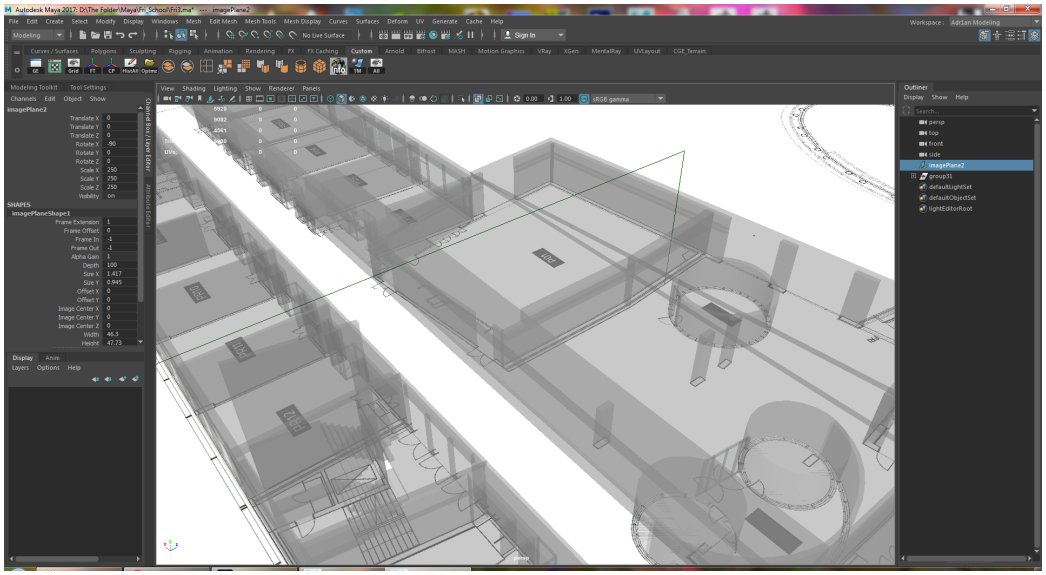
Slika 3.1: Uporaba načrta na ravnini za lažje modeliranje

za naše modele je potekal v pogonu UE.

Čeprav scena ni vsebovala pretirano velikega števila poligonov (le okoli 15.000 trikotnikov), je kljub temu model proti koncu štel že lepo število posameznih elementov geometrije (preko 600). Zaradi urejenosti in lažje orientacije po sceni je koristna tudi smiselna hierarhija in grupiranje predmetov, veliko sorodnih delov pa je bilo tudi združenih v skupne. Na koncu smo se zadovoljili s slabimi 200 elementi geometrije v modelu.

Za izvoz scene smo si pomagali tudi s programom 3ds Max. Razlog za to je bil, da Epic Games zanj ponuja orodje za izvoz scene v formatu *.udatasmith*. Za program Maya ekvivalentno orodje v času pisanja še ni bilo na voljo. Sceno smo poslali v 3ds Max in jo od tam izvozili skupaj z morebitnimi materiali, informacijami o modelu in hierarhiji. Izvožene datoteke lahko zatem uvozimo v UE. Omenjeni format nudi tudi številne prednosti v primerjavi z običajnimi formati, kot je na primer FBX. Za naše potrebe so bile pomembne naslednje prednosti:

- Preprosto urejanje geometrije:



Slika 3.2: Faza izdelave prvega nadstropja

Unreal Studio omogoča osnovno urejanje geometrije, s čimer zmanjšamo skakanje med programoma UE in Maya.

- Hitro uvažanje:
  Ob iteracijah popravkov modela lahko spremembe hitro ponovno uvozimo v sceno, morebitne popravke posameznih delov modelov pa se ohranijo.

- Ohranitev hierarhije:
  Ob uvozu je model še vedno urejen v svoje skupine.

3.2 Priprava scene za VR

Pri izdelavi scene, ki bo namenjena za uporabo v VR moramo za čim boljšo izkušnjo upoštevati nekaj smernic.

3.2.1 Velikost

Ker je eden glavnih ciljev navidezne resničnosti, da se uporabnik počuti kot del virtualnega sveta, je pomembno, da imajo svet in vsi predmeti okoli njega primerno, sorazmerno velikost. Objekti v sceni naj bi karseda točno odražali svojo podobo v realnemu svetu. Najbolje je, da imamo to v mislih že v fazi modeliranja in geometrijo izdelujemo vsaj v pravem razmerju, če ne celo v dejanski velikosti. Pomembno je, da v programih za modeliranje ustrezno in enotno nastavimo enote, v katerih modeliramo. Za mersko enoto v UE velja *Unreal Unit* (UU), ki predstavlja 1 cm. S tem v mislih smo tudi modelirali in izvažali v centimetrih in se tako izognili kasnejšim popravkom velikosti modelov.

3.2.2 Zmogljivost

Poganjanje scene v navidezni resničnosti je večji zalogaj, kot se morda sprva zdi. Standardni način upodabljanja (izrisovanja scene) za razliko od običajnega upodabljanja (v scenah, ki niso VR) poteka dvakrat. Enkrat za vsako od obeh leč. Ker očala pokrijejo celotno vidno polje, je pomembno tudi, da je izkušnja karseda tekoča. Tako za *Oculus Rift* kot za *HTC Vive* velja priporočeni minimum 90 sličic na sekundo [37]. V primeru, da te meje ne dosegamo, uporabniku hitro postane slabo, kar seveda pomeni slabo izkušnjo.

3.2.3 Simulacijska slabost

Simulacijska slabost je oblika gibalne slabosti, do katere lahko pride pri uporabi očal za VR. Lahko jo povzroči negladko delovanje naprav ali pa vizualni efekti, ki niso primerni za uporabo v VR. Do neprijetne izkušnje in slabosti lahko pride tudi ob neidealnih nastavitvah očal in leč (na primer pozicija očal na glavi ali razdalja leč).

Cilj je sceno pripraviti tako, da se čim bolj približamo realni izkušnji, kar pomeni da moramo odmisliti nekatere vizualne efekte, ki so sicer pogosto prisotni v 3D vizualizacijah. Nekaj smernic, čemu se izogibati:

- Odvzemanje nadzora kamere:
  Vse, kar uporabniku povzroča nenadzorovano premikanje kamere, bo nanj delovalo nenaravno. Izogibamo se tudi efektom tresenja kamere.

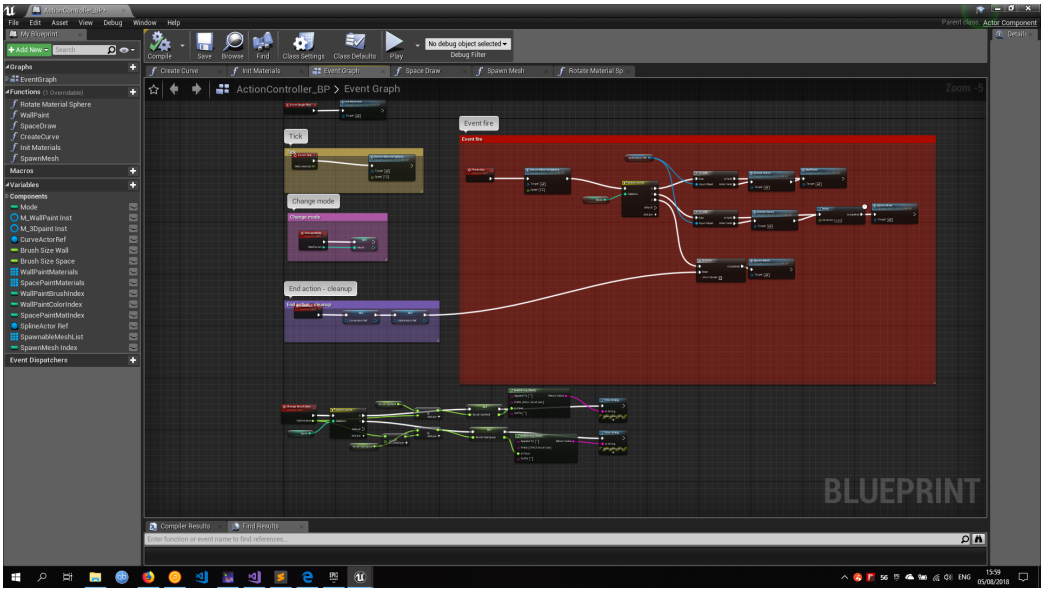
- Pretirana osvetlitev:
  Premočne luči oz. osvetljava lahko slepi uporabnika in povzroča slabost. Poskusimo uporabiti manj svetle in hladnejše luči.

- Postprodukcijski učinki:
  Veliko tovrstnih učinkov se lahko uporabniku zdi nenaravnih in lahko povzroča slabost. Med glavne primere spadajo globinska ostrina (angl. *depth of field* – cinematični učinek, ki posnema fokus leče na določeni razdalji, ostala scena pa je zamegljena), zameglitev zaradi gibanja (angl. *motion blur* – megljenje scene ob hitrih premikih kamere) in učinek lečenja (angl. *lens flare* – odbijanje svetlobe zaradi nepopolnosti leč). Veliko učinkov izklopimo tudi zaradi boljše zmogljivosti.

3.3 Funkcionalnosti

Čeprav je bil velik poudarek na izdelavi modela in okolja, bi se zgolj opazovanja okolice hitro naveličali. Zato je bil naš cilj narediti virtualno okolje tudi interaktivno in dati uporabniku nekaj, kar lahko počne. V naslednjem razdelku razčlenimo implementacijo zadanih funkcionalnosti.

Za vse večje dele logike funkcij je naša implementacija potekala v dveh fazah. V prvi smo logiko implementirali v sistemu *Blueprint*. V tem okolju lahko hitro pridemo do rezultatov, jih testiramo v sceni ter vidimo, kaj deluje, kaj ne in na kakšne težave naletimo. Ob morebitnih napakah tudi hitreje pridemo do hroščev in problemov, na katere nismo računali v fazi načrtovanja. Ko smo bili z delovanjem prototipa zadovoljni, smo sisteme prenesli v C++ kodo. To je bila druga faza implementacije. S prenosom kode v C++ lahko dosežemo občutno hitrejše delovanje, hkrati pa v program vključimo le potrebne module. Sicer je Epic Games v svoj pogon že

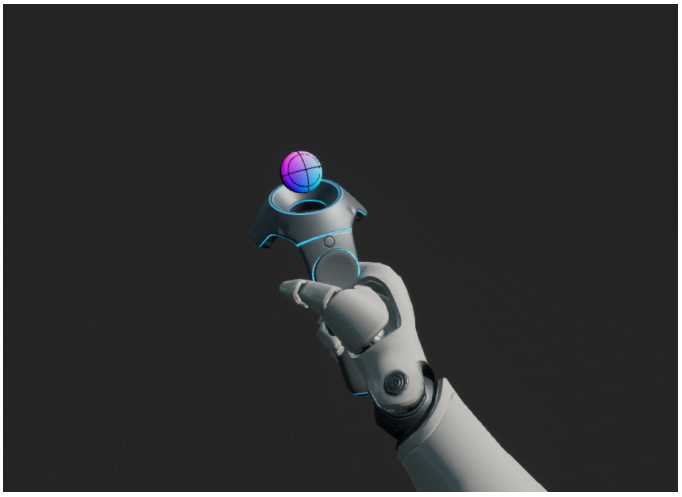


Slika 3.3: Primer prototipa glavnega modula delovanja

implementiral samodejno pretvorbo blueprintov v kodo C++, vendar smo se kljub temu odločili za lastno implementacijo, saj smo imeli s tem več nadzora nad delovanjem. Logika pa kljub temu ni implementirana zgolj v C++. Dobra praksa je ugotoviti, kaj implementirati kot blueprint in kaj v C++, saj s kombinacijo obeh sistemov dobimo najboljše lastnosti vsakega. V naši izvedbi smo logiko funkcij izvedli v C++ komponenti, vse bolj vizualne dele, kot so reference na uvožene predmete in parametre zagona komponente, pa smo izvedli v blueprintih, saj omogočajo bolj agilen pristop do morebitnih sprememb v primerjavi s kodo. Poleg tega smo blueprinte uporabili tudi pri nekaterih povezavah med različnimi sistemi in razredi.

3.3.1 Glavni lik

Naša izkušnja je namenjena tako navidezni resničnosti kot tudi običajni uporabi, zato smo tudi uporabniški lik implementirali za oba načina, enega za vsak način izkušnje. Ob zagonu preverimo izbran način in glede na to aktiviramo ustrezni lik. Vsak od njiju ima svojo logiko premikanja, svoj upo-



Slika 3.4: Prikaz nadzornega krmilnika

rabniški vmesnik in svoje vhode za nadzor lika. Standardni lik, ki ni namenjen načinu VR, nadzorujemo skozi perspektivo prvoosebne kamere, da bi bila izkušnja čim bližja navidezni resničnosti.

V obeh primerih uporabljamo nadzorni krmilnik, ki predstavlja izhodišče naših funkcij, hkrati pa služi tudi za orientacijo, zlasti v načinu VR. Za model tega krmilnika smo vzeli kar model krmilnika Vive, ki je dostopen kot del pogona UE. Model ploščka pa spremlja tudi krogla na vrhu krmilnika. Namenjena je vizualizaciji nekaterih informacij samih funkcionalnosti, poleg obstoječega uporabniškega vmesnika. Razlog takega vizualnega prikazovanja informacij je tudi v tem, da skušamo uporabniku narediti izkušnjo še bolj zanimivo in dinamično.

3.3.2 Akcijska komponenta

Srce in izhodišče vseh zastavljenih funkcij leži v tako imenovani *akcijski komponenti* (angl. *action controller*). Predstavlja jo samostojni razred, v sceni pa je komponenta del uporabnikovega lika. Komponenta hrani stanje in vse informacije, potrebne za vsako implementirano funkcionalnost.

Uporabnikov lik komunicira s komponento preko dveh spremenljivk:

- *bIsFiring* – tipa boolean, ki komponenti sporoča, da od nje zahtevamo delovanje;

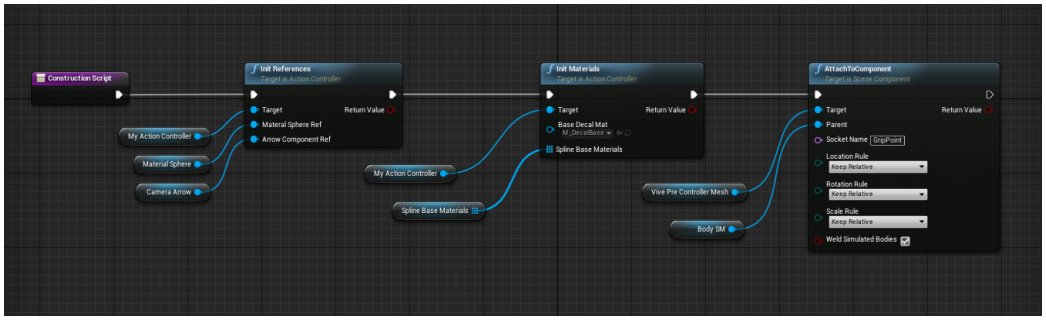
- *mode* – spremenljivka tipa int32 (UE verzija tipa int), ki hrani indeks trenutno aktivne funkcije.

Preden lahko začnemo komponento uporabljati, jo moramo v njeni starševski komponenti inicializirati, kjer ji nastavimo vse potrebne reference. V ta namen sta na voljo dve funkciji:

- *InitReferences* – funkcija, ki pomaga vzpostaviti reference na predmete, ki so del nadrejene komponente (v našem primeru uporabnikovega lika). Referenci, ki ju moramo vzpostaviti sta referenca krogle na krmilniku in referenca na vektorsko komponento (angl. *arrow component*), iz katere pridobivamo informacijo o usmerjenosti krmilnika, kar koristi pri implementacij naših funkcij;

- *InitMaterials* – funkcija, ki je namenjena dinamičnemu ustvarjanju vseh materialov, ki jih potrebujemo pri implementaciji funkcij.

Celotno zaganjanje oz. inicializacija poteka v konstrukcijskem skriptu (angl. *construction script*) aktivnega lika v sistemu *Blueprint* (glej sliko 3.5). Za razliko od uporabnikovega lika ima naša akcijska komponenta *funkcijo Tick*. To je posebna funkcija, ki je delno že implementira s strani pogona UE in se (ponavadi) sproži enkrat na vsako izrisano sličico ter izvaja poljubne ukaze. To je koristno, ko želimo neko dejanje izvajati neprestano in tekoče v nekem časovnem intervalu. Komponenta deluje tako, da vsako sličico preveri, ali je zastavica *bIsFiring* resnična ali neresnična. V primeru, da je, se glede na nastavljeni način prične izvajati izbrano dejanje. Zastavico nastavlja uporabnikov lik preko določenega vhoda (na primer ob kliku na levi miškin gumb se zastavica postavi na *true*, ob spustu gumba pa na *false*).



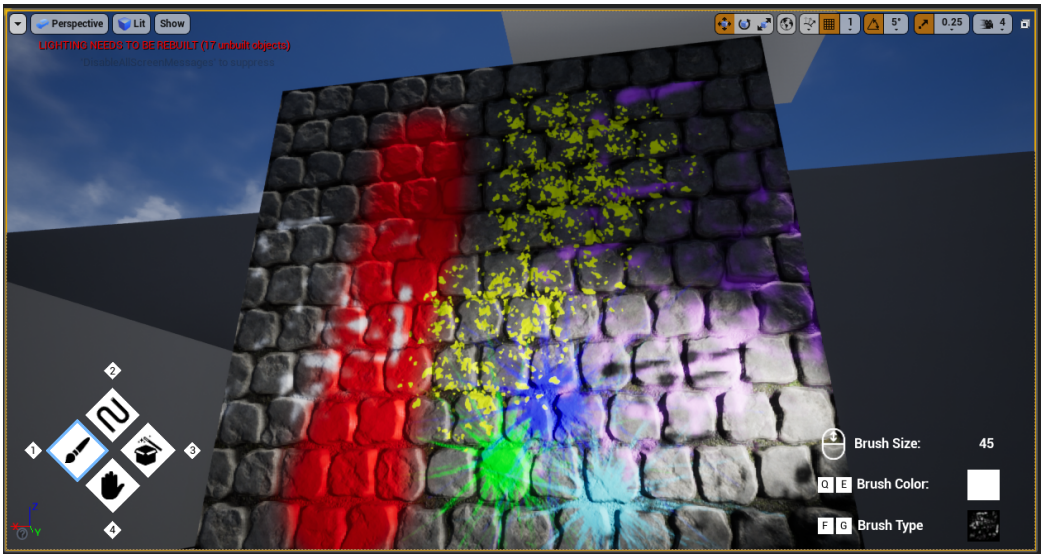
Slika 3.5: Inicializacija glavne komponente

3.3.3 Risanje po površinah

Ideja prve funkcije je možnost poslikave poljubnih površin v sceni. Temu smo dodali še barvno paleto, kjer lahko uporabnik uporabi različne barve, poleg tega pa izbiro razširimo še z naborom različnih tipov in oblik čopiča.

Prototip implementacije smo testirali z dvema različnima metodama. Prvi pristop je bila rešitev s sistemom *Render target*. Tega lahko povzamemo kot posebno teksturo, skupino točk, ki jih lahko prilagodimo v času delovanja programa (angl. *runtime*). Sistem implementiramo tako, da ustvarimo omenjeno teksturo in nanjo rišemo s temu namenjenim materialom, ki vsebuje informacije o barvi, velikosti in lokaciji barvanja. Ob testiranju se je rešitev sicer izkazala za dobro, ima pa nekaj pomanjkljivosti. Ker je bil cilj, da uporabniku omogočimo risati po kateri koli površini v sceni, se je pri tej metodi pojavil problem, saj moramo za vsako površino, na katero hočemo risati, ustvariti novo pojavitev (instanco) omenjene teksture. To bi zahtevalo veliko dodatne logike za dinamično ustvarjanje te teksture, hkrati pa bi bilo to tudi bolj zahtevno za poganjanje. Poleg tega je tekstura odvisna tudi od geometrije, na katero je pripeta, kjer lahko znova naletimo na težave, na primer zaradi neenotnih ali slabih UV-jev.

Na podlagi teh ovir in naših potreb smo se odločili za drugo metodo, ki temelji na sistemu *nalepk*. Nalepke (angl. *decal*) lahko opišemo kot materiale, ki so projicirani na obstoječo površino oz. geometrijo v sceni. Pomemben del



Slika 3.6: Primer različnih primerov barvanja na neenakomerni površini

za to funkcijo je torej material, ki ga bomo uporabili za projekcije. Material ima dva parametra: *vektor*, ki ponazarja barvo v barvnem modelu RGB, in *masko*, ki določa obliko čopiča, s katerim rišemo po površinah.

Omenili smo, da želimo imeti na izbiro več barv in različne čopiče za risanje. Opisani material smo v ta namen uporabili kot bazni material, na podlagi katerega lahko dinamično ustvarjamo pojavitve materialov z različnimi parametri (barvo in masko). Prednost instanciranja materialov v primerjavi s samostojnim materialom je, da zahtevajo manj procesiranja, saj za pojavitve porabimo manj ciklov za izrisovanje materiala, kar je pri veliki količini porisanih površin še posebej pomembno za tekoče izvajanje programa. Bazni material je zastavljen tudi tako, da upošteva lastnosti materiala površine, na kateri stoji nalepka. Mednje spadajo normale (vektorji, ki določajo, kako se bo svetloba odbijala od površine), hrapavost oz. gladkost površine, upošteva pa se tudi sence, ki padajo na površino (slika 3.6).

Zaradi boljše preglednosti nad vsemi ključnimi informacijami in spremenljivkami, ki so pomembne za risanje, smo v logiko dodali novo podatkovno strukturo *struct FBrushOptions*. Ta hrani podatke o velikosti čopiča, seznam

vseh kazalcev na instance materialov, ki so na voljo, aktivni material, indeks barve in maske čopiča. Logika risanja je na strani akcijske komponente implementirana v samostojni funkciji, ob priklicu katere izvedemo črtno sled (angl. *line trace*) iz izhodišča *vektorske komponente* do neke vnaprej določene razdalje (v našem primeru 400 enot ali 4 metre) v smeri vektorja. Ob zadetku geometrije aktiviramo postopek dejanskega risanja na zadeti površini. Pri tem nam pomaga samostojen razred *Spline Actor*. V akcijski komponenti ob zadetku geometrije ustvarimo novo pojavitev tega razreda, ki v nadaljevanju izvaja risanje po površinah.

Gre za pomožni predmet v sceni, katere glavna komponenta je krivulja. Ob klicu risalne funkcije na krivulji dodamo novo točko, skozi katero potuje krivulja. Na mestu te točke ustvarimo novo nalepko. Pripnemo jo zadetemu predmetu tako, da bodo površine porisane tudi ob premikih geometrije. Nalepko še rotiramo na podlagi normale (vektor, pravokoten na površino) zadete točke, da je projekcija pravilo obrnjena glede na podlago.

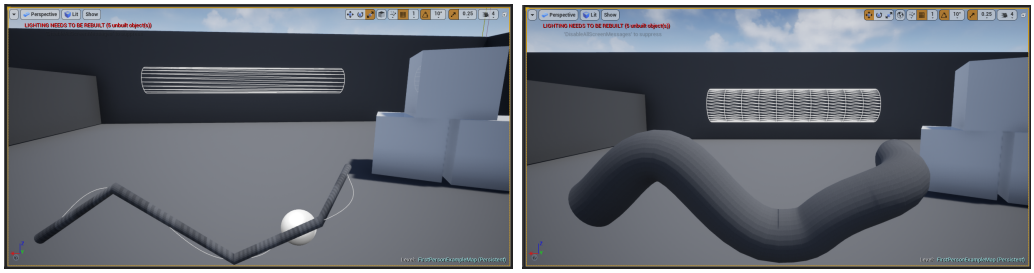
Z opisanim algoritmom bi že dobili funkcionalnost risanja, vendar naletimo na problem, če je razdalja med dvema točkama risanja prevelika. Hitrost poslikave površin je omejena s količino sličic na sekundo, saj lahko med vsako sličico risalno funkcijo kličemo največ enkrat. Omenjen problem smo rešili z algoritmom, ki pomaga zapolniti morebitno razdaljo med klici.

Položaj vseh vmesnih točk d_x izračunamo po formuli:

$$d_x = d_{n-1} + i * \Delta s$$

Δs predstavlja največjo želeno razdaljo med posameznimi vmesnimi točkami, ki jih ustvarjamo, za i pa velja $\forall i : 0 <= i <= \Delta d/\Delta s$, kjer je Δd razdalja med zadnjo in predzadnjo točko, med katerima vstavljamo nove točke.

Veliko igralih pogonov v svojemu naboru funkcij že vključuje sistem za kreiranje *nalepk* na podlagi krivulj (angl. *spline decal*), kar pa v pogonu UE v času pisanja še ni bilo na voljo [23].



Slika 3.7: Primerjava krivulj z geometrijo različnega števila poligonov

3.3.4 Risanje v 3D prostoru

Risanje v prostoru je po delovanju precej podobno že opisanemu risanju po površinah, saj si funkciji delita več lastnosti. Namen te funkcije je bil uporabniku omogočiti risanje v tridimenzionalnem prostoru v sceni, na razpolago pa mu ponuditi več barv in različne materiale.

Logika je v *akcijski komponenti* podobna postopku, opisanem v prejšnjem razdelku, le da ne izvedemo črtne sledi, kjer iščemo zadeto geometrijo, temveč zgolj vzamemo točko na neki razdalji (v našem primeru 200 enot ali 2 metra) in lokacijo znova posredujemo ustvarjeni pojavitvi razreda *Spline Actor*. Ta ob vsakem klicu doda na krivuljo novo točko na prejeti lokaciji. Točke predstavljajo začetek in konec posameznih delov geometrije, ki jim določimo pojavitve na krivulji. Da se geometrija lepo prilega krivulji, so pomembne tudi tangente točk na krivulji. Za osnovo ustvarjene geometrije smo vzeli enostaven cilinder, lahko pa bi vzeli tudi kak drug model. Pomembno je, de se geometrija lepo sestavlja in zliva iz enega odseka v drugega, torej da sta začetek in konec modela enaka (če je to seveda lastnost, ki jo želimo doseči). Za gladko ukrivljanje geometrije moramo poskrbeti tudi, da ima model dovolj veliko število poligonov, saj se v nasprotnem primeru ne more lepo kriviti (slika 3.7).

Ob dinamičnem ustvarjanju geometrije na krivulji lahko pride do grobih stikov na točkah med dvema cilindroma. To smo izboljšali tako, da smo sproti popravljali tangente cilindrov ob dodajanju novih točk, da so zavoji

Slika 3.8: Primer treh različnih krivulj

bolj zglajeni. Uporabniku smo na voljo dali tri različne materiale v desetih različnih barvnih variantah. Pojavitve tudi v temu primeru ustvarjamo dinamično. Poleg tega lahko uporabnik spreminja tudi velikost geometrije. Na narisane krivulje pa deluje tudi naša prva funkcija – risanje po površinah, kar doprinese še večjo možno interaktivnost (slika 3.8).

3.3.5 Upravljanje s predmeti v svetu

V ta sklop štejemo dve sorodni implementirani funkciji – *ustvarjanje predmetov* in *interakcijo s predmeti*. Prva uporabniku omogoča, da na poljubni lokaciji v sceni prikliče določen predmet. V akcijski komponenti se nahaja nabor vnaprej določenih predmetov, ki jih je mogoče priklicati v sceno. Iz vektorja na akcijski komponenti sprožimo snop, s katerim na določeni razdalji (v našem primeru 350 cm) preverjamo za morebitne trke. To pomaga tudi omejiti razdaljo ustvarjanja predmetov. V primeru trka nov predmet na mestu zadete točke, sicer pa ¡predmet prikličemo na maksimalni omenjeni razdalji, na koncu snopa.

Ustvarjeni predmeti so instance razreda *MeshActor*. Glavna komponenta teh predmetov je geometrija oziroma nek objekt. Bolj zanimiv del tega razreda pa je implementacija posebnega vmesnika (angl. *interface*), ki pomaga pri izpeljavi drugega dela funkcij.

Vmesniki so v sklopu pogona Unreal Engine uporabljeni za implementacijo sorodne logike preko na videz nepovezanih razredov. V našem primeru je to odzivanje različnih predmetov na interakcijo uporabnika. Uporabna lastnost vmesnikov je, da lahko definiramo privzeto logiko funkcij vmesnika. Razred, ki ta vmesnik implementira, bo samodejno privzel te definirane funkcije, lahko pa jih samostojno implementira in prepiše znotraj svojega razreda. Privzete logike funkcij nismo določili, saj smo jo v vseh razredih dodali posebej. Vse funkcije lahko razširimo tudi v sistemu *Blueprint* in tako nismo omejeni zgolj na implementacije v C++.

Vmesnik smo za naše potrebe uporabili v dveh različnih razredih. Prvi je že omenjeni razred *MeshActor*. Uporabniku z implementacijo vmesnika omogočimo, da lahko pojavitve tega razreda poljubno zgrabi in premika. Implementacijo izvajamo v dveh funkcijah vmesnika: *PickUp* in *Drop*.

Funkcije vmesnika proži uporabnikov lik. Akcijska komponenta ob dejanju poišče morebitni predmet v dosegu uporabnika. Ob zadetku predmeta preveri, ali zadeti akter implementira omenjeni vmesnik. V primeru, da ga, se kliče implementacija funkcije *PickUp*. Ob koncu dejanja se kliče še druga implementirana funkcija, *Drop*.

Po imenih lahko že sklepamo, katera opravlja kakšno funkcijo. Funkcija *PickUp* ob prijemu ustavi vse simulacije fizike predmeta in geometrijo pritrdi na uporabnikovo roko. To daje vtis, da je uporabnik predmet prijel v roke in ga lahko premika. Ob koncu dejanja, se izvede funkcija *Drop*, ki opravi nasprotje prej omenjene funkcije in predmet odklopi od uporabnika. Uporabnik ima na izbiro tudi, ali se bo ob spustu simulacija predmeta nadaljevala ali ne.

Prednost načina te implementacije, torej z vmesniki, je, da lahko poljubno dodajamo različne implementacije odziva na uporabnikovo interakcijo s pred-

meti. Še en primer, tokrat drugačne logike, so pojavitve vrat v sceni. Želeli smo, da se vrata ob uporabnikovem dotiku odprejo in zaprejo ob ponovitvi dejanja. Ker smo vrata ročno postavljali na ustrezne položaje znotraj scene, smo ta del izvedli v sistemu *Blueprint*. Za ta namen smo ustvarili nov razred, kjer je glavna komponenta geometrija vrat. Razredu smo dodali vmesnik in implementacijo funkcije *PickUp*. V tej se glede na to, ali so vrata odprta ali zaprta, izvede ustrezna animacija vrat. Funkcije *Drop* nismo potrebovali in je tudi nismo implementirali.

3.3.6 Uvažanje poljubnih modelov

Aplikaciji smo želeli dodati tudi vidik razširljivosti, s čimer smo poskusili povečati uporabnost in zanimivost aplikacije za uporabo. V ta namen smo integrirali tudi funkcijo uvažanja lastnih oziroma poljubnih 3D modelov.

Relativno preprost koncept pa je v zakulisju predstavljal bolj zapleten izziv. Pogon Unreal Engine je v splošnem namenjen izvajanju v realnem času in temu primerno običajno ne naletimo na potrebe uvažanja modelov ali drugih sredstev tekom izvajanja programa, zaradi česar pogon tudi nima vključenih funkcij za uvažanje 3D modelov v sceno. Industrija 3D modeliranja in vizualizacij ima v obtoku mnogo različnih formatov datotek. Nismo želeli omejiti podpore uvoženih modelov zgolj na peščico teh, zato smo uporabili knjižnico za branje in uvažanje najbolj pogostih in tudi redkeje uporabljenih formatov za 3D predmete in vse pripadajoče podatke. Knjižnica *Assimp (The Open-Asset-Importer-Lib)* [1, 2] je samostojna, odprtokodna knjižnica, napisana v C++. Seznam vseh podprtih formatov si je mogoče ogledati v njihovi uradni dokumentaciji, med njimi pa so seveda tudi najpogostejši formati, kot so: *.fbx, .obj, .dae, .collada, .gltf, .blend*.

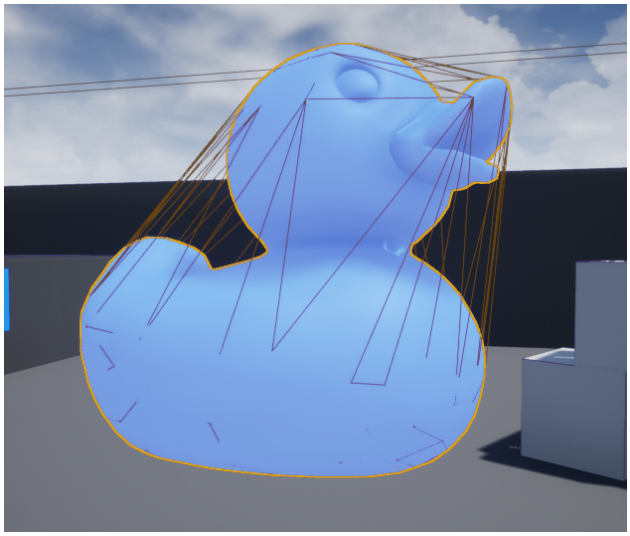
Knjižnico *Assimp* smo v naš program vključili v obliki predhodno prevedenih binarnih datotek, tako za 32- kot tudi za 64-bitne operacijske sisteme Windows, ki smo jih vključili v skriptu za gradnjo (angl. *build*) UE programa. Funkcije knjižnice smo izvajali v novi komponenti *AssimpComponent*. Komponento lahko dodamo kateremu koli predmetu v sceni in v njej neodvisno

izvajamo uvažanje predmetov. V komponenti hranimo vse glavne lastnosti 3D modela v obdelavi: *vozlišča, ploskve, tangente, normale, UV-je in barve vozlišč*. To so osnovni podatki 3D geometrije, s katerimi lahko poustvarimo predmet znotraj scene. Iz knjižnice Assimp uporabimo pravzaprav le logiko za uvažanje (razred *Importer*), ki podatke uvoženega predmeta normalizira in poenoti. Geometrija, ki jo uvozimo, je razdeljena na *sekcije*. To so posamezna vozliča v hierarhiji predmeta. Te sekcije rekurzivno obdelujemo in podatke shranimo znotraj komponente.

Čeprav s pomočjo knjižnice model uvozimo v naš program, ga še vedno nismo nikjer prikazali v sceni, saj UE za ta model še ne ve. Za postopkovno ustvarjanje geometrije v sceni med izvajanjem programa pogon UE ponuja temu namenjen razred *ProceduralMesh*. Uvoz datotek smo izvedli tako, da komponenti podamo pot do mape, kjer se nahajajo vsi modeli, ki jih želimo uvoziti. Preverimo vse datoteke znotraj mape, ali so v podprtem formatu, in vsak podprt predmet posebej uvozimo. Vsakemu modelu ustvarimo novo instanco proceduralnega razreda in ustvarimo posamezne sekcije uvoženega modela. Za to sta potrebni vsaj tabeli *vozlišč* in *ploskev*, vsi ostali podatki so sekundarnega pomena. Simuliranje fizike na naših pojavitvah uvoženih modelov potrebuje tudi telo za trke. Tu uporabimo kar podatke o vozliščih na modelu, kar da precej točno geometrijsko telo za trke (slika 3.9). Vse uvožene modele uporabimo pri funkcionalnosti ustvarjanja predmetov iz nazadnje omenjene implementirane funkcije.

Pred ustvarjanjem geometrije skušamo uvoziti tudi materiale. Knjižnica Assimp skupaj z geometrijo uvozi tudi podatkovno strukturo, ki vsebuje informacije o materialih modela. S pomočjo teh skušamo rekonstruirati material, ki je primeren za uporabo v pogonu UE. Več o postopku uvažanja materialov si bomo pogledali v poglavju o materialih (3.5.3).

Postopek uvažanja smo realizirali v pomožni komponenti. Kot argument potrebuje pot do modelov in vrne seznam uvoženih predmetov. Na ta način ne podvajamo logike uvoza za oba načina uporabe (z in brez naprave za VR). Pri ustvarjanju pojavitev uvoženih modelov nam je pomagal odprto-



Slika 3.9: Primer uvoženega modela skupaj z njegovim telesom za trke

kodni vtičnik za postopkovno ustvarjanje geometrije *Runtime Mesh Component* [21]. Uporabljeni vtičnik je po delovanju zelo podoben implementaciji v pogonu UE, uporabili pa smo ga zaradi boljše zmogljivosti pri delovanju, saj naj bi bil hitrejši in v povprečju porabil le 1/3 spomina v primerjavi s sorodno implementacijo v pogonu UE.

3.4 Uporabniški vmesnik

Vse elemente uporabniškega vmesnika smo zgradili v sistemu *UMG (Unreal Motion Graphics)*. Zaradi možnosti izbire izkušnje v navadnem načinu in načinu VR smo se odločili tudi za dve različni izvedbi uporabniška vmesnika. Tako smo izbrani vmesnik lažje prilagodili izkušnji in razlikam v komunikaciji uporabnika s programom. Implementaciji uporabniškega vmesnika sta sicer neodvisni ena od druge, imata pa kljub temu enake sestavne dele, saj uporabniku sporočata enake podatke.

Uporabniški vmesnik v obeh izvedbah lahko ločimo na dva dela. Prvi del uporabniku vizualizira izbrani način, drugi pa mu sporoča vse relevantne

nastavitve, ki se nanašajo na izbrani način delovanja. Sočasno so uporabniku vidne nastavitve za največ en način. Tako uporabnika ne preplavimo z informacijami.

Ko uporabnik ne uporablja naprave za VR, svoje namene programu sporoča preko tipkovnice in miške. Način uporabe akcijske komponente lahko spreminja s tipkami od 1 do 4.

Na sliki 3.10 je predstavljen primer uporabniškega vmesnika za prikaz aktivnega načina uporabe. Vsak način predstavlja ikona, ki nakazuje na to, kakšno funkcijo izvajamo v ponujenih načinih uporabe. Ikone spremljajo tudi številke, ki uporabniku sporočajo, s katero tipko izbere kateri način uporabe:

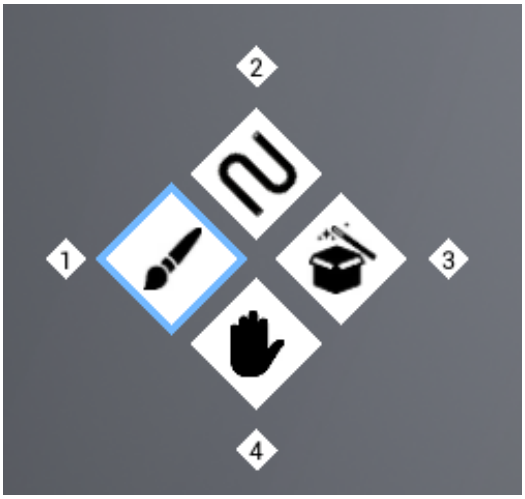
- 1 – barvanje po površinah;
- 2 – risanje v 3D prostoru;
- 3 – ustvarjanje modelov v sceni;
- 4 – interakcija s priklicanimi predmeti.

Na izbrani način opozarja tudi modra obroba ikone, da lahko uporabnik v vsakem trenutku ve, kateri način je aktiviran.

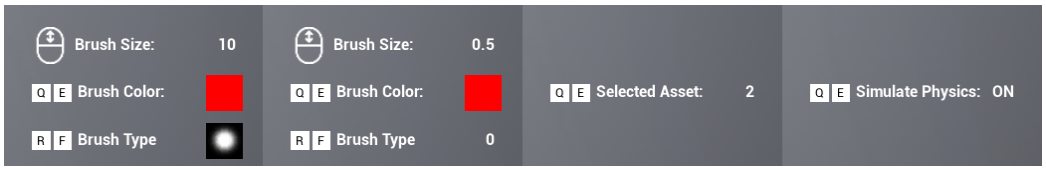
Drugi del uporabniškega vmesnika služi prikazu vseh aktivnih nastavitev izbranega načina uporabe. To zajema vse nastavitve, s katerimi uporabnik lahko upravlja. Prikaz informacij se prilagaja glede na aktivni način. Slika 3.11 prikazuje vse štiri izvedbe, enega za vsako funkcijo.

Uporabniški vmesnik svoje informacije posodablja v trenutku, ko uporabnik na akcijski komponenti spremeni kakšno nastavitev. Temu sta namenjeni dve funkciji, ena za posodobitev informacij pri spremembi načina uporabe in druga pri spremembi nastavitev funkcij.

Več načrtovanja je zahteval način VR, saj ima nadzorni plošček sistema HTC Vive le nekaj gumbov, zaradi česar smo morali najti prilagojeno rešitev. Na voljo imamo dva krmilnika, preko katerih uporabnik izvaja vsa dejanja. Poleg vseh funkcij moramo uporabniku omogočiti tudi gibanje po prostoru. Lahko bi vso logiko ločili na dva dela, na primer premikanje po prostoru na



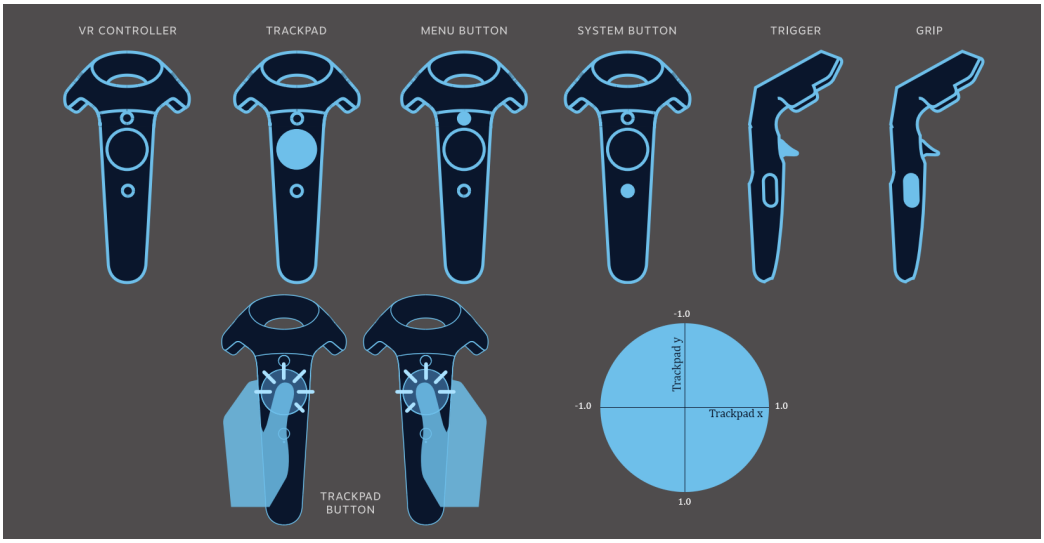
Slika 3.10: Uporabniški vmesnik za način uporabe



Slika 3.11: Uporabniški vmesnik za nastavitve

enemu krmilniku in vse implementirane funkcije na drugem. Vendar smo kljub temu želeli omogočiti vse funkcije na vsakemu ploščku in uporabniku prepustiti način uporabe. S tem dobimo torej na obeh krmilnih enotah enako delovanje, kjer vsaka implementira lastno akcijsko komponento. Iz tega razloga se lahko posvetimo delovanju in implementaciji na zgolj enem krmilniku.

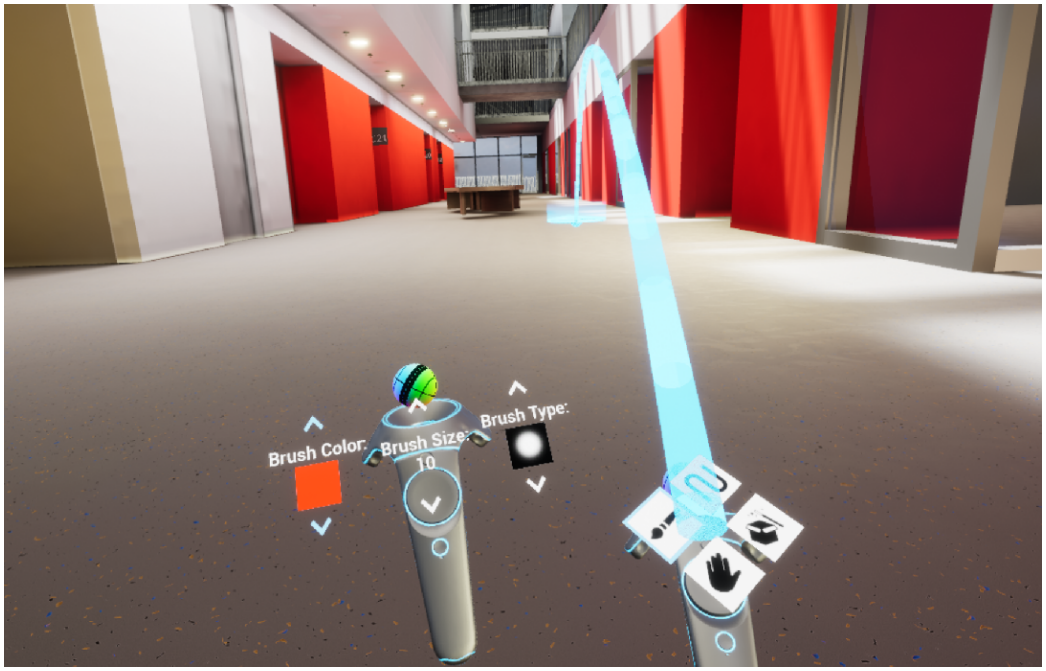
Na sliki 3.12 lahko vidimo shemo vseh vhodov na krmilniku HTC Vive, ki so na voljo uporabniku. Na krmilniku smo implementirali dve stanji, med katerima lahko uporabnik preklaplja. V prvemu, privzetem načinu se lahko premika po prostoru in spreminja načine uporabe akcijske komponente. Premikanje po sceni se sproži s pritiskom na sledilno ploščico (angl. *trackpad button* – glej sliko 3.12). Uporabniku se iz krmilnika prikaže sled in oznaka,



Slika 3.12: Shema vhodov na krmilniku HTC Vive

kam se bo premaknil ob spustu gumba. Med držanjem gumba lahko z drsenjem prsta po sledilni ploščici tudi nadzira orientacijo po premiku. Uporabnik lahko menja izbran način uporabe na akcijski komponenti s stranskim gumbom (angl. *grip button* – glej sliko 3.12).

Če želimo spreminjati nastavitve funkcij akcijske komponente, moramo preiti v sekundarni način krmilnika. Premikanje med načinoma krmilnika uporabnik izvaja z menijskim gumbom (angl. *menu button* – glej sliko 3.12). V sekundarnem načinu ploščka uporabniški vmesnik za izbrane funkcije zamenja vmesnik za nastavitve, prilagojen za način VR (glej sliko 3.13). Funkcija premikanja po prostoru se onemogoči, saj sledilno ploščico v tem primeru uporabimo za navigacijo po nastavitvah. Drsna ploščica zaznava dotike in jim določi vrednosti na oseh x in y, na intervalu [–1,0, 1,0]. V premeru, da se gumb sproži na intervalu [–1,0, –0,5] ali [0,5, 1,0] na osi x, predpostavimo navigacijo med posameznimi nastavitvami. V obratnem primeru spreminjamo nastavitev glede na to, ali je vrednost y pozitivna ali negativna.



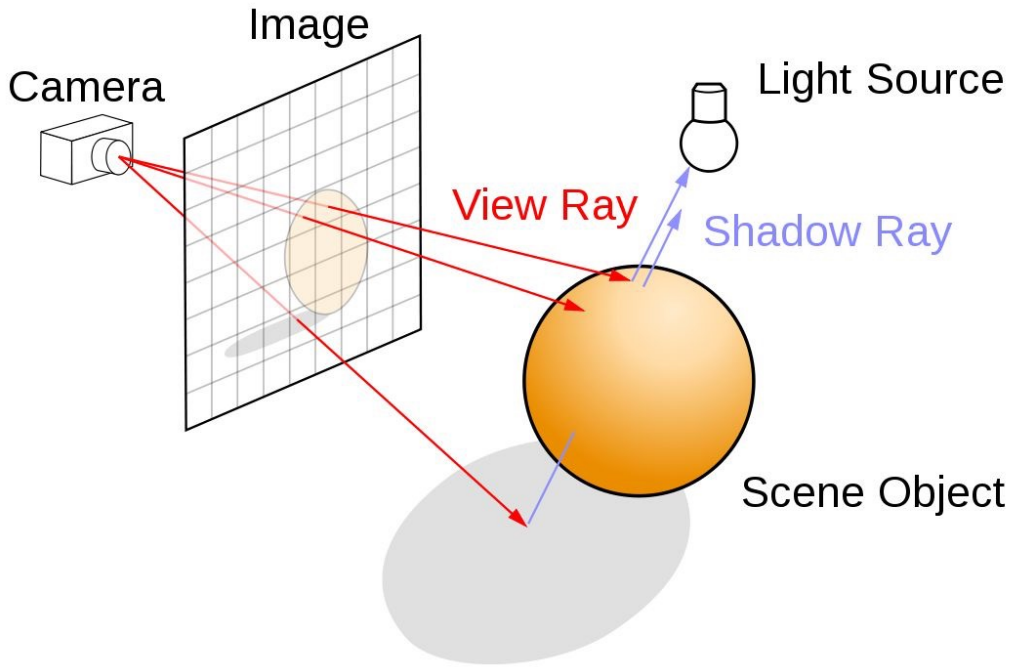
Slika 3.13: Uporabniški vmesnik za način VR

3.5 Okolje in osvetlitev

Osvetljevanje okolja igra pomembno vlogo pri realizaciji zadane scene, saj ustvarjena atmosfera in razpoloženje močno vplivata na uporabnikovo izkušnjo. S temačno in turobno atmosfero mu lahko vzbudimo strah in negativne občutke, lahko pa nanj pozitivno vplivamo s prijetnimi in toplimi lučmi. Cilj za našo sceno je bila realistična osvetlitev pri dnevni svetlobi.

V realnem svetu svetloba potuje iz nekega vira in se na površini odbije, razprši ali ukrivi. Tak model delovanja svetlobe, kjer računamo sledi snopov žarkov (angl. *ray tracing*), ustvarja izredno realistične in kakovostne slike, ki so blizu fotorealizmu, vendar pa zahteva ogromno računske zmogljivosti, če ga želimo posnemati v računalniški grafiki. Ena sama slika, ustvarjena na tak način, lahko zahteva več ur procesiranja, kar seveda ni ugodno za poganjanje scene v realnem času.

V času pisanja se na obzorju že pojavlja strojna oprema in nove teh-



Slika 3.14: Proces sledenja žarkov (vir: Nvidia [15])

nike upodabljanja, ki omogočajo upodabljanje sličic na principu sledenja žarkov [19, 18]. Kljub temu pa se za upodabljanje slik v realnem času običajno še vedno uporablja proces *rasterizacije*.

Rasterizacija je postopek transformacije teles (poligonov in točk) v 3D prostoru na dvodimenzionalno rastrsko sliko – množico slikovnih pik v 2D matriki. Namesto, da v sceni preračunavamo sledi množičnih žarkov, vzamemo posamezne predmete v vidnem polju in procesiramo informacije o geometriji. Vsako oglišče poligona na predmetu vsebuje več informacij, ki opisujejo posamezne dele geometrije. Med pomembnejšimi so: pozicija v prostoru, barve, normale, koordinate tekstur in tangente. Na podlagi teh informacij in podatkov o virih svetlobe v sceni nato s senčilniki (programi za senčenje) izračunamo barve, ki se bodo prenesle na posamezne točke v končni sliki.

Takemu procesu obdelave podatkov in upodabljanja pravimo *grafični ce-*

vovod. Posnemanje lomov ali odsevov svetlobe na tak način pa se izkaže za več kot le trivialno nalogo, saj si predmeti med seboj ne delijo informacij. Pri upodabljanju na principu sledenja žarkov to ni večji problem, saj žarki potujejo iz kamere proti svetlobi. Uporabljeni postopek rasterizacije se uporablja že od prejšnjega stoletja in predstavlja dobro ravnovesje med hitrostjo procesiranja in kakovostjo končne slike.

3.5.1 Sistem upodabljanja

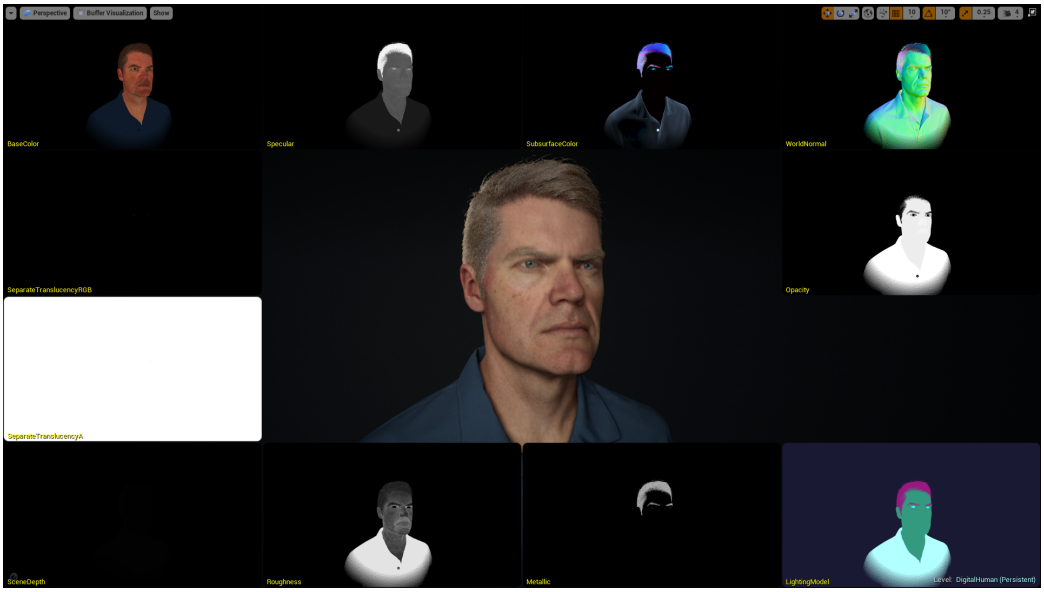
Pogon Unreal Engine za upodabljanje privzeto uporablja sistem *odloženega upodabljanja* (angl. *deferred rendering*). To je tehnika senčenja scene v vidnem polju. Svoje ime je dobil po tehniki senčenja, kjer pride do dejanskega senčenja z upoštevanjem svetlobe šele v drugem obhodu senčilnikov (angl. *shader pass*) pri senčenju oglišč in fragmentov (več o senčilnikih v poglavju 3.5.3).

Prvi obhod senčilnikov je namenjen zbiranju podatkov o geometriji v vidnem kadru. Med te podatke spadajo pozicije v svetu, normale in lastnosti materialov površin. Te informacije se shranijo kot teksture v namenski *geometrijski medpomnilnik* (angl. *geometry buffer – G-buffer*). Nato senčilniki v drugem obhodu na podlagi informacij o lučeh, ki vplivajo na predmete, izračunajo učinek direktne in posredne svetlobe v okolici. Na sliki 3.15 lahko vidimo vizualizacijo nekaterih tekstur, ki se hranijo v geometrijskem medpomnilniku.

Prednost odloženega upodabljanja je, da potrebuje za upodobitev sličice le eno procesiranje geometrij, pri temu pa upoštevamo le tiste luči, ki vplivajo na opazovano geometrijo. To omogoča računanje velikega števila luči v sceni.

3.5.2 Neposredno upodabljanje

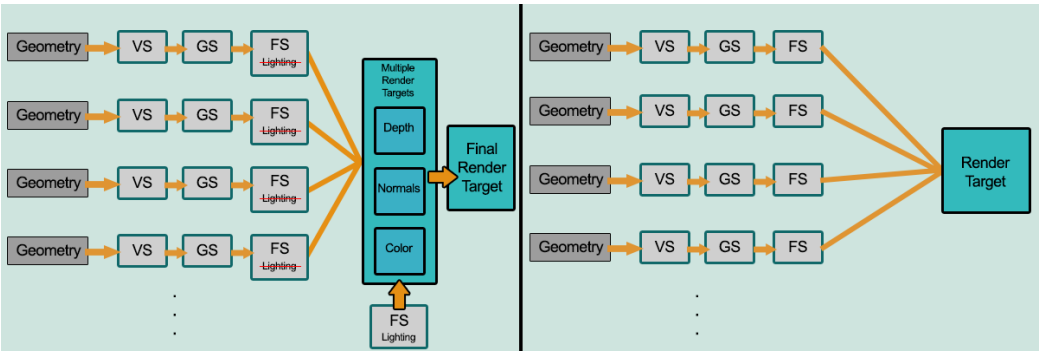
Odloženo upodabljanje pa ima tudi predhodnika – *neposredno upodabljanje* (angl. *forward rendering*). To je sicer že starejša metoda senčenja, ki pa je bila le nedavno tega vključena v pogon Unreal Engine 4 [33]. Omenjeno



Slika 3.15: Vizualizacija tekstur v geometrijskem medpomnilniku

tehniko senčenja je podjetje Epic Games uporabilo pri izdelavi svoje igre za navidezno resničnost *Robo Recall*.

Za razliko od odloženega senčenja se osvetlitev na geometriji pri tej metodi izračunava sproti za vsako geometrijo in za vsako luč, ki vpliva na predmet, posebej (glej shemo na sliki 3.16). Posledično v sceni ne moremo imeti veliko dinamičnih luči, katerih vpliv na osvetlitev bi se izračunaval sproti. V primeru, da v sceni nimamo veliko luči oziroma so statične, pa lahko pridobimo na hitrosti delovanja, kar je za navidezno resničnost še posebej pomembno. Podjetje Epic Games je zabeležilo tudi do 22 % hitrejše delovanje v omenjeni igri *Robo Recall* v primerjavi z odloženim senčenjem. Sistem neposrednega upodabljanja omogoča tudi kvalitetnejše glajenje robov (angl. *anti-aliasing*) na podlagi več vzorcev – *MSAA* (angl. *multisample anti-aliasing*). Opisani sistem senčenja pa seveda ni brez slabosti, saj ne podpira nekaterih funkcij pogona UE. Tako je na razvijalcih, ki se odločijo za neposredno upodabljanje, da razvijajo okoli teh pomanjkljivosti.



Slika 3.16: Primerjava odloženega (levo) in neposrednega senčenja (desno) (vir: Tutsplus [28])

3.5.3 Osvetljevanje v Unreal Engine

Osvetlitev okolice vključuje več gradnikov. V tem poglavju se bomo dotaknili vseh glavnih sistemov, ki smo jih uporabili za doseganje naših ciljev.

Luči

Glavni del osvetljevanja scene so luči oz. viri svetlobe. Pogon luči v sceni obravnava kot običajne predmete, ki imajo svoj prostor v sceni. Vsaka luč ima nabor nastavitev, ki so skupne vsem tipom luči. Najpomembnejše so:

- **intenziteta**: koliko energije oddaja vir svetlobe (v lumnih, kandelah ali enotah pogona UE);
- **barva**: barva svetlobe vira;
- **radij vpliva**: območje, v katerem luč deluje. V njem so vsi predmeti, na katere luč vpliva. Služi tudi za izračun padca intenzitete svetlobe glede na razdaljo od izhodišča.

V pogonu Unreal Engine je na voljo pet različnih tipov luči [12], ki so namenjeni različnim scenarijem osvetljevanja (slika 3.17):

- **točkovna luč**: lahko jo primerjamo z žarnico, ki svetlobo seva v vse smeri iz ene točke v prostoru;

Slika 3.17: Vrste luči in primeri uporabe

- **pravokotna površinska luč**: po delovanju je podobna točkovni luči, le da je vir svetlobe namesto ene točke pravokotna ploskev;
- **reflektorska luč**: luč, ki svetlobo oddaja v eno smer znotraj določenega stožca;
- **usmerjena luč**: simulira sončno svetlobo z vplivom čez celotno sceno;
- **nebna luč**: zajema oddaljene dele naše scene in jih projicira kot svetlobni vir. Za to lahko uporabimo bodisi dejansko sceno ali pa teksturo. Za razliko od ostalih tipov luči nebna luč ne oddaja fotonov, svetloba te luči pa se odbija le enkrat.

Vsak tip ima tudi več individualnih nastavitev. S temi lahko vplivamo na obnašanje luči in svetlobe, na sence, ki jih meče luč, na svetlobne funkcije in odsev luči. V podrobnosti vseh teh nastavitev se tukaj ne bomo spuščali posebej.

Zadnja pomembna lastnost luči in osvetljevanja, ki jo bomo omenili, je *mobilnost*. Večina predmetov v sceni ima možnost nastavitve mobilnosti

na eno od treh opcij. Četudi po imenu nastavitve morda ni intuitivno, ima mobilnost veliko vlogo pri osvetlitvi scene. Poglejmo si, kako vrste mobilnosti luči in osvetljenih predmetov vplivajo na senčenje scene.

Statična: Statično mobilnost uporabimo za vse predmete in luči, ki se med izvajanjem programa ne bodo spreminjali. Vse stacionarne luči bodo del predhodnega preračunavanja svetlobe. Rezultat je *statična osvetlitev*. Del tega računanja bodo vsi statični predmeti v sceni. Pridobljene informacije o osvetlitvi so shranjene ali zapečene v obliki posebnih tekstur. Statična osvetlitev je ugodna za hitrost delovanja, saj ob izvajanju programa dodatne osvetlitve stacionarnih luči ni potrebno na novo preračunavati, ponuja pa dobro kakovost osvetljave. Svetloba in sence, ki so zapečene v te teksture, se med izvajanjem programa ne morejo spreminjati, kar pomeni tudi, da le omejeno vplivajo na dinamične predmete v sceni. Ti ne mečejo senc na podlagi teh luči, še vedno pa so (približno) osvetljeni s pomočjo predpomnilnika neposredne svetlobe.

Stacionarna: Stacionarni so tisti predmeti, ki se ne gibljejo, jim pa kljub temu lahko med izvajanjem programa spreminjamo nekaj lastnosti. Stacionarnim lučem lahko nastavljamo intenziteto in barvo luči. Luči so še vedno del predhodnega izračunavanja svetlobe, lahko pa tudi osvetljujejo dinamične, premične predmete. Stacionarno osvetljavo lahko uvrstimo med dinamično in statično svetlobo. Spremembe lastnosti luči med izvajanjem programa vplivajo le na direktno osvetlitev. Vnaprej izračunane, posredne osvetlitve ni mogoče spreminjati.

Premična: Premični predmeti v sceni so popolnoma dinamični. Dinamična geometrija v sceni ne bo del predhodnega preračunavanja svetlobe. Lahko je osvetljena s strani statičnih luči na podlagi predpomnilnika svetlobe ali s strani dinamičnih luči. Dinamični predmeti bodo metali tudi dinamične sence (v realnem času), če jih osvetljuje stacionarna ali dinamična luč. Z vidika zmogljivosti je to veliko bolj potratna operacija kot dinamična osvetljava brez senc in je tudi do 20-krat

	Statična	**Stacionarna**	**Premična**
Kakovost	Srednja	Visoka	Visoka
Zahtevnost	Nizka	Srednja	Visoka
Dinamičnost	Nizka	Srednja	Visoka

Tabela 3.1: Lastnosti luči glede na mobilnost

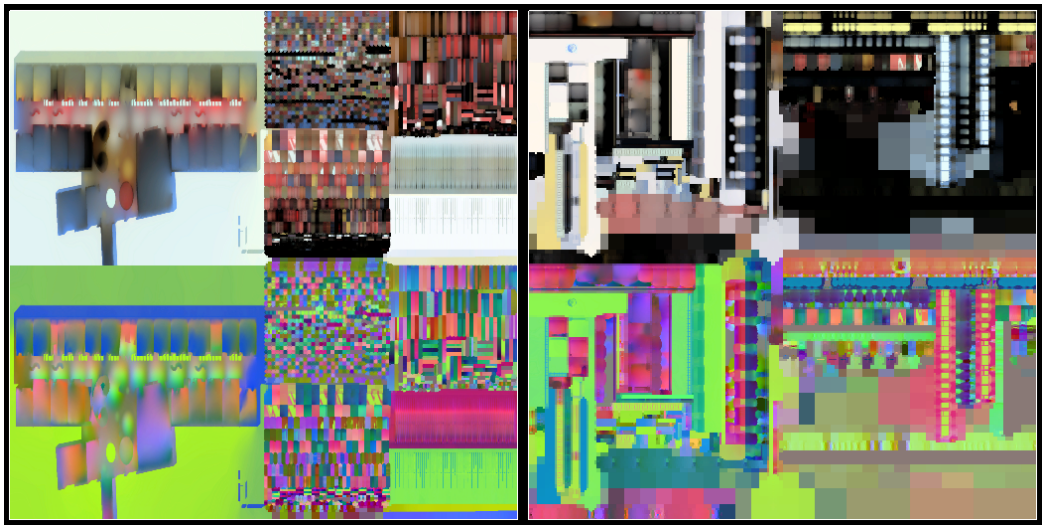
počasnejša [11]. Premične luči oddajajo povsem dinamično svetlobo in sence, kar pomeni, da se svetloba in sence računajo sproti, se ne odbijajo med površinami in jim lahko med izvajanjem programa spreminjamo praktično vse lastnosti.

Vse opisane vrste mobilnosti in načini delovanja imajo v sceni svojo vlogo in služijo različnim namenom. Na nas je, da različne tipe in nastavitve luči pametno izberemo glede na potrebe. V tabeli 3.1 je predstavljen povzetek povprečnih lastnosti za vsako vrsto mobilnosti [14].

V naši sceni je zelo malo premičnih in dinamičnih elementov. To so izključno predmeti, ki jih želimo premikati. Mednje štejejo predmeti razreda *MeshActor* in premični predmeti, ki implementirajo naš vmesnik *InteractionInterface*. Med te spadajo predmeti, s katerimi želimo upravljati – na primer vrata, ki jih odpiramo ali zapiramo, in predmeti, ki jih hočemo premikati po želji. Omenjeni akterji imajo torej *premično* mobilnost, vsa ostala geometrija pa je statična. Zaradi manjšega števila premičnih predmetov se bomo tudi osredotočili na statično, vnaprej preračunano svetlobo.

Lightmass

Sistem v pogonu Unreal Engine, ki skrbi za vso statično, vnaprej izračunano osvetlitev, se imenuje *Lightmass*. Za razliko od dinamičnih luči sistem Lightmass računa več odbojev svetlobe, s čimer dobimo bolj realne sence in osvetlitev. Ocenjevanju svetlobe na podlagi več odbojev od površin pravimo *globalna osvetlitev* (angl. *global illumination* – GI). Statična osvetlitev ima pomembno vlogo pri optimizaciji scen, ki tečejo v realnem času, saj je zahtev-



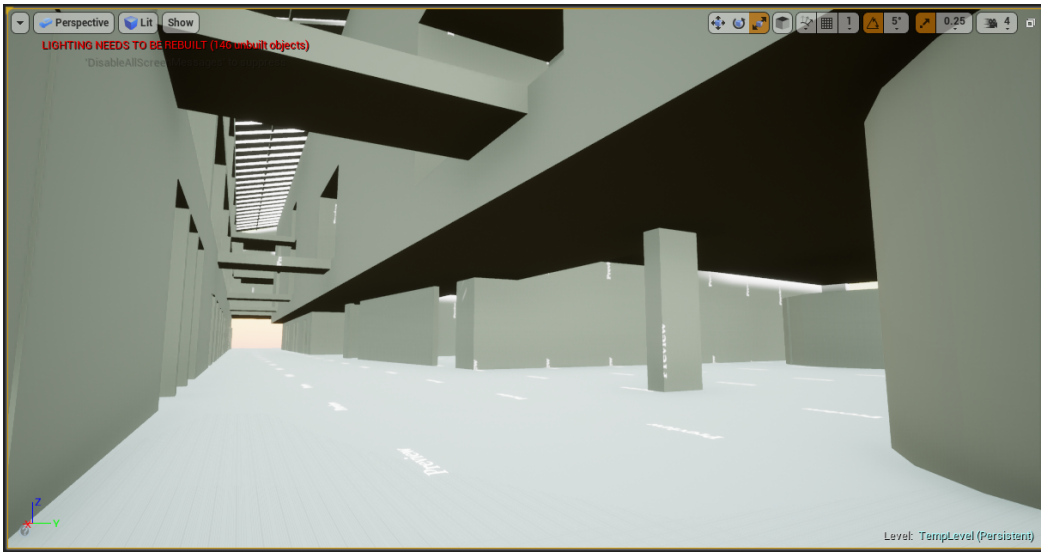
Slika 3.18: Primer dveh svetlobnih tekstur

nost prikazovanja skoraj ničelna. Glavni del tega sistema so posebne teksture (angl. *lightmaps*), ki vsebujejo vse glavne podatke o osvetlitvi in senčenju (slika 3.18). Če ovrednotimo sistem predhodnega računanja svetlobe:

- + dobra kakovost svetlobe;
- + nizka zahtevnost procesiranja ob izvajanju programa;
- – večja poraba pomnilnika zaradi večje količine tekstur v spominu;
- – počasnejše iteracije zaradi grajenja scene s svetlobo;
- – statičnost senc in osvetlitev.

Postavitev scene

Izdelavo scene smo začeli z izdelanim modelom fakultete v formatu *datasmith*. S tem smo uvozili tudi celotno hierarhijo modela iz programa Maya. V okolje dodamo še smerno in nebno luč za osnovno osvetlitev (slika 3.19).



Slika 3.19: Osnova scena z modelom fakultete

Za nadaljnje delo smo na Fakulteti za računalništvo in informatiko naredili nabor referenčnih slik, ki so pomagale poustvariti fakulteto. Pogoste informacije, ki smo jih črpali iz posnetkov, so bile:

- vrste materialov posameznih predmetov;
- videz predmetov, ki jih nismo pripravili v fazi modeliranja;
- tip in položaj luči.

Prvi korak je bil priprava glavnih materialov na površinah modela, kot na primer zidov in tal. Podrobnosti glede priprave teh bomo natančneje opisali v poglavju o materialih (3.5.3). V drugem koraku smo sceno opremili z dodatnimi predmeti. Tu smo nekatere modele dobili zastonj s spletnih strani za objavo 3D modelov (Turbosquid [27] ali Free3D [7]), nekatere pa smo izdelali sami. Za enostavne predmete smo geometrijo naredili kar v pogonu s pomočjo sistema *BSP* (angl. *binary space partitioning*). Gre za enostaven sistem, ki ga ponuja UE in kjer imamo na voljo nekaj osnovnih geometrijskih oblik, katerim določimo, ali prostor bodisi zapolnjujejo bodisi

Slika 3.20: Primer dobrih in slabih UV-jev

praznijo. Na ta način lahko hitro sestavimo enostavne predmete, namesto da bi jih izdelovali v programu za modeliranje in jih uvažali v UE. Slaba plat omenjenega sistem je, da včasih ustvari slabe UV-je, ki jih moramo popraviti v zunanjem programu. V nasprotnem primeru lahko pride do težav v sistemu Lightmass, ki računa svetlobo v sceni, kar lahko tudi v celoti pokvari zapečene svetlobne teksture (slika 3.20).

Stopnje podrobnosti

Modele lahko dodatno optimiziramo tako, da jih opremimo z različnimi stopnjami podrobnosti (angl. *level of detail* – LOD). Model menja stopnje podrobnosti glede na to, kako blizu jih opazujemo. Ko predmet gledamo od blizu, bo vidna najbolj podrobna verzija predmeta. Bolj ko se oddaljujemo od njega, manj podrobno bo predmet prikazan, saj na večjih razdaljah podrobnosti niti ne pridejo do izraza. Sistem LOD pomaga zmanjšati število poligonov, ki se v vidnem polju preračunavajo in izrisujejo. Za vsak geometrijski predmet imamo lahko več stopenj podrobnosti ali pa ne (le LOD0). Unreal Engine je opremljen s svojim sistemom za samodejno ustvarjanje sto-

Slika 3.21: Vizualizacija stopenj podrobnosti predmetov glede na oddaljenost od kamere

penj, kjer pogonu za posamezne predmete povemo, za kakšen predmet gre (majhno ali veliko geometrijo, rastje itd.). Za bolj kompleksne predmete pa lahko te stopnje izdelamo sami v poljubnem programu za modeliranje ali pa s programom za optimizacijo 3D modelov kot na primer *Simplygon* [22]. V naši sceni smo uporabili do štiri stopnje. Veliko predmetov, kot so na primer zidovi, pa so tudi brez dodatnih stopenj podrobnosti, saj so že v osnovi preprosti (slika 3.21).

Konfiguracija sistema Lightmass

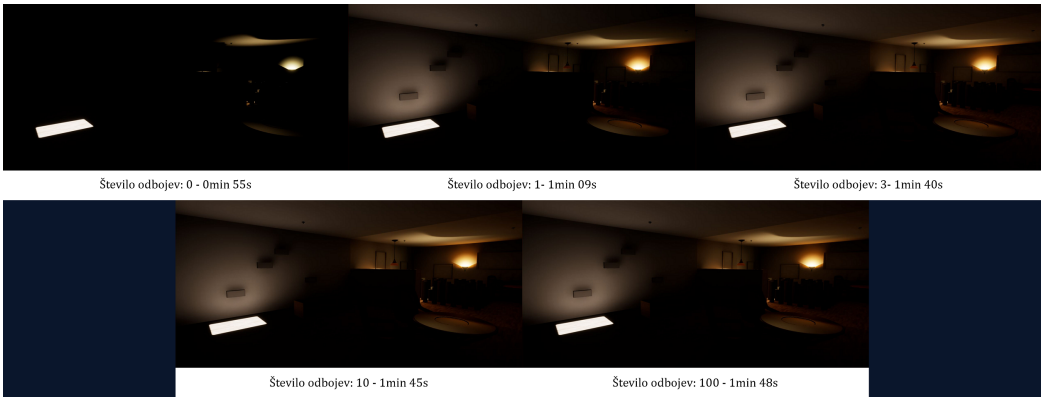
Ko so bili glavni predmeti na svojem mestu, smo se lotili nastavitev sistema *Ligthmass*. Pogon UE na voljo ponuja veliko možnosti nastavitev. Ugotoviti smo morali, katere nastavitve so prave za našo uporabo. Cilj je bil, da smo naredili sceno čim bolj realistično, zato bi lahko impulzivno izbrali vse najzahtevnejše opcije, a smo morali kljub temu razmisliti o nastavitvah in kaj prinašajo s sabo, saj zahtevnejše možnosti ne pomenijo nujno najboljših rezultatov. Poleg tega smo morali paziti tudi na učinkovitost delovanja, saj

kot že omenjeno, VR zahteva veliko zmogljivosti.

Najprej smo sistemu Lightmass povedali, kje je scena oziroma območje, kjer je pomembna kakovostna osvetlitev (*Lightmass importance volume*). Ponavadi je to območje, kamor uporabnik lahko dostopa. Temu območju smo dodali tudi območje za postprodukcijske učinke. Pri nastavitvah sistema Lightmass smo se osredotočili na naslednje nastavitve:

- **število posrednih odbojev svetlobe:** določa, kolikokrat se bodo fotoni iz virov svetlobe odbili med površinami – 0 odbojev pomeni, da ostanemo le z direktnimi osvetlitvami;
- **velikost scene za osvetlitev:** določa, v kakšnem razmerju velikosti (glede na mere sveta; ponavadi velja 1 UU = 1 cm) se bo računala svetloba – manjša je računana velikost, tem manjši bodo vzorci v sceni, za katere se bo računala svetloba. Vzorčimo torej več delov scene in tako dobimo več podrobnosti, a hkrati postanejo sence vse manj ostre;
- **kakovost posredne osvetlitve:** nastavitev, ki regulira faktor kakovosti odbite svetlobe – dobra praksa je, da je zmnožek *velikosti scene za osvetlitev* in *kakovosti osvetlitve* enak ali blizu 1;
- **glajenje posredne osvetlitve:** faktor glajenja odbite svetlobe – pomaga lahko zgladiti svetlobo in sence na površinah na račun podrobnosti senc.

Za testiranje nastavitev smo postavili preprosto sobo, ki jo osvetljuje več virov svetlobe. Preizkušali smo, kako omenjene nastavitve vplivajo na kakovost osvetlitve in senc ter zabeležili rezultate skupaj s sliko scene in časom, ki smo ga porabili za preračunavanje statične svetlobe. V prvem preizkusu smo ugotavljali, kako število odbojev svetlobe vpliva na videz okolja in čas grajenja scene. Ugotovili smo, da po številu odbojev, večjem od 10, v osvetlitvi ni opaznih vizualnih razlik. Čas računanja luči pa se prav tako občutno povečuje samo v začetnih ponovitvah (slika 3.22). Za naše potrebe smo ostali pri petih ali šestih odbojih.



Slika 3.22: Primerjava scene z različnimi števili odbojev in časom preračunavanja luči

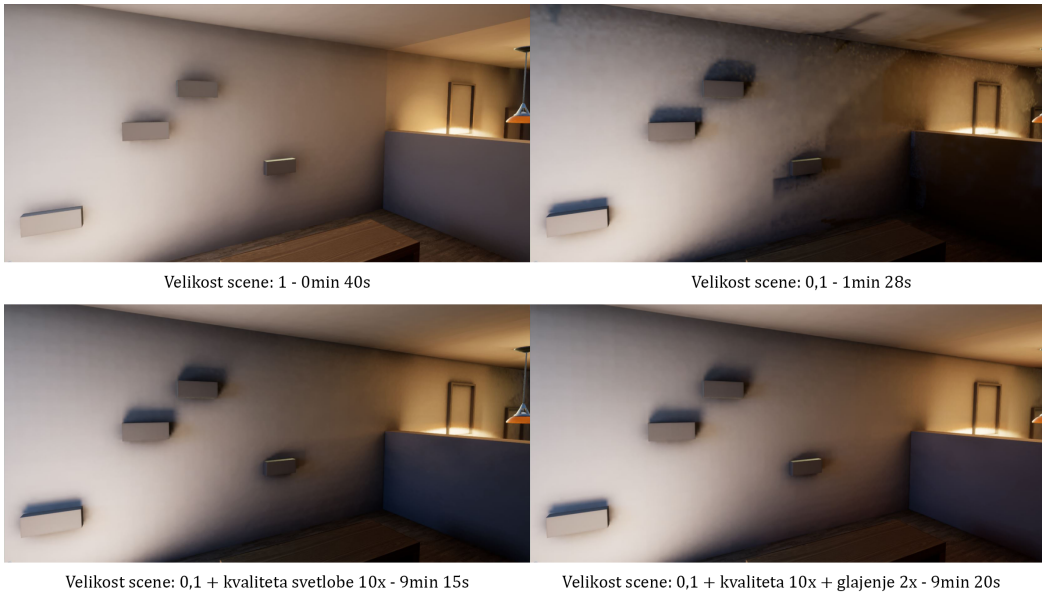
Preizkusili smo tudi, kakšno kakovost lahko dosežemo z manjšo velikostjo scene pri računanju svetlobe. Primerjali smo rezultate pri privzeti velikosti 1 in pri velikosti 0,1. Na sencah in osvetlitvi je bilo, kot pričakovano, veliko šuma. Dobili pa smo vseeno več podrobnosti na posameznih delih osvetljene geometrije, saj smo zajemali manjše dele predmetov. Šum v teksturah smo poskusili reševati s povečanim faktorjem kakovosti svetlobe. Kot omenjeno, je dobro oceniti razmerje velikosti scene in kakovosti tako, da je njun zmnožek blizu 1. Glede na to smo kot faktor kakovosti vzeli 10:

$$faktorVelikost \times faktorKakovosti = 0,1 \times 10 = 1$$

Videz scene se je s tem bistveno izboljšal, občutno pa se je povečal tudi čas preračunavanja. Ostalo je še nekaj negladkih prehodov v senčenju, kar smo odpravili z večjim faktorjem glajenja (slika 3.23). Za našo glavno sceno smo nastavili velikost okoli 0,5 in ustrezno prilagodili še kakovost in glajenje.

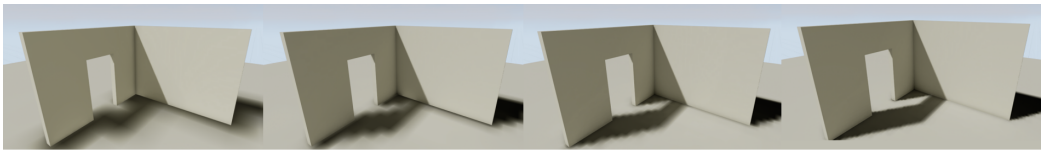
Svetlobne teksture

Ker se informacije o osvetlitvi zapisujejo v teksture, smo morali pozornost nameniti tudi ločljivosti teh tekstur na posameznih predmetih v sceni. Zagotoviti smo morali, da je ločljivost dovolj visoka, da so svetloba in sence

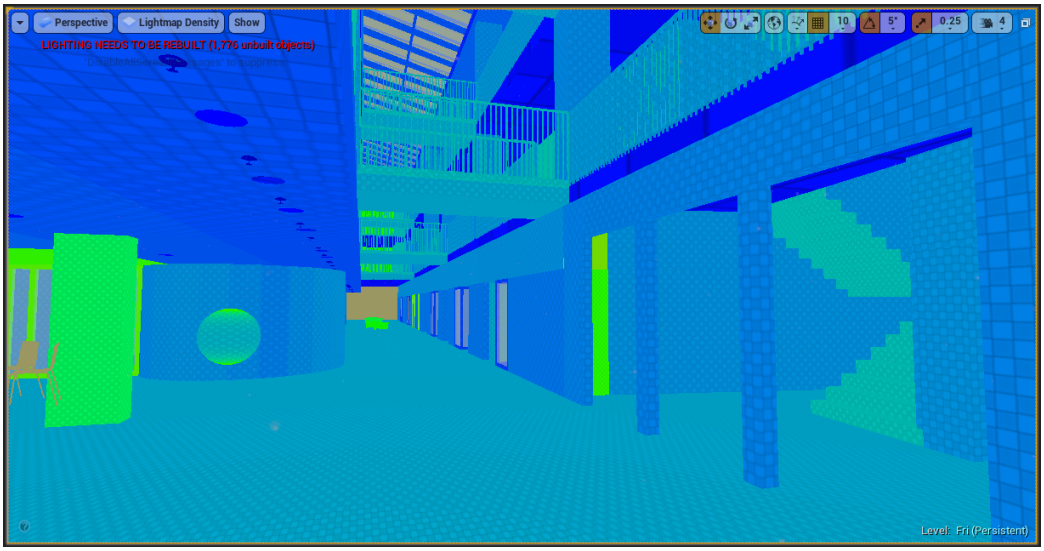


Slika 3.23: Primerjava scene z različnimi velikostmi scene in časi preračunavanja luči

dobro razločne in kakovostne. Pogon UE ob uvozu geometrije že sam oceni, kakšno ločljivost svetlobnih tekstur naj bi predmet imel, za izjeme, kjer te ocene niso bile ustrezne, pa smo jih določili ročno. Ločljivost je odvisna od velikosti delov predmeta in priprave UV-jev. Večji predmeti običajno zahtevajo višjo ločljivost svetlobnih tekstur, če želimo doseči enako kakovost kot pri manjših predmetih. Manj praznega prostora je med posameznimi deli v UV-prostoru, nižje ločljivosti teksture lahko določimo za enako kakovost. Prenizka ločljivost svetlobnih tekstur lahko povzroči nedefinirane sence (slika 3.24) ali pronicanje svetlobe med geometrijami. Poskusili smo ujeti tudi čim bolj enotne resolucije tekstur predmetov v sceni, da je statična svetloba čim bolj uniformna (glej sliko 3.25) in hkrati nismo pretiravali z velikostjo tekstur.



Slika 3.24: Primerjava različnih ločljivosti svetlobnih tekstur



Slika 3.25: Vizualizacije ločljivosti svetlobnih tekstur v sceni

Osvetlitev

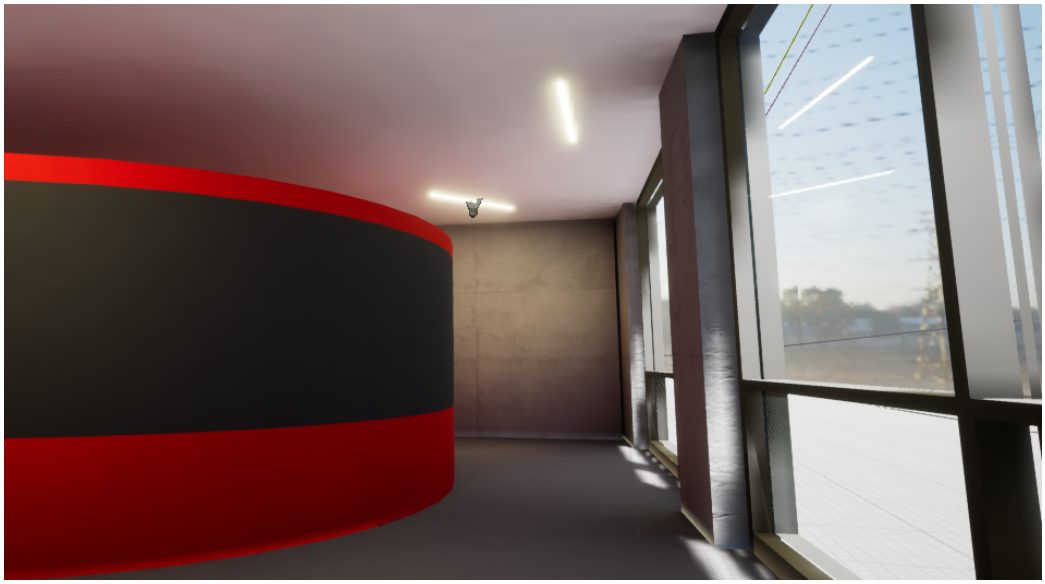
Kot že rečeno, smo se zaradi gladkega delovanja izogibali dinamični svetlobi. Zato smo za osvetlitev uporabili zgolj statične luči. Posledica tega je, da pri gibljivih predmetih izgubimo sence. Za vse luči znotraj modela fakultete smo uporabili točkovne luči in žaromete. Če smo se želeli približati referenčnim slikam fakultete, smo morali v sceno umestiti veliko število luči. Pogon ima omejitev treh statičnih luči, ki se med seboj lahko prekrivajo, zato smo morali osvetlitev prilagoditi tudi temu. To smo reševali na dva načina. Prva optimizacija je bila, da smo namesto večjega števila manjših luči blizu skupaj uporabili le eno večjo luč (slika 3.26).

Slika 3.26: Primer ene luči, ki nadomešča več manjših

Druga rešitev je bil material, ki oddaja svetlobo. Ta na predmetih daje vtis, da oddajajo lastno svetlobo (kot na primer luč), čeprav (privzeto) ne vplivajo na osvetlitev scene. Na ta način niti nismo potrebovali dejanskega vira svetlobe in smo zmanjšali število luči v sceni, ki ne vplivajo močno na sceno, ter skrajšali tudi čas računanja svetlobe (slika 3.27). Nekaterih luči pa enostavno nismo osvetlili in je namesto teh le model ugasnjene luči.

Portali

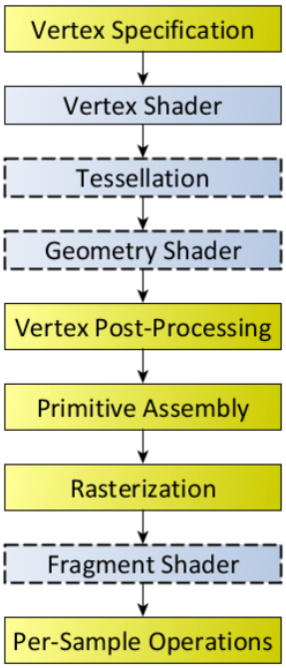
V prostorih, kjer svetloba prihaja v glavnini s strani nebne luči, lahko sistemu Lightmass povemo, kam naj usmeri pozornost pri osvetljevanju prostora z nebno lučjo. Ker ta vrsta luči ne oddaja fotonov, določimo od kod naj prihaja luč iz zunanjosti (npr. vrata, majhna okna). To smo storili s postavitvijo tako imenovanih *portalov* (angl. *Lightmass portals*) v sceni na omenjenih točkah.



Slika 3.27: Dve luči, od katerih le ena dejansko oddaja svetlobo v okolje

Iteracije osvetljevanja

Proti koncu izdelave smo imeli v sceni že lepo število predmetov (okoli 1700). Temu primerno je naraščal tudi čas preračunavanja svetlobe, kar smo morali storiti po vsakemu premiku, dodajanju, brisanju ali drugi spremembi geometrije, če smo želeli videti rezultate. Skozi izdelavo scene smo morali zato smiselno prilagajati nastavitve in optimizirati iteracije grajenja luči. Pogon za izračun luči ponuja štiri stopnje kakovosti. Vsaka višja stopnja vrne lepše in natančnejše rezultate, poviša pa se čas gradnje scene. Med izdelovanjem smo večinoma uporabljali dve najnižji stopnji kakovosti, da smo hitro prišli do informacij o tem, ali so geometrije dobro pripravljene in ali se luči obnašajo, kot pričakujemo. Po seriji večjega števila sprememb smo svetlobo gradili še na višjih stopnjah, kjer smo se osredotočili na podrobnosti. V zadnjih fazah izdelave smo povišali tudi opisane nastavitve sistema Lightmass, opazovali rezultate in po potrebi prilagodili nastavitve.



Slika 3.28: Grafični cevovod po specifikaciji OpenGL (vir: Khronos Group [10])

3.6 Materiali in senčilniki

Glavna vloga materialov je določiti, kako bodo videti površine. Materiali so abstrakcija *senčilnikov* (angl. *shader*), posebnih programov, ki se zaradi svoje paralelne narave običajno poganjajo na grafični kartici. Senčilniki se izvajajo v programabilnih stopnjah *grafičnega cevovoda*. Gre za proces, ki pretvori 3D sceno iz računalniškega zapisa v bitno sliko. Proces sestavlja več stopenj, od katerih so nekatere programabilne, kjer uporabnik lahko določi, kako se bodo informacije o geometriji preoblikovale (slika 3.28, stopnje v modrem).

S tem lahko dosežemo poljubno senčenje, videz predmeta v sceni in odziv svetlobe na površinah. Glede na stopnjo cevovoda obstaja več tipov senčilnikov. Najbolj osnovni tipi so:

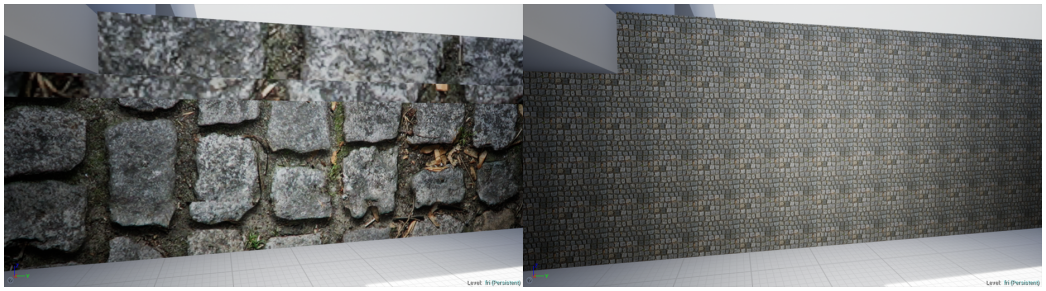
- **senčilnik oglišč:** transformacije na ogliščih in geometriji;

- **mozaičenje:** deljenje poligonov na več manjših enot;
- **senčilnik geometrije:** spreminja, dodaja ali briše primitive (točke, linije, poligone iz predhodnih faz cevovoda);
- **senčilnik fragmentov:** včasih imenovan tudi senčilnik slikovnih točk – izračunava informacije, ki se bodo prenesle na slikovne točke na končni sliki.

Material je lahko kombinacija več senčilnikov. Lahko ima tudi več ponovitev senčenja, kjer za en obhod šteje ena iteracija skozi omenjene senčilnike v grafičnem cevovodu.

V pogonu Unreal Engine senčenje privzeto poteka na podlagi sistema PBR (angl. *physically-based rendering*). To je model senčenja, ki si prizadeva čim bolj natančno poustvariti obnašanje svetlobe na površinah. Uveljavljen je v pogonih za vizualizacijo v realnem času, uporablja pa se tudi na drugih področjih. Model je na primer uporabil tudi studio Walt Disney v svojih animiranih filmih [4]. Sistem PBR lahko označimo bolj kot koncept [17], saj gre za nabor principov, ki so splošno uveljavljeni, obstaja pa več različnih izpeljav in implementacij. Gre za skupek matematičnih modelov, ki si zaslužijo svoje lastno delo, zato se tudi ne bomo globoko spuščali vanje. Glavne lastnosti materialov v sistemu PBR, ki jih najdemo tudi v pogonu UE, so: *barva, hrapavost, kovinskost, sijaj.* Na podlagi teh in nekaterih ostalih splošnih lastnosti smo gradili materiale za našo sceno.

Izdelava materialov ni bil glavni namen naloge in materiale navadno pripravljajo umetniki, ki imajo veliko več znanja na temu področju. Ker pa nam je bil videz naše scene vseeno pomemben in na to močno vplivajo tudi materiali, smo za bolj kompleksne materiale uporabili vtičnik *Substance* podjetja Allegorithmic [25]. Ta omogoča uporabo materialov, pripravljenih v njihovem sistemu *Substance Painter* oz. *Substance Designer*. Dostopali pa smo lahko tudi do nekaterih že pripravljenih materialov, ki so na voljo za Unreal Engine. Uporabili smo materiale, ki za samostojno pripravo niso preprosti, saj vsebujejo nabor tekstur za barvo, hrapavost, normale in sence. Glavna



Slika 3.29: Polaganje teksture na podlagi UV-jev (levo) in na podlagi položaja v sceni (desno)

prednost je, da lahko tem materialom spreminjamo parametre kar znotraj pogona UE in ne rabimo menjati med programi. Ob tem vtičnik sam ustvari vse potrebne teksture. Te materiale smo uporabili za lesene, betonske in nekatere kovinske površine.

Preostanek materialov smo pripravili sami. Med njimi je nekaj preprostejših materialov, kjer sta določena zgolj barva in hrapavost, pa tudi nekaj kompleksnejših. V nadaljevanju bomo razčlenili nekaj zanimivejših pristopov glede materialov v naši sceni.

3.6.1 Sestavljanje tekstur

Naš model sestavlja več posameznih kosov geometrije. Ti pa imajo na nekaterih delih tudi isti material. V primeru, da imamo v uporabljenem materialu teksturo, lahko nastane problem, saj se morajo teksture neopazno nadaljevati iz ene geometrije na drugo. To ujemanje pa je težavno loviti, saj se UV zemljevidi najverjetneje ne ujemajo popolnoma. Rezultat so teksture, ki so različno velike in se ne ujemajo iz enega predmeta na drugega (slika 3.29 levo). Problem smo rešili s polaganjem tekstur na podlagi položaja v sceni. Tako se bodo teksture po površini predmeta razporejale glede na koordinate x, y in z. S tem zagotovimo, da se bodo teksture, ki so pripete na različne predmete, lepo ujemale po robovih (slika 3.29 desno).

3.6.2 Steklo

V naši sceni pomembno vlogo igra tudi steklo, saj je v njej veliko steklenih površin. Iz tega razloga se nismo zadovoljili zgolj z enostavnim prosojnim materialom. Začnimo s *Fresnelom*, poznanim izrazom za učinek iz sveta 3D grafike, ki se imenuje po francoskemu fiziku Augustinu-Jeanu Fresnelu. Učinek opisuje razmerje med kotom, iz katerega gledamo predmet, in količino svetlobe, ki se odbija od površine. Za primer lahko vzamemo kar steklo. Večji je kot, pod katerim gledamo na stekleno površino, tem več svetlobe se bo odbijalo od nje. Rezultat Fresnelovega učinka je podlaga za ostale učinke. Steklu smo dodali tudi dve barvi, ki ju na podlagi rezultata Fresnela linearno interpoliramo. Za prosojnost stekla prav tako uporabimo Fresnelov učinek, s katerim lahko reguliramo, kako prosojni naj bodo robovi stekla. Steklu smo dodali tudi šum, ki naredi material moten. Za to smo uporabili primer Perlinovega šuma [39]. Ko svetloba potuje skozi steklo, pride tudi do njenega lomljenja. Določimo bazni skalar za lom svetlobe z vrednostjo 1,52 za steklo [20] in ga množimo s teksturo šuma. Na lom svetlobe bo tako vplival tudi šum stekla. Material smo pripravili z instanciranjem v mislih, kar pomeni, da lahko poljubno spreminjamo parametre na posamezni pojavitvi materiala. To omogoča, da v sceni uporabljamo več instanc istega materiala z različnimi parametri, odvisno od želja in potreb (slika 3.30).

3.6.3 Emisivni material

Materiale, ki oddajajo svojo svetlobo, smo že omenili v poglavju o osvetljevanju (3.5.3), lahko pa jih še malo bolj razčlenimo. Enostavnih emisivnih materialov običajno drugi viri svetlobe ne osvetljujejo. S tem prihranimo na zahtevnosti materiala. Material ima dva parametra – barvo in intenziteto. Po potrebi smo naredili več pojavitev materiala in jim neodvisno spremenili parametra. Uporabno je tudi, da lahko te materiale uporabimo kot dejansko statično luč (slika 3.31). V naših scenah tega sicer nismo uporabili in smo osvetljevali zgolj z lučmi.

Slika 3.30: Tri različne instance materiala stekla

Svetilni oz. emisivni materiali so nezahteven način za osvetlitev prostora brez dodajanja novih luči.

3.6.4 Zahtevnost materialov

Pri izdelavi materialov lahko prehitro vso pozornost usmerimo na estetski del in pozabimo na računsko zahtevnost senčilnikov. Če si ogledamo vizualizacijo zahtevnosti materialov v naši sceni na sliki 3.32, vidimo, da močno izstopajo steklene površine. Prosojni materiali imajo večjo zahtevnost izračunavanja, saj se upodabljajo tudi površine in materiali za prosojno površino. Posebno zahtevni so večkratni nizi prosojnih površin. Naš material za steklo je z dodanimi lomi svetlobe in šumom površine najzahtevnejši senčilnik v sceni. Vsi ostali so občutno manj zahtevni. Če pogledamo še število operacij na nekaterih materialih, lahko vidimo razlike v zahtevnosti (glej tabelo 3.2).

Slika 3.31: Emisivni material, uporabljen za osvetlitev

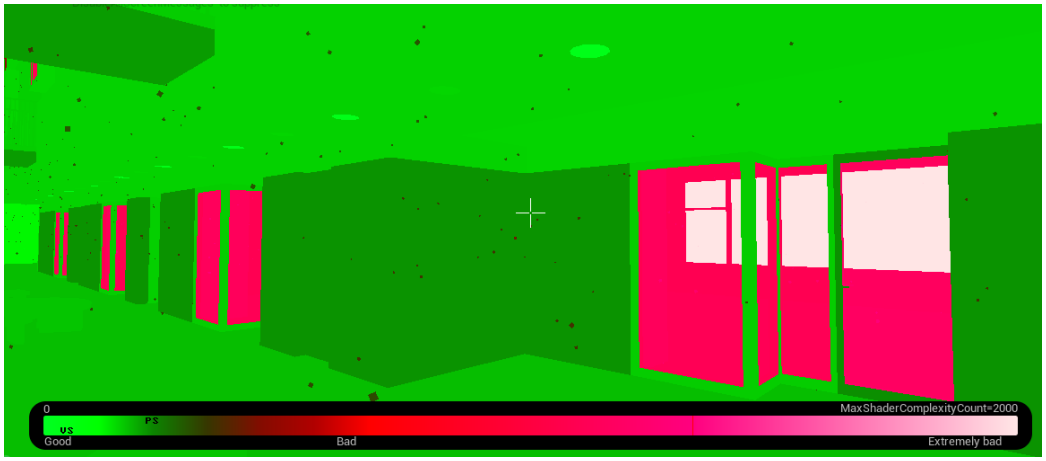
	Material zidov	**Emisivni material**	**Steklo**
Število operacij	101	30	871

Tabela 3.2: Število operacij treh različnih senčilnikov

3.6.5 Uvažanje materialov

Med podatki uvoženega modela so tudi informacije o materialih na predmetu. Vendar pa ti niso v obliki, ki bi jo lahko neposredno prenesli na materiale v UE. Razlog za to je, da so materiali lahko pripravljeni v različnih programih in na osnovi različnih modelov senčenja, ki niso nujno PBR. Za pretvorbo iz enega modela senčenja v drugega ponavadi ni splošnega algoritma. Tisti, ki se tega lotijo, pa običajno rezultate prilagodijo za svoje potrebe in se zadovoljijo z določenimi približki.

Naš cilj je bil uporabiti najpogostejše lastnosti materialov. Za namen uvažanja smo zastavili bazni material v pogonu UE. Pri uvažanju ustvarimo pojavitve tega baznega materiala s podatki o uvoženemu materialu. Vse vrednosti baznega materiala smo zastavili kot nastavljive parametre. Pripravili smo jih tako, da ima material nevtralne vrednosti, tudi ko nimamo informacij

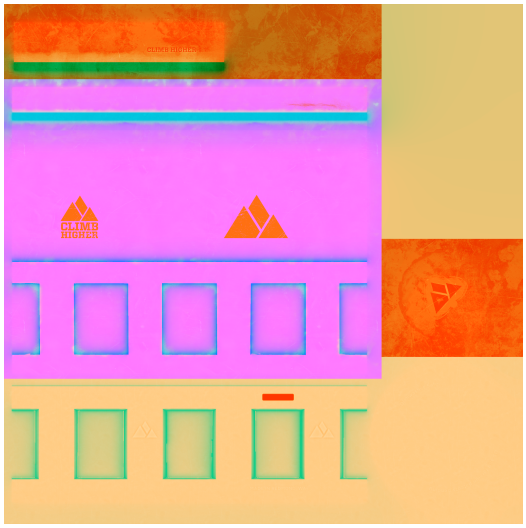


Slika 3.32: Zahtevnost materialov v delu scene

o vseh vrednostih in jih ne moremo nastaviti.

Najprej zberemo informacije o barvah materiala. Uporabili smo bazno in emisivno barvo. V primeru, da barvi nista definirani, smo nastavili privzete vrednosti bele barve za bazno in črne barve za emisivno. Zatem preverimo, ali so na voljo teksture. Naša implementacija podpira teksture za bazno barvo, normale, emisijo, senčenje, hrapavost in kovinskost. Pogosto se srečamo tudi s teksturami, ki vsebujejo različne informacije na različnih barvnih kanalih. To kombiniranje več tekstur lahko uporabimo, ko potrebujemo zgolj singularne vrednosti (od 0 do 1) za lastnost materiala. Take lastnosti so na primer hrapavost, kovinskost in senčenje. Te sivinske teksture so pogosto združene v eno, kjer vsako lastnost predstavlja en barvni kanal (slika 3.33).

Tudi te teksture smo upoštevali. Ob prisotnosti omenjenih tekstur se uporabi običajni kanal za vsako vrednost. Sistem bo deloval v primeru združenih in v primeru samostojnih tekstur. Sistem, ki smo ga implementirali, omogoča uvoz tekstur, ki so ločene od modela. To pomeni, da niso del binarnega zapisa predmeta, ki ga uvažamo, in jih lahko najdemo kot samostojne slikovne datoteke. Teksture, ki so del binarnega zapisa modela, so pogosto tudi kompresirane in bi jih morali še dekompresirati z implementacijo ustreznih algoritmov ali z uporabo temu namenjenih knjižnic (na premer DeviL, libjpeg,



Slika 3.33: Primer kombinirane teksture

libpng ali D3DX).

Ob uvozu geometrije shranimo tudi indekse materialov na posameznih delih modela, saj lahko več delov uporablja isti material. Uporabimo jih za pravilno naslavljanje materialov na površine.

3.7 Testiranje

Po implementaciji vseh delov izdelka je sledila faza testiranja. Osredotočili smo se na delovanje programa na sistemu HTC Vive. Če bo delovanje zadovoljivo na sistemu za VR, potem smo lahko precej prepričani, da bo tudi delovanje v klasičnem načinu tekoče. Sistem, na katerem smo testirali, ima naslednje glavne specifikacije:

- CPE: Intel Xeon E5-2630 (2,60 GHz)
- RAM: 32 GB
- GPE: Nvidia Geforce GTX 1080

Pri testiranju smo ciljali na najmanj 90 sličic na sekundo, kar je priporočena spodnja meja za navidezno resničnost. 90 slik na sekundo pomeni, da ima vsaka sličica na voljo 11,1 ms, da se preračuna in izriše. Uporabnik ima možnost ustvarjanja poljubnega števila poslikav in predmetov v sceni, ki je omejeno praktično le s hitrostjo izrisovanja. Zato smo si prizadevali sceno čim bolj optimizirati. To je po drugi strani na žalost pomenilo onemogočanje nekaterih vizualnih učinkov, zaradi katerih je bila scena videti bolje.

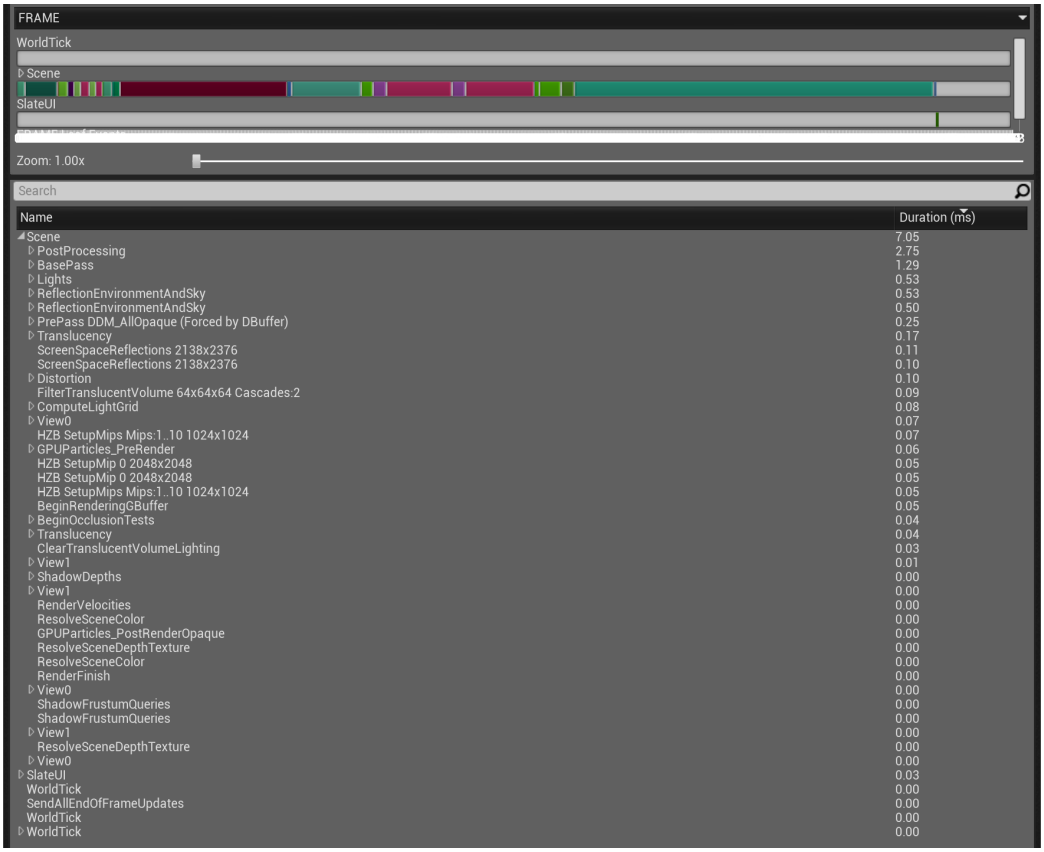
Že takoj smo ugotovili, da dinamična osvetlitev in sence gibljivih predmetov ne pridejo v poštev, saj imajo visoko zahtevnost izračunljivosti, ki je za delovanje VR nesprejemljiva. Ena slika je po optimizaciji luči potrebovala okoli 10 ms, od tega 7 ms računanja na grafični kartici (slika 3.34). Čas je sicer pod našo mejo, vendar v primeru ustvarjanja predmetov v sceni številka hitro naraste. Zavoljo optimizacije smo naredili še nekaj korakov.

Odločili smo se, da uporabimo sistem *neposrednega upodabljanja*, kar je tudi priporočilo Oculusa [6]. Ta način senčenja ima nižjo bazno zahtevnost izračunavanja slikovnih točk, kar postane v primeru odloženega upodabljanja težava že pri večjem številu slikovnih točk na končni sliki. Z uporabo neposrednega upodabljanja lahko izkoristimo tudi naprednejši način odstranjevanja robov MSAA. Rezultat tega so bolj ostre končne slike, kar je pri uporabi očal za VR bolj ugodno.

Iz slike 3.34 lahko vidimo tudi, da veliko časa pri procesiranju vzamejo postprodukcijski učinki. Izklopili smo vse učinke, ki smo jih opisali v poglavju o pripravi scene (3.2.2), nekatere pa smo zgolj malo prilagodili.

Požrešen proces pri delovanju na očalih za VR je tudi dvojno izrisovanje slik, enkrat na vsaki leči. Veliko podatkov iz slike na eni leči bi bilo možno reciklirati tudi za drugo. Pogon omogoča instanciranje določenih delov podatkov, kar ponekod pomeni, da se popolnoma rešimo računanja na drugi leči.

Po optimizacijah je bil čas izrisa ene slike 8 ms. Na sliki 3.35 lahko vidimo čas operacij na grafični kartici, vsota katerih je 5,65 ms. Opaziti je tudi manjše število operacij v primerjavi z meritvami pred optimizacijami.



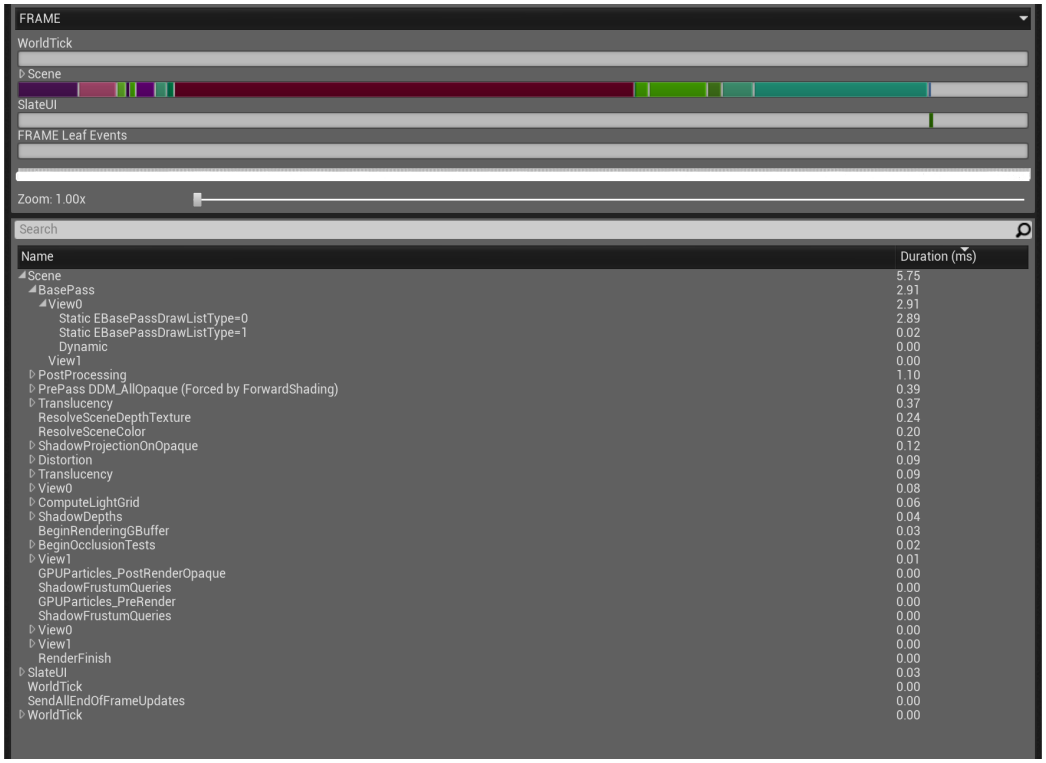
Slika 3.34: Časi računanja ene slike na grafični kartici

Videti je tudi čase računanja na vsaki leči (View0 in View1), kjer je ponekod na drugi leči tudi 0.

Z rezultati optimizacij smo bili zato zadovoljni, saj smo beležili 20 % boljše delovanje, čeprav na račun slabše vizualizacije.

3.8 Povzetek implementacije

Celotna implementacija dela je zajemala veliko različnih področij, zato povzetek dela na posameznih delih ni odveč. Naš cilj je bil postaviti Fakulteto za računalništvo in informatiko v virtualno okolje, kamor se bo možno podati v navidezni resničnosti. Bistvo je bilo ugotoviti, kaj vse stoji za procesom



Slika 3.35: Časi računanja ene slike na grafični kartici po optimizacijah

umeščanja take scene v navidezno resničnost, poleg tega pa okolje narediti čim bolj realistično in interaktivno.

Začeli smo z izdelavo modela fakultete. Modeliranje je potekalo v programu Maya in deloma v 3ds Max za izvoz geometrije. Pozornost smo usmerili na pravilno pripravljeno geometrijo (nizko število poligonov, UV zemljevidi) in na pravilno velikost predmetov (realna merila in razmerja). Preostanek implementacije je potekal v pogonu Unreal Engine. Zadane funkcije so bile: risanje po površinah, risanje v 3D prostoru in interakcija s predmeti v sceni.

Risanje po površinah temelji na sistemu nalepk, ki se ustvarjajo na risalnih površinah. To omogoča risanje po kateri koli površini v sceni. Uporabniku so na voljo različne barve, velikost in vzorci čopiča.

Risanje v prostoru je implementirano na podlagi krivulj, na katere lepimo

majhne koščke geometrije različnih velikosti, barv in materialov.

Interakcija s predmeti v sceni poteka skozi programski vmesnik, ki ga implementirajo vsi predmeti, s katerimi želimo upravljati. Uporabnik ima možnost v sceno postaviti poljubne predmete ter jih premikati in obračati.

Program omogoča tudi uvoz lastnih modelov. Za uvoz in normalizacijo različnih formatov smo si pomagali s knjižnico Assimp. Uvožene modele nato proceduralno zgradimo na podlagi informacij o geometriji.

Izdelali smo dve verziji uporabniškega vmesnika, enega za način z napravo za VR in enega za uporabo brez nje.

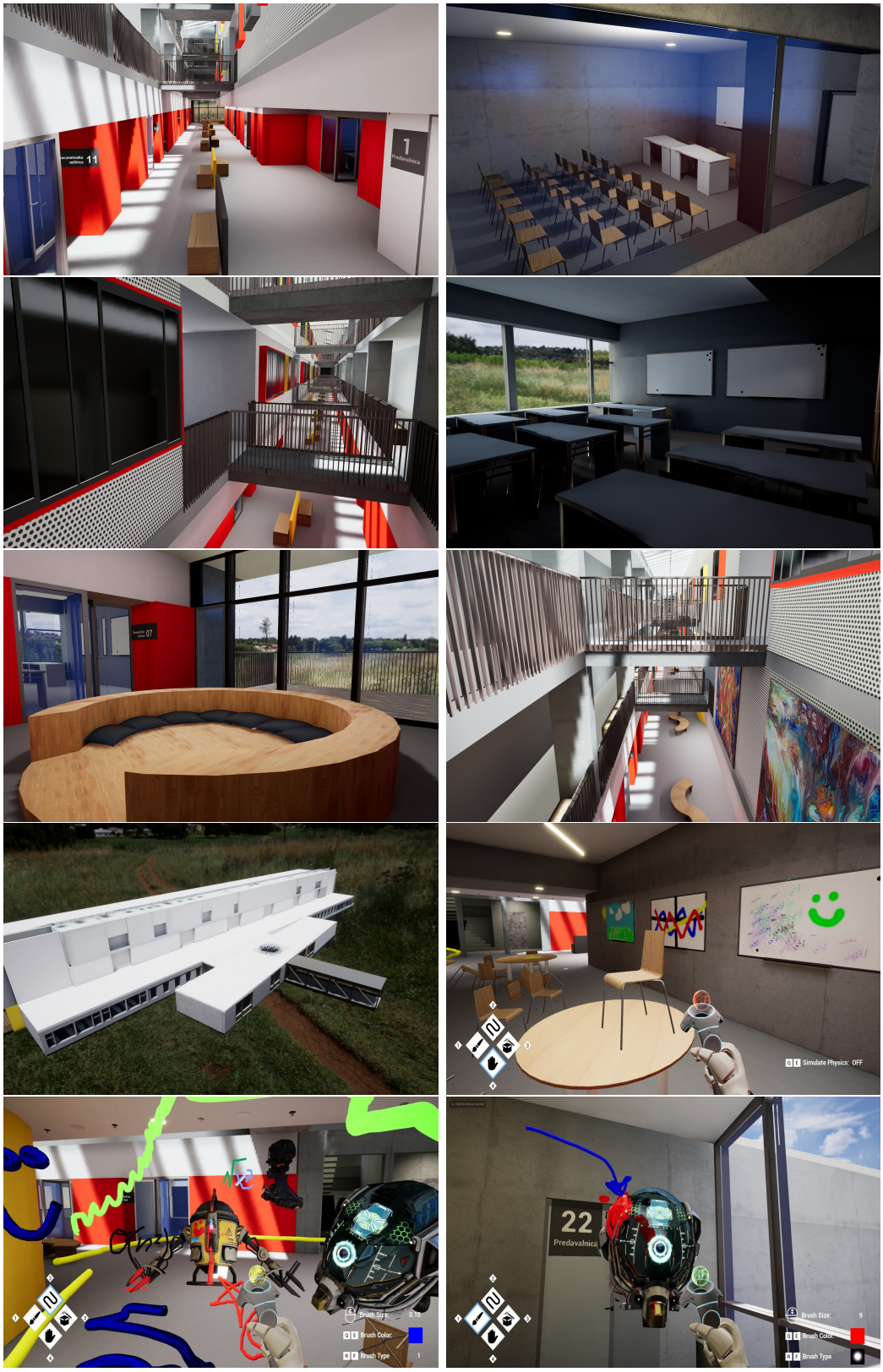
Sledil je še vizualni del razvoja. Sceno smo morali osvetliti in opremiti z lučmi, materiali in dodatnimi modeli. Glavnina osvetlitve je statična, kar pomeni, da je osvetlitev preračunana vnaprej. Posledica je, da med izvajanjem programa ni dinamične svetlobe in senc na gibljivih predmetih. Po drugi strani pa to omogoča nižjo zahtevnosti delovanja programa ter kakovostno osvetlitev in sence na statičnih predmetih. To je zlasti pomembno za navidezno resničnost, ki je zaradi uporabe v visokih ločljivosti na dveh lečah še posebej zahtevna. Zaradi velikega števila predmetov v sceni smo morali poskrbeti tudi za pravilno ujemanje plošč tekstur in ločljivost zapečenih svetlobnih tekstur na predmetih.

Trud smo vložili tudi v podrobnosti in funkcije, ki sicer niso bile osrednjega pomena za diplomsko delo, so pa kljub temu pripomogle k boljši izkušnji končnega uporabnika ali pa so olajšale sam razvoj. Za način brez naprave za VR smo pripravili animacije, kjer smo zastavili končni avtomat stanj in sistem prehajanja animacij. Programski del smo skušali poenostaviti z lastnimi podatkovnimi strukturami, ki so skrajšale količino potrebne logike, hkrati pa razjasnile naše namene v izvorni kodi. Pri postavljanju scene smo si pomagali s sistemom, ki je samodejno postavljal predmete v sceno na določeni ravnini. S tem smo prihranili čas, zmanjšali napake in ohranili urejenost scene. Testirali smo tudi bolj podrobne nastavitve sistemov in ocenjevali, kako vplivajo na delovanje in videz scene, kjer pa ni smiselno, da bi vse omenjali.

3.8.1 Možne izboljšave

Omenimo še, kaj bi bilo možno dodatno izboljšati. Veliko nadaljnjega dela bi lahko še vložili v model fakultete, ki bi s pomočjo grafičnih oblikovalcev potekalo še bolj učinkovito. Prostore bi lahko dodatno opremili z novimi modeli in materiali, dokončali pa bi lahko še ostala nadstropja in okolico. Pod drobnogled bi lahko vzeli tudi funkciji risanja po površinah in prostoru ter ugotovili, kako bi ju lahko optimizirali. Morda bi ju lahko implementirali na podlagi drugih sistemov in jima prilagodili način uporabe, kar bi potencialno pospešilo delovanje in izboljšalo izkušnjo v navidezne resničnosti. Uvažanje poljubnih modelov bi lahko izboljšali z dopolnitvijo sistema uvažanja materialov in podporo različnih modelov senčenja.

Prostora za izboljšave je še nekaj, kar pa ne pomeni, da z delom nismo zadovoljni. Realizirali smo namreč vse zadane cilje. Ustvarili smo realistično repliko fakultete in uporabniku ponudili interaktivno izkušnjo.



Slika 3.36: Deli končne scene in primeri implementiranih funkcij

Poglavje 4

Zaključek

Tehnologija navidezne resničnosti že omogoča, da dosežemo realistične in interaktivne izkušnje v virtualni okolici. To smo ugotovili tudi v tem diplomskem delu. Pri razvoju izkušnje za navidezno resničnost moramo biti pozorni na veliko vidikov – od optimizacij zaradi hitrosti delovanja do natančne priprave scene in predmetov. Če teh vidikov ne upoštevamo, izkušnja za uporabnike ne bo prijetna. Zaradi količine različnih dejavnikov, na katere moramo paziti, smo sklenili tudi, da je dobro ločiti razvoj izkušnje za VR naprave od izkušnje brez nje. Zahtevnost delovanja sistema VR, kot je HTC Vive, je od nas zahtevala veliko prilagajanja nastavitev in optimizacij. Posledica je bilo nazadovanje pri videzu in realističnosti scene. Kljub temu menimo, da smo dosegli cilj ustvariti realistično okolico, ki dobro posnema svojo resnično različico. V dobri VR izkušnji uporabnika ne omejujemo in mu ponudimo čim več interakcije z okolico, kar smo skušali ponuditi tudi uporabnikom tega izdelka.

Tehnologija VR ima še vedno veliko prostora za izboljšave. Z večanjem zmogljivosti računalniških sistemov bomo imeli na voljo vse več moči, ki bo omogočala izdelavo vse boljših in zahtevnejših virtualnih izkušenj. Na pohodu so novi sistemi z boljšimi lečami, sliko in brezžično tehnologijo, kar bo zagotovo pripomoglo k boljši izkušnji uporabnikov. Smo pa mnenja, da lahko že danes uporabimo obstoječo tehnologijo za izdelavo realističnih izkušenj za

namen sodelovanja z drugimi in vizualizacije različnih okolij in scenarijev.

Literatura

[1] Knjižnica Assimp. Dosegljivo: http://www.assimp.org/. [Dostopano: 19. 9. 2018].

[2] Assimp Github. Dosegljivo: https://github.com/assimp/assimp. [Dostopano: 19. 9. 2018].

[3] Autodesk. Dosegljivo: https://www.autodesk.com. [Dostopano: 25. 6. 2018].

[4] Walt Disney Animation Burley, Brent in Studios. Physically-based shading at disney. 2012.

[5] Epic Games. Dosegljivo: https://www.epicgames.com. [Dostopano: 25. 6. 2018].

[6] Unreal Forward Shading Renderer. Dosegljivo: https://developer.oculus.com/documentation/unreal/latest/concepts/unreal-forward-renderer/. [Dostopano: 11. 11. 2018].

[7] Free3D. Dosegljivo: https://free3d.com/. [Dostopano: 01. 10. 2018].

[8] Guinness World Records. Most successful videogame engine. Dosegljivo: http://www.guinnessworldrecords.com/world-records/most-successful-game-engine. [Dostopano: 25. 6. 2018].

[9] Htc. Dosegljivo: https://www.htc.com. [Dostopano: 25. 6. 2018].

[10] Khronos. Rendering pipeline overview. Dosegljivo: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview. [Dostopano: 03. 12. 2018].

[11] Light Mobility. Dosegljivo: https://docs.unrealengine.com/en-us/Engine/Rendering/LightingAndShadows/LightMobility. [Dostopano: 16. 10. 2018].

[12] Types of Lights. Dosegljivo: https://docs.unrealengine.com/en-us/Engine/Rendering/LightingAndShadows/LightTypes. [Dostopano: 14. 10. 2018].

[13] Autodesk Maya. Dosegljivo: https://www.autodesk.com/products/maya/. [Dostopano: 25. 6. 2018].

[14] M. A. Moniem. *Unreal Engine Lighting and Rendering Essentials*. Packt Publishing, 2015.

[15] Nvidia. Ray tracing. Dosegljivo: https://developer.nvidia.com/discover/ray-tracing. [Dostopano: 03. 12. 2018].

[16] Oculus to Cover Royalty Fees for Unreal Engine Developers. Dosegljivo: https://www.unrealengine.com/en-US/blog/oculus-to-cover-royalty-fees-for-unreal-engine-developers. [Dostopano: 25. 6. 2018].

[17] Physically-Based Rendering, And You Can Too! Dosegljivo: https://marmoset.co/posts/physically-based-rendering-and-you-can-too/. [Dostopano: 31. 10. 2018].

[18] Announcing Microsoft DirectX Raytracing! Dosegljivo: https://blogs.msdn.microsoft.com/directx/2018/03/19/announcing-microsoft-directx-raytracing/. [Dostopano: 10. 10. 2018].

[19] NVIDIA RTX Ray Tracing. Dosegljivo: https://developer.nvidia.com/rtx/raytracing. [Dostopano: 10. 10. 2018].

[20] Using Refraction. Dosegljivo: https://docs.unrealengine.com/en-us/Engine/Rendering/Materials/HowTo/Refraction. [Dostopano: 02. 11. 2018].

[21] Runtime Mesh Component Github. Dosegljivo: https://github.com/Koderz/RuntimeMeshComponent. [Dostopano: 24. 9. 2018].

[22] Simplygon. Dosegljivo: https://www.simplygon.com/. [Dostopano: 22. 10. 2018].

[23] Spline Decals? Dosegljivo: https://forums.unrealengine.com/unreal-engine/feedback-for-epic/108081-spline-decals. [Dostopano: 26. 8. 2018].

[24] Steam VR. Dosegljivo: https://store.steampowered.com/steamvr. [Dostopano: 25. 6. 2018].

[25] Substance. Dosegljivo: https://www.allegorithmic.com/. [Dostopano: 31. 10. 2018].

[26] Ivan E. Sutherland. The ultimate display. In *Proceedings of the IFIP Congress*, pages 506–508, 1965.

[27] Turbosquid. Dosegljivo: https://turbosquid.com/. [Dostopano: 01. 10. 2018].

[28] Tutsplus. Forward rendering vs. deferred rendering. Dosegljivo: https://gamedevelopment.tutsplus.com/articles/forward-rendering-vs-deferred-rendering--gamedev-12342. [Dostopano: 03. 12. 2018].

[29] Animated Children's Series ZAFARI Springs to Life with Unreal Engine. Dosegljivo: https://www.unrealengine.com/en-US/enterprise/blog/animated-children-s-series-zafari-springs-to-life-with-unreal-engine. [Dostopano: 25. 6. 2018].

[30] Datasmith and Unreal Engine Drive Real-time Architectural Design at Neoscape. Dosegljivo: https://www.unrealengine.com/en-US/enterprise/blog/datasmith-and-unreal-engine-drive-real-time-architectural-design-at-neoscape. [Dostopano: 25. 6. 2018].

[31] Why games technology is driving automotive design. Dosegljivo: https://www.just-auto.com/interview/why-games-technology-is-driving-automotive-design-unreal-engine-qa_id171982.aspx. [Dostopano: 25. 6. 2018].

[32] UE Awards. Dosegljivo: https://www.unrealengine.com/en-US/awards. [Dostopano: 25. 6. 2018].

[33] Forward Shading Renderer for VR. Dosegljivo: https://docs.unrealengine.com/en-us/Engine/Performance/ForwardRenderer. [Dostopano: 05. 11. 2018].

[34] Unreal Studio. Dosegljivo: https://www.unrealengine.com/en-US/studio. [Dostopano: 25. 6. 2018].

[35] Valve. Dosegljivo: https://www.valvesoftware.com. [Dostopano: 25. 6. 2018].

[36] Projected size of the augmented and virtual reality market 2016-2022. Dosegljivo: https://www.statista.com/statistics/591181/global-augmented-virtual-reality-market-size/. [Dostopano: 10. 11. 2018].

[37] VR Best Practices. Dosegljivo: https://docs.unrealengine.com/en-us/Platforms/VR/ContentSetup. [Dostopano: 30. 7. 2018].

[38] VRroom. 'the sword of damocles', 1st head mounted display. Dosegljivo: https://vrroom.buzz/vr-news/guide-vr/sword-damocles-1st-head-mounted-display. [Dostopano: 03. 12. 2018].

[39] Wikipedia, the free encyclopedia. Perlin Noise. Dosegljivo: https://en.wikipedia.org/wiki/Perlin_noise. [Dostopano: 02. 11. 2018].

[40] Wikipedia, the free encyclopedia. View-master. Dosegljivo: https://en.wikipedia.org/wiki/View-Master. [Dostopano: 03. 12. 2018].

[41] Wikipedia, the free encyclopedia. Virtual boy. Dosegljivo: https://en.wikipedia.org/wiki/Virtual_Boy. [Dostopano: 03. 12. 2018].

[42] Wikipedia, the free encyclopedia. Autodesk Maya. Dosegljivo: https://en.wikipedia.org/wiki/Autodesk_Maya, 2018. [Dostopano: 25. 6. 2018].

[43] Youtube. Autodesk vr center of excellence in munich. Dosegljivo: https://www.youtube.com/watch?v=E5H3c1euX9o. [Dostopano: 03. 12. 2018].