Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Pavel Stare

VIZUALIZACIJA OMREŽIJ S POMOČJO OBOGATENE RESNIČNOSTI

diplomsko delo
na univerzitetnem študiju

Mentor: prof. dr. Franc Solina

Ljubljana, 2013

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01949 / 2013
Datum: 5.9.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **PAVEL STARE**

Naslov: **VIZUALIZACIJA OMREŽIJ S POMOČJO OBOGATENE RESNIČNOSTI**
VISUALIZATION OF NETWORKS USING AUGMENTED REALITY

Vrsta naloge: DIPLOMSKO DELO UNIVERZITETNEGA ŠTUDIJA

Tematika naloge:

Pri vizualni predstavitvi neplanarnih omrežij v dveh dimenzijah pride do prekrivanja povezav med vozlišči, kar ovira uporabnika pri preiskovanju omrežij. Z obogateno resničnostjo lahko uporabniku omrežje predstavimo v njemu bolj intuitivnih treh dimenzijah. Preglejte področji vizualizacije omrežij in obogatene resničnosti ter implementirajte prototip aplikacije, ki bo omogočala interaktivno vizualizacijo omrežij s pomočjo obogatene resničnosti v treh dimenzijah. Pri razporejanju vozlišč omrežja si pomagajte z obstoječimi rešitvami. Uporabite tehnologijo, ki omogoča stereografski prikaz obogatene resničnosti. Pri interakciji je pomembno, da se navidezni objekti in realni svet čim bolj zlivata, zato bodite pozorni na ustrezno obravnavanje delnega prekrivanja navideznih objektov in objektov iz realnega sveta. Ocenite tehnološke omejitve predstavljenega pristopa in predlagajte izboljšave.

Mentor: Dekan:

prof. dr. Franc Solina prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko
2
UNIVERZA V LJUBLJANI

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Pavel Stare, z vpisno številko **63080135**, sem avtor diplomskega dela z naslovom:

Vizualizacija omrežij s pomočjo obogatene resničnosti

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom prof. dr. Franca Soline,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 6.12.2013 Podpis avtorja:

Zahvaljujem se družini, prijateljem, sošolcem in vsem ostalim, ki so me podpirali pri študiju in izdelavi diplomskega dela. Hvala tudi mentorju, prof. dr. Francu Solini, še posebej pa as. Bojanu Klemencu za vso potrpežljivost, nasvete in dobro voljo.

Kazalo

Povzetek

V diplomski nalogi je prikazana izvedba vizualizacije omrežij v treh dimenzijah s pomočjo obogatene resničnosti. Za potrebe zaznavanja uporabnika v prostoru je bila uporabljena naprava Microsoft Kinect, za zajem stereoskopske slike ozadja, prikaz grafa in zaznavanje orientacije glave uporabnika pa očala za obogateno resničnost Vuzix Wrap 920AR. V prvem delu naloge se posvetimo pregledu obstoječih pristopov in tehnologij, ki smo jih proučili. V drugem delu opišemo metode dela, uporabljena orodja in tehnologijo, njihovo podrobnejše delovanje in pristope, ki smo jih uporabili v rešitvi. V tretjem, zadnjem, delu predstavimo težave, na katere smo naleteli, rezultate, ki smo jih dosegli, in ideje za nadaljevanje dela.

Ključne besede: omrežje, vizualizacija, obogatena resničnost, računalniška grafika, objektno programiranje.

Abstract

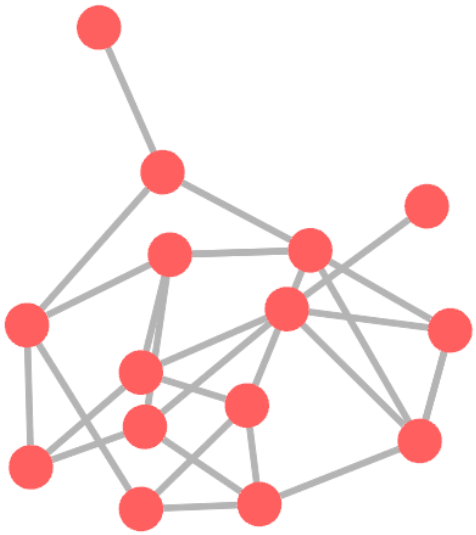
This thesis focuses on the implementation of a 3D network visualization application using augmented reality. Microsoft Kinect sensor was used for the purpose of user detection and Vuzix Wrap 920AR augmented reality eyewear for the purpose of acquiring the background, detecting head orientation and displaying the final image to the user. In the first part of the thesis we do an overview of the existing approaches and technologies pertaining to augmented reality and network visualization. In the second part we describe in greater detail the methodology, tools and technology we used in our implementation. In the third and final part we present the results, problems we encountered and ideas for further work.

Keywords: network, visualization, augmented reality, computer graphics, object-oriented programming

Poglavje 1

Uvod

Podatke o omrežjih najbolj naravno predstavimo s pomočjo povezanih grafov, torej vozlišč (angl. *nodes*) in povezav (angl. *edges*) – tako, kot lahko vidimo na sliki 1.1. Tak graf lahko učinkovito predstavimo s programskimi objekti,



Slika 1.1: Primer omrežja z vozlišči in povezavami med njimi.

ki se dobro obnesejo za analize s pomočjo raznih algoritmov (npr. iskanje poti, povezav, sorodstev, reševanje optimizacijskih problemov). Težave pa nastopijo, ko skušamo omrežja predstaviti ljudem grafično v dveh dimenzijah,

saj že pri razmeroma majhnem številu vozlišč tipično pride do velikega števila presečišč med povezavami in posledično do nepregledne slike.

Težavo lahko omilimo z uporabo različnih algoritmov, ki vozlišča razporedijo po površini tako, da minimalizirajo število presečišč med povezavami, ali pa izpostavijo določene pravilnosti, ki veljajo v grafu. S tem sicer problema ne rešimo, vseeno pa dele grafa, ki se nam zdijo pomembni že pred njegovim prikazom, lahko naredimo bolj jasne in pregledne.

V našem delu se prikaza lotevamo z drugega vidika. Graf s pomočjo računalniške grafike izrišemo v treh dimenzijah. Tako je problem presečišč med povezavami rešen, saj lahko potekajo na različni globini, pojavi pa se težava učinkovitega načina opazovanja in premikanja grafa v treh dimenzijah.

Dobro rešitev nam ponuja obogatena resničnost, s pomočjo katere lahko "v zrak" pred uporabnika na videz postavimo poljuben 3D objekt (v našem primeru graf) in s pomočjo razmaknjenih slik (disparitete, slika 1.2) ustvarimo

Slika 1.2: Stereoskopska fotografija - če sliko na levi gledamo z levim očesom, sliko na desni pa z desnim, dobimo zaradi različnik na obeh slikah vtis globine [36].

iluzijo, zaradi katere ga vidi globinsko, s pomočjo Kinecta in očal pa zaznavamo njegovo dejansko pozicijo v fizičnem prostoru in navidezne objekte prilagajamo njegovim premikom. Kinect omogoča tudi, da zaznamo, kje se

nahajajo uporabnikove roke, in dobimo podatke o tem, ali so zaprte ali odprte. Tako lahko uporabnik graf intuitivno manipulira in opazuje določene njegove lastnosti.

Pri tem pristopu imamo torej opravka z dvema "svetovoma". Prvi je fizični svet, kjer uporabnik z očali za obogateno resničnost stoji pred Kinectom, se giblje, obrača glavo in premika roki, drugi pa virtualni svet, v katerem se nahajajo predstavitev grafa in drugi nevidni objekti, ki so nam v pomoč pri prikazu. Uporabnik tako s pomočjo očal, ki zmorejo predvajati vsakemu očesu svojo sliko, naenkrat gleda v oba svetova in dobi vtis, da so tudi virtualni objekti del resničnosti. Zato je ključnega pomena, da sta svetova dobro poravnana in se ob uporabnikovih premikih tudi do čim večje mere enako obnašata.

Na podlagi te ideje smo si zastavili naslednje konkretne cilje, ki jih mora naša rešitev izpolnjevati:

- Prikazati graf in resnično sliko stereoskopsko, torej pri obeh prikazih ustvariti iluzijo globine.
- Zaznavati uporabnikove premike in v skladu z njimi prilagajati njegovo pozicijo v virtualnem svetu.
- Omogočiti uporabniku, da z gibi in gestami svojih rok lahko graf premika in ureja.
- Programsko rešitev napisati čim bolj modularno in pregledno.

V prvem delu naloge najprej opravimo hiter pregled področja vizualizacije grafov in obstoječe rešitve, potem pa si ogledamo še nekaj bolj znanih orodij za generiranje, ustvarjanje in urejanje omrežij. Nadaljujemo s podrobnejšim opisom uporabljenih tehnologij, njihovega delovanja, alternativnih možnosti in dodamo utemeljitve za njihovo izbiro. To so:

- Razvojno okolje Microsoft Visual C++ 2010 Express.
- Višje-nivojska grafična knjižnica OGRE (Object-Oriented Graphics Rendering Engine).

- Microsoft Kinect (strojna oprema in knjižnice za delo z njim).
- Vuzix Wrap 920AR (strojna oprema in podporne knjižnice).

V drugem delu najprej predstavimo dele razvojnega okolja in opozorimo na nekatere težave in nejasnosti, ki se pojavijo pri njegovi vzpostavitvi. Nato se osredotočimo na uporabo teh komponent in njihovo medsebojno delovanje znotraj našega projekta ter predstavimo konkretne rešitve, ki smo jih pri izvedbi uporabili. Pregledamo in natančneje opišemo tudi delovanje nekaterih pomembnejših delov izvorne kode.

V zadnjem delu se posvetimo težavam, na katere smo naleteli in jih ni bilo potrebno popolnoma odpraviti, oceni uspešnosti našega dela in idejam za nadaljevanje razvoja ter možnostih za razširitev funkcionalnosti.

Poglavje 2

Pregled področja

Naše delo združuje dve širši področji: obogateno resničnost in prikazovanje omrežij. Obe sta v zadnjem času močno napredovali (prvo zaradi vedno cenejših in učinkovitejših tehnoloških rešitev, drugo pa predvsem zaradi potrebe po analizi podatkov o družbenih in drugih podobnih omrežjih) in bili deležni tako v akademskih krogih, kot tudi v širši javnosti precejšnje količine pozornosti. Ne obstaja pa veliko projektov, ki bi obe ideji združevali in ju povezali v samostojno rešitev.

Ideja obogatene resničnosti (angl. *augmented reality*), torej vstavljanja neresničnih, navideznih predmetov v prostor pred uporabnika, ni popolnoma nova, čeprav je tehnološko mogoča šele zadnjih nekaj desetletij. V širšo javnost prodira ravno v današnjem času predvsem zaradi zadostne stopnje miniaturizacije procesnih komponent in splošnega napredka izdelave vse zmogljivejših in cenejših tankih zaslonov (npr. LCD, OLED), pa tudi vedno kompaktnejših zmogljivih digitalnih kamer. Pregled tedanje tehnologije in mogočih aplikacij na tem področju je že leta 1997 opravil Ronald T. Azuma [3], ki opiše in pregleda tudi možne načine uporabe na številnih področjih. Štiri leta kasneje je s še nekaterimi soavtorji pregled ponovil in ga dopolnil glede na tehnološke novosti [4]. V teh pregledih ugotovijo, da tehnologija obogatene resničnosti precej zaostaja za sestrsko navidezno resničnostjo predvsem zaradi težavnosti izvedbe v prenosljivi obliki in natančnem prekrivanju, vseeno pa bi bila uporabna v številnih scenarijih. Pomembna je tudi ideja, da uporabnik

lahko vso potrebno opremo in procesno moč nosi s seboj na podoben način, kot bi bila del obleke. Ta vidik se imenuje "nosljivo računalništvo" (angl. *wearable computing*), natančneje pa ga proučijo avtorji članka Augmented Reality Through Wearable Computing [17]. Najbliže realizacije obogatene resničnosti namenjene širši javnosti in ne samo ožjem krogu ljudi pa prav gotovo predstavlja Googlov Project Glass [38], ki je trenutno že v fazi beta testiranja. Njegove potencialne možnosti prouči R. Furlan v članku [11], kjer se poleg tega osredotoči tudi na samo zgradbo naprave in idejo izdelave njene reprodukcije s pomočjo delov ter naprav, ki so že na voljo v prosti prodaji. Zaključi, da bo po vsej verjetnosti v naslednjih desetih letih podobna tehnologija v razcvetu, saj je kljub temu, da je ta trenutek še v povojih, že v celoti izvedljiva, zanimanje zanjo pa tudi iz meseca v mesec samo še narašča.

Za razliko od obogatene resničnosti pa omrežja in povezani grafi skupaj s potrebno teoretično podlago segajo precej dlje v preteklost. To, da z njimi lahko učinkovito predstavimo podatke, ki predstavljajo entitete in povezave med njimi, so ugotovili že v prvi polovici osemnajstega stoletja. Za začetnika teorije grafov velja Leonhard Euler, ki jo je utemeljil v svoji rešitvi problema sedmih mostov Königsberga leta 1735 [1]. Kasneje so poleg splošnih grafov avtorji posebno pozornost posvečali tudi specifičnim podvrstam grafov, predvsem drevesom. Grafi so se kasneje izkazali za uporabne na številnih znanstvenih področjih za nazoren opis in posredovanje določenih problemov drugim strokovnjakom ter so tako pomagali k uspešnejši komunikaciji med njimi. K popularizaciji in širši uporabi grafov je pripomogel predvsem Frank Harary, profesor matematike na univerzi v Michiganu, ki je leta 1960 izdal učbenik z naslovom Graph Theory [12], ki je s teorijo grafov ter njihovo uporabnostjo seznanil študente in znanstvenike z različnih področij. Grafi so zaradi svoje strukture entitet in povezav zelo primerni za računalniško analizo. S pomočjo algoritmov na grafih lahko učinkovito rešujemo določene probleme, kot je na primer barvanje zemljevida s štirimi barvami [2], katerega so celo dokazali z uporabo računalnikov.

V večini primerov, ko grafe analiziramo, pa jih želimo na neki stopnji tudi

grafično prikazati. Z njihovo vizualizacijo se še vedno tudi v Sloveniji ukvarjajo številni raziskovalci. Na tem mestu lahko omenimo članke, kot sta na primer Pajek: A Program for Large Network Analysis [5], kjer avtorja predstavita zmogljivosti in delovanje programa za analizo velikih omrežij "Pajek" ter Analiza kompleksnih omrežij: osnovni pojmi in primeri uporabe v praksi [18], kjer avtorji najprej predstavijo teorijo grafov, nato pa se osredotočijo na možnosti uporabe omenjenih postopkov v okviru javne uprave. Iz članka je razvidno, da ustrezen prikaz podatkov lahko močno vpliva na količino informacije, ki jo graf posreduje opazovalcu in je zato pomemben vidik dela z grafi.

Združevanje prikazovanja grafov s pomočjo obogatene resničnosti je relativno novo področje. Podobno kot sama tehnologija obogatene resničnosti, se ideje prikazovanja in upravljanja z grafi na ta način pojavljajo šele v zadnjem času. Točno ta vidik obravnava v enem svojih člankov D. Bechler [6], kjer razišče uporabo simulatorjev dotika (angl. *tangible interface*) in klasični stereoskopski prikaz. Zaključi, da metoda, pri kateri dobi uporabnik tudi čutni odziv dotika za analizo povezav v grafih obnese zelo dobro, po drugi strani pa zgolj stereoskopska globinska predstava ne zadošča povsem. Nekoliko podobne težave se je v svojem magistrskem delu lotil tudi R. Liebo [14], kjer s pomočjo obogatene resničnosti prikazuje grafe matematičnih funkcij in uporabniku omogoča njihovo boljšo predstavo.

Poglavje 3

Pregled tehnologij in orodij

Eden večjih izzivov pri sami implementaciji naše rešitve tiči v učinkovitem kombiniranju precejšnjega števila različnih knjižnic (angl. *library*) in orodjarn (angl. *toolbox*) Na nekaterih področjih izbire praktično ni bilo, na drugih pa smo se lahko odločali med celo vrsto opcij, izmed katerih ima vsaka svoje dobre in slabe strani.

3.1 Operacijski sistem

Želeli smo si, da bi bila rešitev čim bolj neodvisna od operacijskega sistema, a na žalost zaradi določenih dejavnikov to ni bilo izvedljivo. Čeprav je večina orodij, kot sta na primer OGRE [31] in OpenCV [32] platformno neodvisna in ju lahko prevedemo tako v okoljih Linux, Windows in OSX, uradni gonilniki za Kinect in Vuzix Wrap niso odprtokodne narave in so prevedeni samo za uporabo v operacijskem sistemu Windows.

Zato smo se odločili za kompromis in izbrali okolje Windows, obenem pa med izvedbo pazili, da je rešitev čim bolj modularna in bi kasneje lahko popravili ali zamenjali samo določene komponente, odvisne od operacijskega sistema. Na ta način bi bilo mogoče ob izdaji primernih gonilnikov in knjižnic tudi za Linux in OSX izvorno kodo razmeroma enostavno prilagoditi za uporabo z njimi.

3.2 Programski jezik in prevajalnik

Podobno, kot pri izbiri operacijskega sistema, je tudi tukaj odločitvi v največji meri botrovala omejenost s knjižnicami. Velika večina vseh orodij, ki smo jih našli med raziskovanjem področja, je namreč napisana v jeziku C ali C++, vključno z grafično knjižnico OGRE ter uradnimi gonilniki in orodji za delo s Kinectom in očali Vuzix.

Posledično smo si za razvoj izbrali jezik C++, ki je poleg združljivosti z danimi orodji tudi hitrejši od večine alternativ, kot sta na primer C# in Java ali celo interpreterski jeziki, kot je Python [10] in je za naš namen prikazovanja 3D objektov v realnem času, kjer je hitrost prikazovanja pomembna, primernejši.

Ob teh predpostavkah smo pri izbiri prevajalnika imeli dve očitni možnosti. Ena je odprtokodna zbirka orodij MinGW, ki vključuje večino zbirke programov GNU, prirejenih za okolje Windows, druga pa Microsoftov prevajalnik Visual C++ 2010 Express.

MinGW[1] (Minimalst GNU for Windows) je minimalistično razvojno okolje namenjeno izdelavi aplikacij Windows. Vključuje prevajalnike in razhroščevalnike za jezike C, C++, ADA in Fortran, ki so prirejeni specifično za Windows iz odprtokodnega paketa razvijalske opreme GNU, ki je v osnovi namenjen uporabi na sistemih Linux. Kljub temu deluje izključno na Windowsovih knjižnicah in ne implementira svojega okolja POSIX. Prav tako okolje samo na sebi ne vključuje grafičnega uporabniškega vmesnika, ki bi programerju delo olajšal z dopolnjevanjem imen, hitrim označevanjem napak ali izdelavo oken na način "vleci in spusti". Obstajajo pa številne druge kvalitetne in proste rešitve, ki te funkcionalnosti dodajo. Med bolj znanimi sta na primer Eclipse[2] in Code::Blocks[3].

MS Visual Studio[4] je paket programske opreme za razvijalce v Windows okolju, ki ga ponuja Microsoft. Poleg svojih prevajalnikov za jezike C/C++, C# in Visual Basic vsebuje tudi orodja za razhroščevanje, delo s

[1] MinGW – uradna stran. Dostopno na: http://www.mingw.org/
[2] Eclipse – uradna stran. Dostopno na: http://www.eclipse.org/
[3] Code::Blocks – uradna stran. Dostopno na: http://www.codeblocks.org/
[4] Visual Studio - uradna stran. Dostopno na: http://www.visualstudio.com/

projekti, pomoč pri pisanju kode in grafični vmesnik za izdelavo oken. Je tudi uradno okolje za razvoj programske opreme v okolju Windows. Poleg nekaj plačljivih različic (Ultimate, Premium in Professional), za nekomercialno rabo obstaja tudi različica Express, ki jo lahko kdorkoli brezplačno naloži z uradnega medmrežnega strežnika in uporablja za razvoj programske opreme v nekomercialne namene. Izbire tudi tukaj pravzaprav ni bilo, saj sta uradni knjižnici tako za Kinect, kot tudi za Vuzix očala, ki smo jih že od začetka nameravali uporabljati, že vnaprej prevedeni in delujeta samo v kombinaciji z MS Visual C++. Večina ostalih knjižnjic (npr. OGRE in OpenCV) je sicer že na voljo v prevedeni obliki za MinGW, ampak lahko brez težav dobimo njihovo izvorno kodo in jo prevedemo v (načeloma) poljubnem prevajalniku za C++.

Tako je uporaba vseh potrebnih komponent v resnici mogoča samo s pomočjo Microsoftovega Visual Studia. Odločili smo se, da našo rešitev izpeljemo s pomočjo Visual C++ 2010 Express.

3.3 Orodja za delo z omrežji

Področje ustvarjanja in urejanja omrežij je sicer za večino običajnih računalniških uporabnikov precej slabo poznano in posledično tudi nekoliko slabše zastopano z vidika obstoječih rešitev. Vsaj v primerjavi z urejevalniki besedil, slik, multimedije in podobnih vsebin, kjer je izbira zelo pestra.

Kljub vsemu je v raziskovalnih krogih, kjer je tipično v igri večja količina podatkov in je prisotna tudi potreba po prikazu zapletenih omrežij, precej razširjenih kar nekaj rešitev. Z njihovo pomočjo lahko omrežja kreiramo na novo, jih ustvarimo na podlagi obstoječih podatkov, ali pa uvozimo že obstoječe in jih samo na želen način uredimo.

Orodja so si sicer med seboj večinoma podobna in nam omogočajo precej enakovredne operacije, se pa v določenih pogledih tudi razlikujejo. Poleg nekoliko drugačnih uporabniških vmesnikov, nekatera vsebujejo tudi razvojna orodja in knjižnice ter jih je tako mogoče integrirati in neposredno uporabljati

v drugih aplikacijah. Druga so izdelana zgolj z namenom prikazovanja in izrisovanja grafov, spet tretja pa bolj za gradnjo in urejanje.

V nadaljevanju bomo na kratko predstavili nekaj orodij, ki smo si jih v okviru projekta ogledali, in pobliže spoznali Tulip, ki smo ga uporabili v sklopu naše rešitve.

3.3.1 GraphViz

GraphViz [24] je svojo pot začel še pred letom 2000 v komercialnih vodah, kjer ga je začel razvijati razvojni oddelek podjetja AT&T Labs. Kljub vsemu so se razvijalci odločili, da svoj izdelek izdajo v odprti kodi pod okriljem Eclipse Public Licence. S tem so k nadaljevanju razvoja pritegnili še druge razvijalce, še vedno pa tudi sami skrbijo za uradne in stabilne izdaje svoje kreacije.

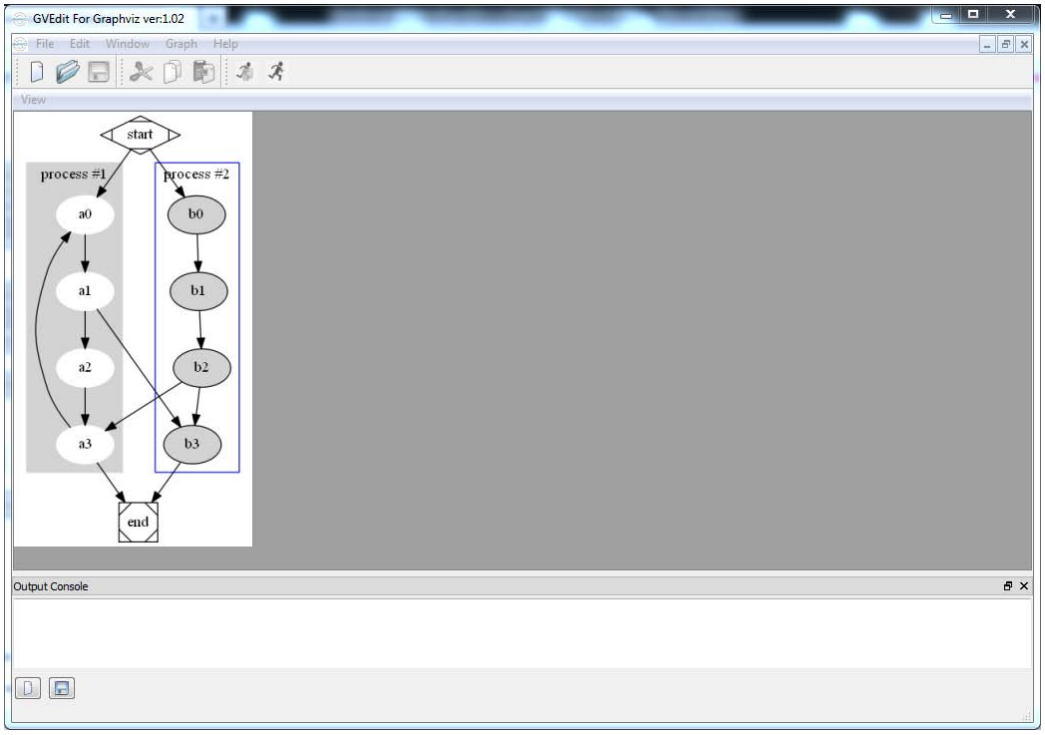
Orodje je namenjeno predvsem vizualizaciji diagramov in omrežij večinoma v dveh dimenzijah. Podatke zmore uvoziti iz različnih tekstovnih oblik, strukturo uvoženih grafov pa naravno hrani zapisano v jeziku DOT[5]. Sestavljeno je iz različnih komponent, ki opravljajo svoje funkcije.

Jedro skrbi le za uvoz in obdelavo podatkov v jezik DOT ter za izgradnjo in izvoz diagramov v obliki vektorskih in bitnih slik (npr. SVG , PDF, PostScript, PNG, JPG). Samo po sebi ima vmesnik samo na nivoju ukazne vrstice in tako deluje skoraj izključno kot prevajalnik opisa oblike diagrama v slikovno reprezentacijo.

Paketu so dodani še drugi moduli, ki uporabniku omogočajo urejanje prikaza grafov na različne bolj specifične načine, dodajajo možnosti za obdelavo večjih podatkovnih množic in implementirajo grafični uporabniški vmesnik, kot ga vidimo na sliki 3.1.

Orodje je napisano v jeziku C/C++, in razen vmesnika z ukazno vrstico ne premore paketa knjižnic, ki bi jih lahko neposredno uporabljali v drugih programih. Obstaja le vmesnik za Javo, ki pa v ozadju prav tako uporablja ukazno vrstico, kar pomeni, da na ukaz le zažene jedro GraphViza v ločenem

[5]Gramatika DOT jezika. Dostopno na: http://www.graphviz.org/doc/info/lang.html



Slika 3.1: GVEdit - grafični vmesnik v zbirki GraphViz

procesu, ki nato izvede zahtevano operacijo. Poleg tega je usmerjen predvsem v prevajanje že izdelanih DOT skript v praviloma 2D reprezentacije in ne toliko v samo razpostavljanje množice podatkov v 3D prostor na učinkovit način. Tako za naš namen ni najbolj primeren.

3.3.2 GUESS

GUESS - The Graph Exploration System [25] je odprtokodno orodje izdelano v Javi in namenjeno predvsem urejanju strukture prikaza grafov v 3D prostoru. Vključuje osnovni grafični uporabniški vmesnik, ki uporabniku omogoča uvoz podatkov v večini standardnih formatov (npr. GML, Pajek), nato pa tudi urejanje njihove razporeditve v 3D prostoru z algoritmi ali ročno. Uporabnik si nato lahko izbere želen zorni kot pogleda na oblikovan graf in pogled izvozi v enem od številnih vektorskih ali rasterskih grafičnih formatov (npr. PNG,

GIF, PDF, PostScript, SVG).

Program je že pred leti izdelal Eytan Adar na podlagi svojega prejšnjega projekta Zoomgraph [47] v okviru raziskovalne skupine pod okriljem podjetja HP.

Vsebuje tudi možnost pisanja skript v jeziku Jython [16] – Pythonu, ki deluje na javanski osnovi. Poleg neposredne uporabe z grafičnim vmesnikom, pa je mogoče do njegovih komponent dostopati neposredno v Javi. Že od Zoomgrapha naprej specifično zanj obstaja tudi vmesnik za uporabo v statističnem programskem orodju R[6].

Kljub temu, da dobro podpira 3D razporeditve grafov, na žalost omogoča programski vmesnik samo za Javo in R [13], zato bi ga bilo v C/C++ zelo težko vključiti.

3.3.3 IGraph

IGraph [26] je odprtokodna zbirka funkcij z GNU licenco namenjena uvozu, obdelavi, izvozu in prikazovanju omrežij. Napisana je v C/C++ specifično za uporabo v Pythonu in jeziku za statistične obdelave podatkov R.

Glede na to, da je samo programska knjižnica namenjena uporabi v sklopu drugih skriptnih jezikov, ne vsebuje grafičnega vmesnika. Niti ni enostavno dostopna iz kode C/C++ in temelji predvsem na urejanju in prikazovanju grafov v dveh dimenzijah.

3.3.4 UCINET

UCINET [8] je močno orodje namenjeno predvsem analizi in vizualizaciji podatkov, pridobljenih na podlagi družabnih omrežij. Izdelali so ga Lin Freeman, Martin Everett in Steve Borgatti na univerzi v Harvardu. Za razliko od večine ostalih orodij je plačljivo, omogoča pa 90 dnevno preizkusno obdobje brez omejitev.

[6] R – uradna stran. Dostopno na: http://www.r-project.org/

Deluje izključno v operacijskem sistemu Windows, omogoča pa uvoz, pretvorbo podatkov, izvoz v številnih formatih, algoritme za analizo in razporejanje vozlišč, možnost pisanja in uporabe lastnih algoritmov, urejanje podatkov v tabelah in na grafu ter še celo vrsto drugih opcij, ki se jim nismo posebej posvetili.

Kljub vsemu pa ne vsebuje knjižnic, ki bi jih bilo mogoče programsko uporabiti.

3.3.5 Pajek

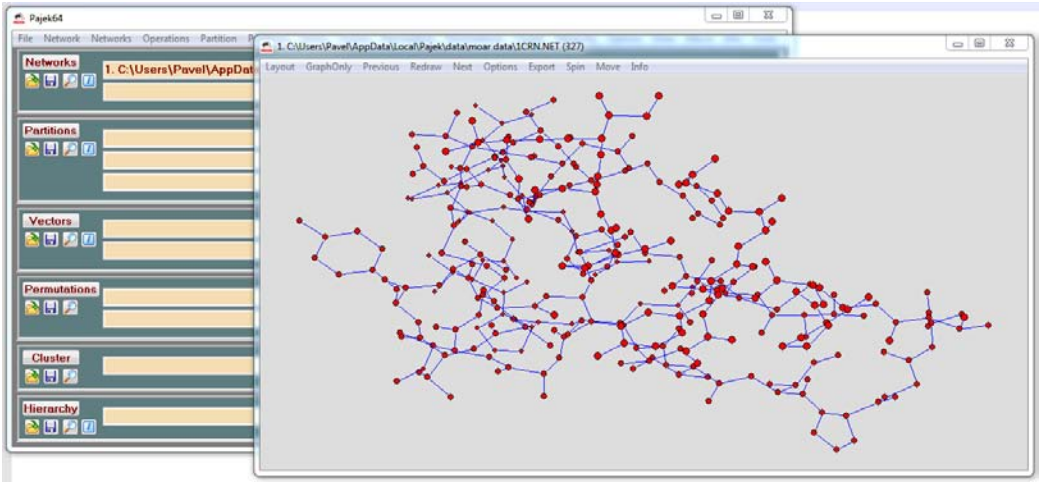
Pajek [5, 35] je prosto dostopno, a ne odprtokodno, orodje za analizo in prikaz velikih omrežij, ki so ga razvili Vladimir Batagelj, Andrej Mrvar in Matjaž Zaveršnik na fakulteti za matematiko in fiziko univerze v Ljubljani. Sprva je bilo namenjeno obdelavi podatkov o socialnih omrežjih, ki so tipično zelo velika in zapletena. Zato ga lahko učinkovito uporabimo tudi za analizo in prikaz vsakršnih zelo velikih podatkovnih množic.

Orodje je svetovno znano in po zmogljivostih in namenu še najbolj podobno UCINET-u. Omogoča zajem, obdelavo, izvoz in prikaz podatkov. Čeprav je prosto dostopno, deluje samo v okolju Windows in ne vsebuje programskih vmesnikov v obliki knjižnic, ki bi omogočali neposredno uporabo preko drugih aplikacij. Izgled grafičnega uporabniškega vmesnika prikazuje slika 3.2.

3.3.6 GEPHI

Gephi [23] je odprtokodno in prosto dostopno orodje s poudarkom na urejanju in oblikovanju prikaza grafov v treh dimenzijah. Njegov razvoj je v rokah skupine navdušencev pod vodstvom Mathieua Bastiana. Poleg standardnih funkcij uvoza, izvoza in zajema zaslonskih slik omogoča tudi razporejanje vozlišč in pisanje algoritmov za njihovo razporejanje.

Orodje temelji na Javi, kar nam sicer omogoča, da ga lahko uporabljamo na vseh sodobnejših operacijskih sistemih, po drugi strani pa pri večjih grafih postane nekoliko počasno in ne najbolj uporabno.



Slika 3.2: Začetno okolje programa Pajek z odprtim urejevalnikom grafa

Z uporabniškim vmesnikom, ki je obenem enostaven za uporabo in precej močan, lahko spreminjamo velikosti vozlišč, njihovo obliko, barvo, tip in izgled povezav med njimi. Lahko tudi avtomatično analiziramo graf in ga tako tudi vizualno na primer razdelimo na med seboj bolj povezane dele s pomočjo novega položaja vozlišč in njihove barve.

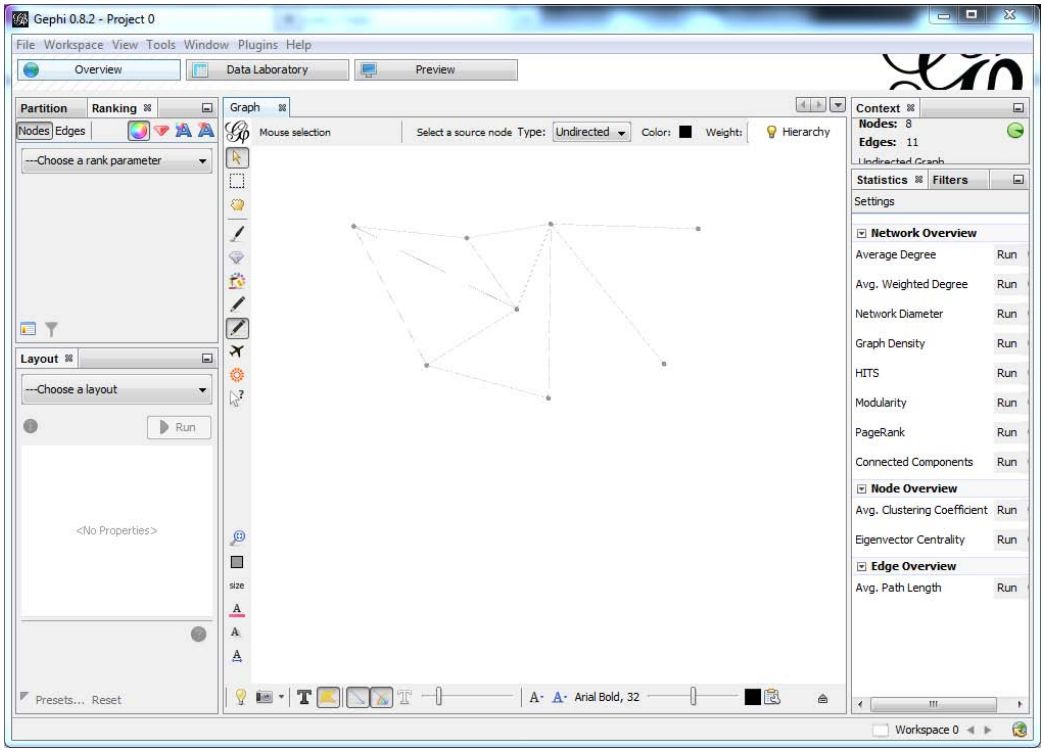
Poleg grafičnega vmesnika (ki je prikazan na sliki 3.3) nam Gephi ponuja tudi širok javanski programski vmesnik [22], ki je pregleden, dobro dokumentiran in nam omogoča enostavno uporabo vseh funkcij, ki so dostopne tudi grafično.

Na žalost pa ga ni mogoče vključiti neposredno v projekte C/C++ brez izdelave dodatnega vmesnika.

3.3.7 Tulip

Tulip [43] je več-platformna odprtokodna rešitev, ki so jo začeli razvijati v laboratoriju LaBRI na univerzi v Bordeauxu, kjer še vedno bdijo nad uradnimi izdajami, sodelujejo pa tudi z zunanjimi razvijalci, ki lahko dodajajo svoje prispevke.

V svoji osnovi je to knjižnica programskih funkcij, napisana v jeziku



Slika 3.3: Okolje Gephi, način neposrednega urejanja grafov

C/C++. Poleg dobro dokumentiranega programskega vmesnika vsebuje tudi grafični uporabniški vmesnik, ki je v svojih funkcionalnostih precej podoben Gephijevemu.

V celoti deluje v okoljih Windows (s pomočjo ogrodja QT[7]), Linux in OSX in zmore učinkovito delo tudi z velikimi grafi (več kot 1000 vozlišč).

Največja lepota Tulipa pa tiči v njegovi razširljivosti. Zgrajen je namreč na način, da skoraj vse komponente - od algoritmov za uvoz, izvoz, iskanje pravilnosti, do razporejanja vozlišč v 2D in 3D prostoru ter mnogih drugih - lahko napišemo sami v obliki posebnega modula (s pomočjo jezika Python ali pa na nižjem in zmogljivejšem nivoju s C/C++) in jih nato vstavimo med ostale komponente, kjer se vsaj na videz z njimi popolnoma zlijejo.

Glede na to, da je prav tako kot naša rešitev zgrajen v jeziku C/C++ in

[7]Qt – uradna stran. Dostopno na: http://qt-project.org/

vsebuje dobro dokumentiran programski vmesnik, smo se odločili, da bomo del naše rešitve namenjen oblikovanju grafov zgradili na njegovi osnovi.

Zato se mu bomo na tem mestu tudi nekoliko podrobneje posvetili in si ogledali nekatere zmogljivosti, ki smo jih uporabili.

Grafični uporabniški vmesnik

Po namestitvi s pomočjo čarovnika, v kateri lahko izberemo pot, kjer Tulip ustvari svoje delovno okolje in kamor naloži tudi knjižnice svojega programskega vmesnika, lahko zaženemo grafični uporabniški vmesnik.

Najprej se na zaslonu pojavi okno, v katerem lahko izbiramo med delom s projekti (torej neposredno z grafi), ali pa urejamo zbirko modulov. Pregledovalnik modulov nam poleg vklapljanja in izklapljanja posameznih komponent omogoča tudi iskanje po širšem spletnem repozitoriju, kjer lahko z uradnega strežnika naložimo nove algoritme in tako razširimo svojo orodjarno. Tu lahko tudi uvozimo in vklopimo module, ki smo jih napisali sami in jih želimo uporabiti v katerem od projektov.

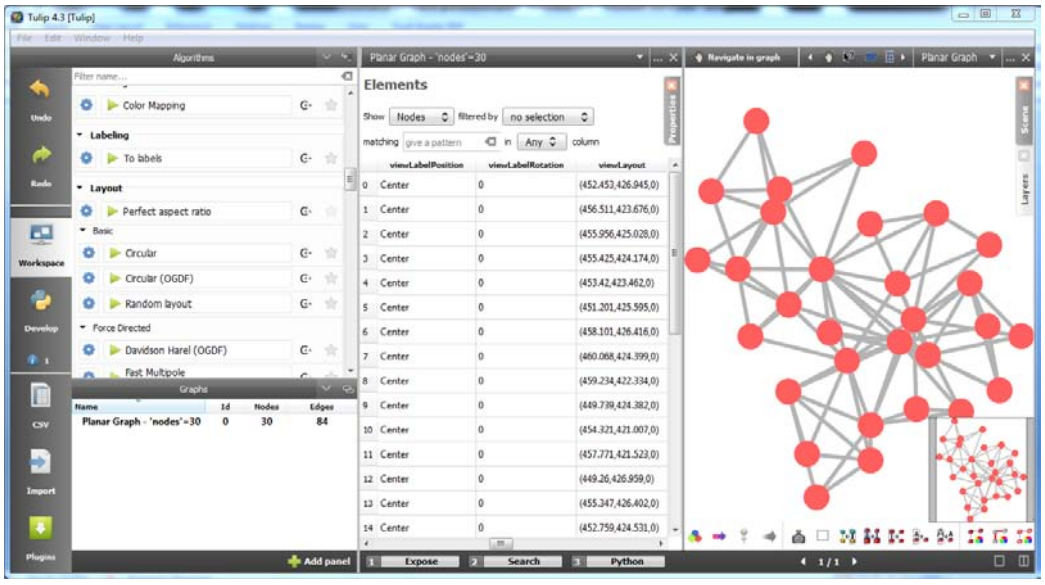
Vsak graf v Tulipu je vsebovan v svojem projektu. Začetek novega projekta nam tako ponudi prazno delovno okolje, v katerem lahko izdelamo in oblikujemo svoj graf. To lahko storimo na tri načine:

- Graf generiramo naključno s pomočjo enega od vgrajenih algoritmov.
- Graf izdelamo sami z določanjem in povezovanjem vozlišč.
- Graf s pomočjo enega od algoritmov generiramo na podlagi zunanjih podatkov (npr. datotečni sistem, facebook).

Glavna razlika med različnimi naključno generiranimi grafi je v njihovi obliki in omejitvah generiranja. Lahko ga kreiramo povsem naključno in določimo samo želeno število vozlišč in povezav, ki jih potem računalnik poljubno razporedi (tako določena vozlišča lahko ostanejo tudi brez povezav), lahko izberemo polno povezan graf (vsako vozlišče je povezano z vsakim) in tako določimo samo število vozlišč, lahko pa se odločimo za katero od bolj pravilnih variant, kot je mrežasta razporeditev, pri kateri so vozlišča razporejena

v vrstice in stolpce ter povezave lahko tečejo samo med neposrednimi sosedi, ali pa postavimo pogoj, da mora biti graf "planaren" in se v 2D predstavitvi nobena povezava med dvema vozliščema ne sme sekati s katerokoli drugo povezavo.

Že na začetku, v praznem okolju, ali pa na osnovi naključno zgrajenega grafa, lahko seveda vse poglede grafa tudi ročno urejamo. Tako lahko v pogledu tabele, ali pa s pomočjo orodij v grafičnem prikazu dodajamo ali brišemo vozlišča, dodajamo ali odstranjujemo povezave med njimi, vozliščem lahko spreminjamo tudi lokacijo, barvo in prosojnost, obrobo, obliko, velikost in še mnoge druge lastnosti, ki jih nosijo. Na tem mestu lahko vozlišča tudi preimenujemo, ali pa jim dodamo določene vrednosti, ki sicer na sam prikaz ne vplivajo, pomagajo pa pri uporabnikovem določanju pomena vozlišč. Vse poglede prikaza besedila seveda lahko prav tako urejamo preko lastnosti vozlišča. Na sliki 3.4 je prikazan grafični vmesnik.



Slika 3.4: Tulip, na skrajni levi so splošne opcije okolja, poleg njih zgoraj seznam algoritmov, spodaj pa seznam odprtih grafov, na sredini je tabelarični pogled na lastnosti vozlišč, na desni pa grafični prikaz grafa; spodaj lahko odpremo tudi Pythonovo ukazno vrstico.

V verziji 4.3 so tudi že vgrajeni trije algoritmi za uvoz podatkov iz zunanjih virov in njihovo pretvorbo v omrežja. Tako lahko z grafom prikažemo drevesno strukturo svojega datotečnega sistema, poljubne spletne strani, ali pa uvozimo podatke o prijateljstvih z družbenega omrežja Facebook. Poleg njih, nam Tulip ponuja še možnost, da odpremo projektno datoteko iz katerega od drugih orodij za delo z grafi. Na ta način lahko odpremo in uporabljamo še Gephijev GFX format, graf zapisan v splošnem GML (Graph Markup Language), ki ga uporabljajo številne rešitve, Graphvizov projekt, pa tudi grafe shranjene s Pajkom in UCINET-om.

V vsakem primeru pa lahko graf, ki ga zgradimo na tak ali drugačen način naknadno z algoritmi preurejamo. Večinoma gre tukaj za oblikovno razporeditev vozlišč in boljšo preglednost grafa (angl. *layout*). Lahko pa tudi topološko analizo (npr. iskanje ciklov, preverjanje planarnosti, povezanosti) ali pa avtomatično preurejanje grafa na način, da kateremu izmed teh topoloških pogojev zadosti. Poleg teh algoritmov obstaja še množica takih, ki so namenjeni ugotavljanju določenih mer v grafu ter nekaj bolj splošnih. Med njimi je na primer tudi orodje za določanje velikosti vozlišč glede na njihovo pomembnost, ki jo lahko ugotovimo na primer glede na skupno število povezav, ki se stikajo v posameznem vozlišču.

Programski vmesnik

Ker Tulip v svojem jedru ni grafično orodje za neposredno delo z grafi, ampak knjižnica programskih funkcij, sta temu primerna tudi njihova fleksibilnost in moč.

Programski vmesnik nam ponuja dva načina dostopa. Klasičnega preko neposredne uporabe programskih knjižnic iz jezika C/C++ (z dodanim paketom Qt), ali pa posrednega s skriptami, ki jih napišemo v jeziku Python in nam omogočajo malo šibkejšo, a veliko dostopnejšo pot.

V našem projektu smo se odločili za drugo pot. Z uporabo skript smo lahko dosegli vse želene operacije, obenem pa smo se izognili težavam v skladnosti s programskim okoljem Visual C++ 2010, saj je uradno v operacijskem sistemu

Windows namenjena samo rabi v kombinaciji s prevajalnikom MinGW/GCC. Obenem pa za pravilno prevajanje projektov, ki jo uporabljajo potrebuje nameščeno tudi razvojno okolje Qt. Glede na to, da smo stremeli k čim večji neodvisnosti naše rešitve od obsežnih in nepotrebnih zunanjih knjižnic, smo se zato v tem primeru raje odločili za uporabo skript.

3.4 Grafični pogoni

Za učinkovito delo in prikazovanje grafičnih objektov v treh dimenzijah, se skoraj ne moremo izogniti uporabi grafičnih pogonov. To so programski vmesniki, ki nam omogočajo, da grafične objekte zgradimo, postavimo v 3D prostor in rezultat iz želenega zornega kota pokažemo uporabniku na 2D ekran. Pri tem pa nam delo olajšajo s tem, da zapletenejše matematične operacije in strojne vidike delovanja večinoma skrijejo v ozadje, nam pa ponudijo učinkovite višje nivojske funkcionalnosti, s pomočjo katerih se lahko osredotočimo na objekte in njihov prikaz.

Podobno, kot pri orodjih za delo z omrežji, tudi tu obstaja cela vrsta orodjarn, ki so nam na voljo. Med sabo se seveda razlikujejo v številnih pogledih. V glavnem pa bi jih lahko razdelili na tri skupine:

- grafična API-ja DirectX in OpenGL,
- grafični pogoni,
- pogoni za igre.

Grafična API-ja DirectX in OpenGL oba predstavljata temelj za vsakršno resnejše delo z računalniško grafiko. Nista prava grafična pogona, saj delujeta na precej nižjem nivoju neposrednega dostopa do grafične strojne opreme in tako predstavljata osnovo, na kateri je velika večina grafičnih pogonov zgrajena. Obe knjižnici ponujata precej podobne funkcionalnosti, razlikujeta se predvsem v tem, da je DirectX[8] komercialno orodje pod okriljem

[8]DirectX – neuradna stran s pomočjo. Dostopno na: http://www.computerhope.com/directx.htm

Microsofta in zato deluje samo na operacijskih sistemih Windows (kot tudi aplikacije, ki jo posredno ali neposredno uporabljajo), OpenGL[9] pa prosto dostopna knjižnica, ki pod okriljem neprofitne organizacije Khronos Group deluje na številnih modernejših operacijskih sistemih. Lahko jih uporabljamo neposredno, še večkrat pa služita kot podlaga za prave grafične pogone na višjem nivoju.

Grafični pogoni (angl. *graphics engines*) so orodjarne zgrajene na podlagi osnovnih grafičnih knjižnic, ki skrijejo kompleksnost dela z njimi in omogočajo programerju, da obide tehnične podrobnosti postopka prikazovanja grafičnih objektov na ekran in se tako osredotoči na širšo sliko delovanja programa in manipulacije z želenimi objekti. Po drugi strani pa praviloma ne podpirajo nobenih ne-grafičnih vidikov, kot so predvajanje zvoka, napreden zajem interakcij ali implementiranje fizikalnega modela delovanja med objekti.

Pogoni za igre (angl. *game engines*) so grafični pogoni razširjeni drugimi dodatki, ki omogočajo tudi razne ne-grafične funkcionalnosti, predvsem fizikalnim pogonom. Poleg postavitve in prilagajanja grafičnih objektov v prostor, tako omogočajo tudi simulacijo interakcije med zapletenejšimi objekti, kot so trki, deformacije in podobno. Večinoma se uporabljajo za razvoj računalniških iger, od koder izvira tudi njihovo ime.

Prva skupina se od ostalih dveh v večini primerov precej razlikuje, saj DirectX in OpenGL vsebujeta samo funkcije in podatkovne strukture, ki nam omogočajo neposreden dostop do strojnih zmogljivosti grafičnih kartic in tako ne posegata na višji nivo. Poleg tega tako grafični pogoni, kot tudi pogoni za igre skoraj vedno neposredno uporabljajo eno od teh dveh knjižnic na način, da jo ovijejo z vmesnikom na višjem nivoju, (nekateri ovijejo celo obe in tako programerju ali uporabniku pred samim zagonom programa dajo na izbiro, katero želi, da uporabijo) potem pa njune funkcionalnosti samo še razširijo z dodatnimi, bolj specializiranimi opcijami.

Po drugi strani pa med grafičnimi pogoni in pogoni za igre pogosto ni povsem jasne ločnice. Še najpomembnejši kriterij je vsebovanost fizikalnega

[9]OpenGL – uradna stran. Dostopno na: http://www.opengl.org/

pogona, ki poskrbi za (čim bolj) realistično dinamiko in vpliv med navideznimi predmeti (npr. gravitacija, trki). Kljub temu tudi določeni sicer grafični pogoni lahko vključujejo (okrnjen) fizikalni pogon. Razliko tako določa predvsem odločitev avtorjev, kako svoj pogon poimenujejo in na čem je njegov poudarek, kot tudi najpogostejši način uporabe pogona s strani razvijalcev, ki ga uporabijo v svojih rešitvah.

V našem primeru smo se tako morali odločiti, katere funkcionalnosti potrebujemo in katera od rešitev je v danih okoliščinah najbolj smiselna. Neposredna uporaba DirectX ali OpenGL je namreč razmeroma zapletena in bi zahtevala veliko dodatnega dela že pri postavitvi in upodabljanju povsem enostavnih objektov, po drugi strani pa za prikaz grafov ne potrebujemo naprednih vmesnikov za "igralsko" interakcijo z uporabnikom, niti polnopravnega fizikalnega pogona. Zato smo se odločili, da so najboljša opcija grafični pogoni.

Ugotovili smo, da je izbire na tem področju veliko. V nadaljevanju bomo zato predstavili nekaj rešitev, med katerimi smo se odločali.

3.4.1 SDL (Simple Directmedia Layer)

SDL [39] je odprtokodna rešitev, ki integrira tako DirectX, kot OpenGL in poenostavi delo z njima. Sicer ni povsem polnopraven grafični pogon, saj funkcij grafičnih knjižnic ne skrije povsem, niti ne implementira grafa scene, vseeno pa močno olajša delo s poenotenim pristopom in načini za manipulacijo grafičnih objektov na višjem nivoju. Podpira delo z 2D in 3D grafiko in vsebuje veliko uporabnih operacij, kot je na primer uvoz in urejanje tekstur, rezanje in lepljenje njihovih delov, pa tudi funkcije za risanje nekaterih primitivov (npr. kvader, krogla, kvadrat, krog, elipsa, črta) in delo z njimi. Vsebuje celo nekaj komponent za odpiranje in predvajanje zvočnih datotek, čeprav za pravilno delovanje teh funkcij zahteva nameščen OpenAL.

Kot večina orodij, ki imajo neposreden opravek z grafičnimi knjižnicami, je napisan v jeziku C++, preko katerega ga najlaže uporabljamo. Kljub temu vsebuje že vgrajene možnosti za povezavo z drugimi jeziki, kot sta C# in

Python.

SDL je razmeroma majhna orodjarna in se ravno v tem tudi odlikuje, saj za začetek razvoja ne potrebuje veliko priprav. Prav tako za delovanje potrebuje samo eno dodatno .dll datoteko. Je hitra in razmeroma enostavna za uporabo. Poleg tega je zelo dobro dokumentirana in velikokrat uporabljena. Brez napora lahko hitro najdemo številne primere uporabe raznih vidikov, ki so nam pri razvoju lastne aplikacije lahko v veliko pomoč.

Ima pa predvsem zaradi svoje enostavnosti in jasnosti tudi nekaj pomanjkljivosti. Glavna izmed njih je pomanjkanje grafa scene, ki pri nekoliko zahtevnejših 3D projektih postane skoraj nepogrešljivo orodje. Zato ga je v takih primerih skoraj nujno kombinirati še s kakšnimi drugimi orodji, kot je na primer OSG (Open Scene Graph).

3.4.2 GamePlay3D

GamePlay [21] je odprtokodni visokonivojski pogon za igre, zgrajen na osnovi OpenGL. Fizikalnega pogona sicer nima vgrajenega, vsebuje pa že izdelane vmesnike za integracijo z zmogljivim Bullet Physics [20]. Je kompakten in mogoče v nekaterih pogledih celo preveč poenostavljen. Zgrajen je v jeziku C++, s pomočjo katerega je tudi najlaže dostopati do njega, podpira pa vse modernejše operacijske sisteme. Z določenimi omejitvami tudi mobilne.

Dokumentiran je sicer dobro, ga pa v primerjavi z SDL, OGRE ali Unity uporablja nekoliko manjša skupina ljudi, kar se hitro opazi pri količini uporabnih nasvetov in dobrih zgledov na medmrežnih forumih.

Kljub temu, da je eleganten, čist in enostaven za uporabo ter s tem omogoča hitrejši razvoj in uporabo običajnih prvin, ki jih pri razvoju iger pogosto srečamo, pa mu določene podrobnosti enostavno manjkajo. Na primer, ne podpira enostavnega načina, s katerim bi lahko dosegli učinek deljenega zaslona (angl. *split screen*), ki ga v našem projektu potrebujemo. Na podobne težave naletimo tudi pri nekaterih drugih nastavitvah, kot je na primer izbira zaslona za prikaz, ki brez poseganja v globino in neposrednega dela z OpenGL enostavno niso na voljo.

3.4.3 OGRE

Object-Oriented Graphics Rendering Engine, ali s kratico OGRE [30], je vsestranski in zelo zmogljiv odprtokodni grafični pogon. Omogoča izbiro uporabe DirectX ali OpenGL tik pred zagonom same aplikacije. Knjižnici povsem ovije in programerju ponudi v uporabo svoje funkcije in objekte, tako da se mu s specifikami ene ali druge ni treba ukvarjati. Teče na vseh modernejših operacijskih sistemih, v nekoliko okrnjeni različici pa celo na iOS in Android mobilnih napravah.

Dokumentacija zanj je kvalitetna in obširna. Poleg običajnih referenc in pomoči za posamezne podatkovne strukture in funkcije, na uradni spletni strani [30] obstaja tudi tečaj (angl. *tutorial*) z več nivoji, s pomočjo katerega se lahko nov uporabnik hitro nauči osnov. Ko te osvoji, lahko nadaljuje še z določenimi naprednimi napotki in triki, ki so opisani v nadaljevalnem razdelku.

Glede na to, da gre za enega najbolj razširjenih odprtokodnih grafičnih pogonov, je temu primerno tudi veliko število že izvedenih uporabnikov. Zato je skoraj na vsako splošno vprašanje, ki se pojavi ob programiranju z relativno malo truda mogoče dobiti vsaj dober odgovor ali namig, pogosto pa celo primer kode, ki prikazuje pravilno (ali vsaj delujočo) rešitev.

Po drugi strani pa je prav zaradi svoje širine in širokega spektra funkcionalnosti tudi dokaj velik. Sicer je razdeljen na ločene module, ki jih lahko izpustimo (s tem se izognemo uporabi nekaterih DLL knjižnic v končnem programu), ampak tudi v svoji najbolj osiromašeni obliki po velikosti preseže tako SDL, kot GamePlay3D. Poleg tega je moteče še, da se že osnovno okolje na nekoliko manj zmogljivem računalniku za povezovanje pri gradnji projekta in za nalaganje pri zagonu programa porabi kar znatno količino časa. Če uporabljamo določene naprednejše funkcije, pa se ta čas še dodatno podaljša.

Poleg samega nabora orodij in pripomočkov za delo z grafiko vsebuje še razne elemente za gradnjo menijev, pa tudi drugih pripomočkov, ki nam pomagajo pri interakciji z uporabnikom.

3.4.4 Unity 3D

Unity [44] je edini pravi pogon za igre, ki smo si ga ogledali. Za razliko od ostalih treh ni odprtokodne narave, je pa v nekoliko okrnjeni različici dostopen brezplačno. Deluje na vseh modernih operacijskih sistemih, kot tudi na mobilnih platformah iOS in Android ter temelji na OpenGL.

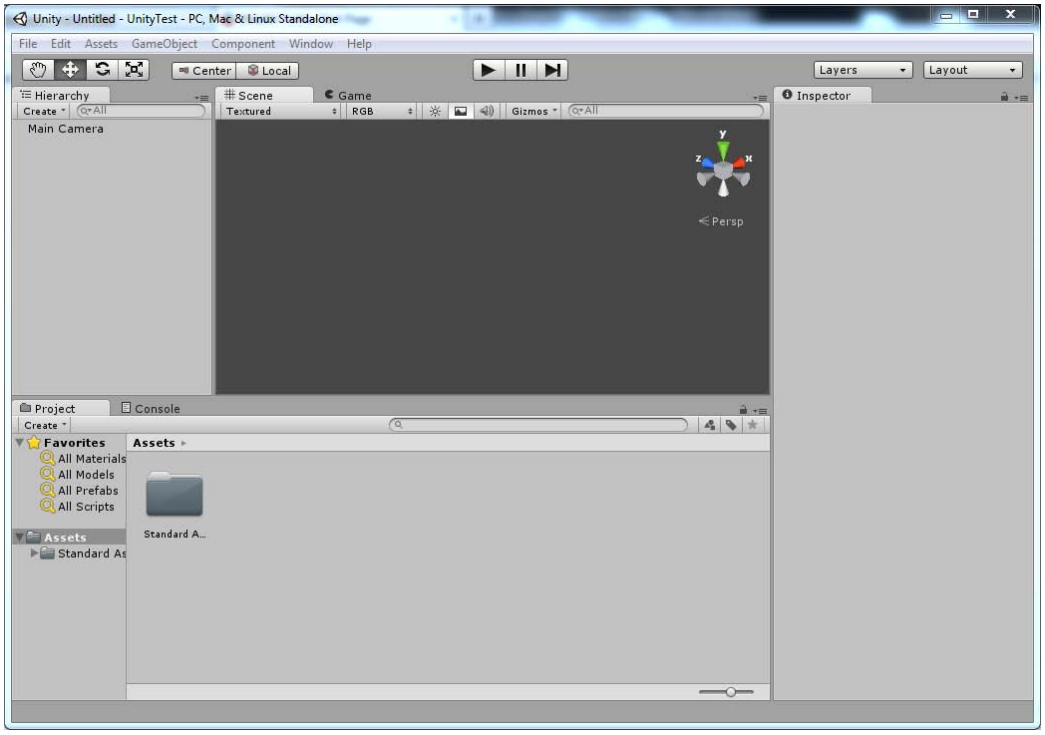
Funkcionalnosti programskega vmesnika segajo daleč preko grafičnih elementov. Poleg dela z zvokom, nadzora nad interakcijo z uporabnikom, gradnjo menijev, fizikalnim pogonom in podobnimi dodatki, vsebujejo orodja na še višjem nivoju, kot so denimo skripte za določanje obnašanja interaktivnih predmetov s sprožilci, detekcijo bližine, stikal in podobnih kriterijev, s pomočjo katerih lahko tipično igralec v igri vpliva na svoje okolje, ali pa okolje vpliva nanj.

Poleg programskega vmesnika pa Unity ponuja še zmogljiv grafični uporabniški vmesnik (prikazan na sliki 3.5), ki je predvsem namenjen vnaprejšnji gradnji scen in nivojev, v katere bo med igro igralec postavljen, ter določanju njihovega obnašanja s pomočjo skript. Pravzaprav v skrajnejših primerih lahko samo s pomočjo klikanja z miško in pisanjem skript ustvarimo celo aplikacijo (igro).

Programsko ogrodje je temu primerno zapleteno in obsežno. Čeprav je mogoče uporabljati samo grafični vidik, je to vseeno le del celotnega paketa in pogosto naletimo na nevšečnosti, kjer moramo v svoj program zaradi reševanja zelo specifičnega in relativno preprostega problema vstaviti velike module s kupom funkcionalnosti, ki jih sploh ne nameravamo uporabiti.

Temu primerno dolg je tudi čas, ki ga vsakič posebej potrebujemo za gradnjo in zagon projekta, ter število in velikost dodatnih knjižnic.

Dokumentacija je dovolj dobra. Prav tako lahko zaradi velikega števila uporabnikov najdemo celo vrsto poučnih tečajev – od začetniških do takih, ki dajejo poudarek na spopadanje s specifičnimi in zahtevnejšimi izzivi. Tudi primerov kode je dovolj.



Slika 3.5: Uporabniški vmesik okolja Unity3D

3.4.5 Izbira

Ogledali smo si še nekaj drugih pogonov, ki pa večinoma že na prvi pogled niso bili ustrezni. Bodisi zaradi izrazite usmerjenosti na področje izdelave iger in poudarka na klasični interaktivnosti, bodisi zaradi prevelike preprostosti ali celo izključne uporabe 2D grafike.

Na koncu smo se odločili za OGRE. Po eni strani je dovolj zmogljiv, da lahko z njegovo uporabo enostavno dosežemo vse zastavljene cilje, po drugi pa ne vsebuje veliko nepotrebnih delov, ki se jim ne bi mogli izogniti in bi povzročili slabšo preglednost. Obenem je dobro dokumentiran, z veliko primeri specifičnih rešitev. Opogumilo nas je tudi dejstvo, da so OGRE že večkrat uporabljali v raziskovalno vizualizacijske namene in je očitno za tako rabo primeren.

Kot primer lahko podamo delo I. Milna in G. Rowa z univerze v Dundeeju,

ki sta s pomočjo knjižnice OGRE vizualno predstavila postopek strukture in teka objektne programske kode [15]. Delo je namenjeno začetnikom, ki se programiranja (oziroma objektnega programiranja) šele učijo in jim pomaga pri predstavi napisane kode. Seznam primerov uporabe lahko najdemo na uradni OGRE-ovi spletni strani [41].

3.5 MS Kinect

Microsoft Kinect[10] je naprava, ki poleg zvoka in slike (kar zmore vsaka običajna spletna kamera) zaznava še globinsko sliko prostora pred seboj. Prvič je prišla na prodajne police v novembru 2010, kot naprava za izvedbo naravnega vmesnika za igranje iger na konzoli Xbox 360. Že takoj ob izidu je bila izjemno uspešna, saj je v prvih dveh mesecih dosegla kar osem milijonov prodanih enot.

V juniju 2011 so pri Microsoftu zanjo izdali prvo verzijo gonilnikov za Windows, zbirko orodij za razvoj aplikacij (angl. *Software Development Kit* ali s kratico *SDK*) in adapter, ki vtikač, namenjen specifično Xboxu 360, pretvori v kombinacijo standardnega USB-vtikača in transformatorja za dodatno napajanje. S tem so omogočili, da so lahko zainteresirani razvijalci Kinect priklopili na osebni računalnik in začeli razvijati svoje aplikacije, ki so lahko izkoriščale njegove zmožnosti. V februarju 2012 so izdali še posebno različico Kinecta, specifično namenjeno uporabi na osebnih računalnikih z operacijskim sistemom Windows. S programskega vidika sta oba sistema povsem ekvivalentna. Glavne spremembe pri novejši napravi so tako le v malo manjših dimenzijah senzorja, nekoliko zanesljivejšem zaznavanju na kratkih razdaljah (izboljšani ločljivosti) in drugačnih priključnih kablih.

Kasneje so pri Microsoftu izdali tudi programsko zbirko Kinect Toolbox, ki nadgradi osnovne funkcionalnosti SDK-ja z naprednejšimi algoritmi, ki olajšajo delo z zajetimi podatki. Od takrat so oboje že nekajkrat posodobili

[10]Kinect za Windows - uradna stran. Dostopno na: http://www.microsoft.com/en-us/kinectforwindows/

in nadgradili. Trenutna najnovejša različica orodjarn je 1.8, ki je na voljo od septembra 2013.

3.5.1 Delovanje strojne opreme

Senzor, ki je prikazan na shemi 3.6, sestoji iz naslednjih elektronskih komponent:

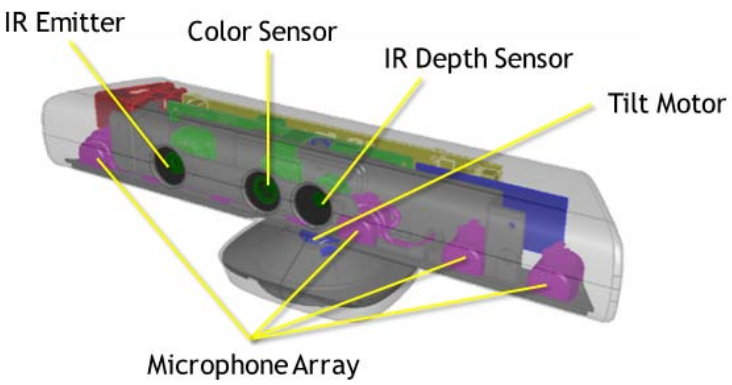
- infrardečega projektorja mreže točk,
- infrardeče kamere,
- barvne kamere,
- sistema štirih mikrofonov,
- motorja za nagib ohišja,
- senzorja nagiba.

V nadaljevanju si bomo ogledali, čemu so posamezne komponente namenjene in kako sistem z njihovo pomočjo lahko zaznava predmete v prostoru pred seboj.

Infrardeča projektor in kamera

IR projektor in kamera, ki sta označena na sliki 3.6, sta del istega podsistema, ki deluje na sledeč način:

Projektor z infrardečo svetlobo, ki jo človeško oko ne zaznava, v prostor pred sabo projicira mrežo točk v posebnem vzorcu (kot lahko vidimo na sliki 3.7), za katere si zapomni natančne kote, pod katerimi jih projicira. IR kamera, ki je nameščena vzporedno s projektorjem na natančno določeni razdalji, pa točke spet zazna pod vpadnim kotom, ki je odvisen od razdalje predmeta, na katerega je točka projicirana. Tako za vsako zaznano točko natančno določi razdaljo in ustvari globinsko sliko prostora pred seboj v



Slika 3.6: Shematski prikaz komponent senzorja MS Kinect [42].

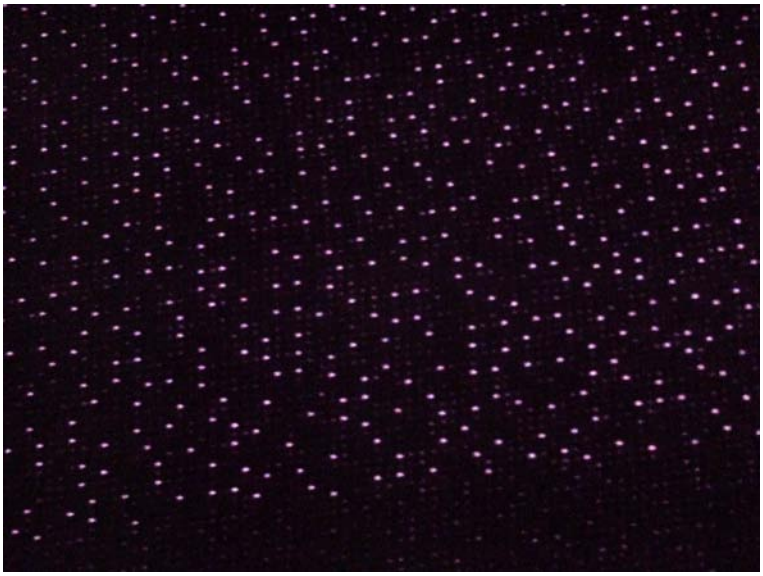
ločljivosti 640x480. Shematski prikaz triangulacije posamezne točke lahko vidimo na sliki 3.8.

Ta pristop je pri Kinectu edinstven in rešitev se je izkazala za dobro. Sistem je namreč zaradi lastnega vira infrardeče svetlobe v veliki meri neodvisen tako od osvetlitve okolice, kot tudi od barve in oblike ozadja. Specifično ta dva dejavnika pri glavnini drugih podobnih sistemov, ki se zanašajo zgolj na računalniški vid, predstavljata največ težav. Poleg tega je zaradi stalne in nepremične namestitve komponent v ohišje že vnaprej dobro umerjen in tako s strani uporabnika ne potrebuje nobenega posebnega postopka kalibracije.

Barvna kamera

Barvna kamera je povsem ekvivalentna običajni spletni kameri. Njena posebnost je v največji meri ta, da je poravnana s komponentama senzorja globine in tako točke na globinski sliki dobro sovpadajo z barvnimi točkami, ki jih zajame.

To omogoča izvedbo raznih efektov, kot je na primer t.i. učinek zelenega zaslona (angl. *green screen*), ki vse točke slike, ki predstavljajo ozadje, v realnem času nadomesti z drugačno vsebino. Tako na prikazani sliki lahko opazovalec dobi vtis, kot da se uporabnik v resnici nahaja nekje drugje, kot je prikazano na sliki 3.9. Poleg tega je barvna kamera lahko uporabna kot čisto



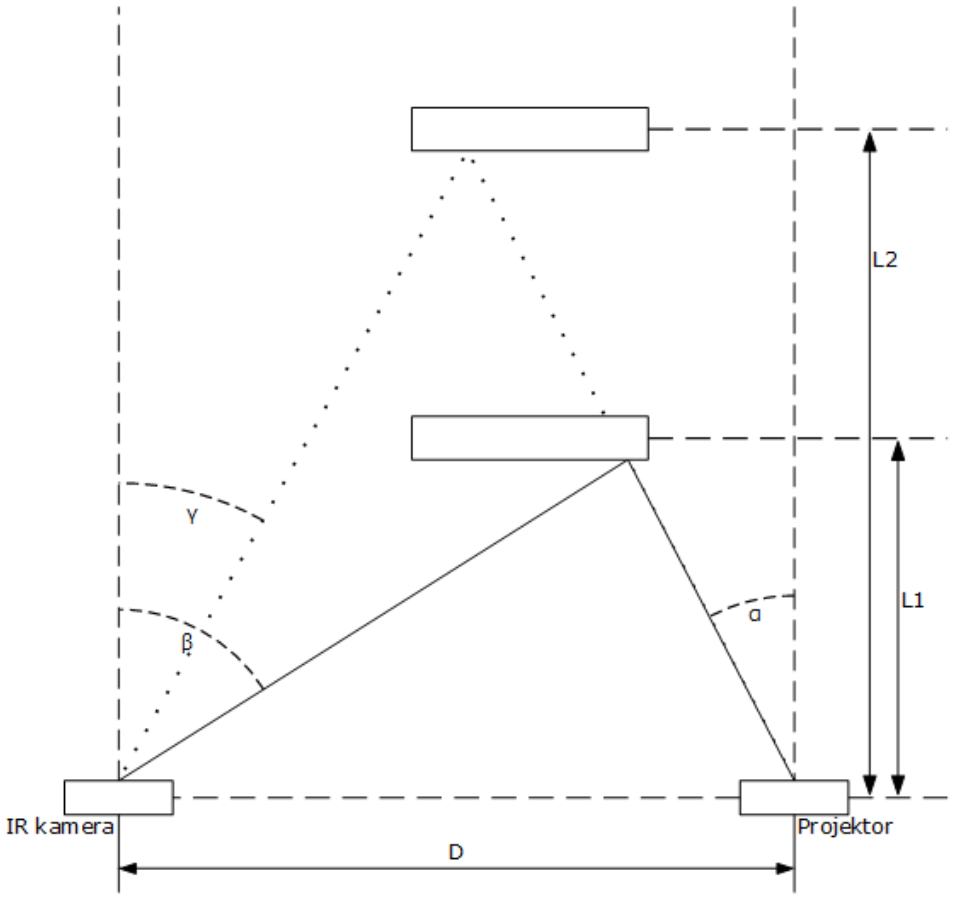
Slika 3.7: Točke, ki jih Kinect v IR svetlobi projicira v prostor pred seboj (v tem primeru na ravno steno [37].

običajna spletna kamera za videokonference, izdelavo posnetkov in podobna vsakdanja opravila.

Sistem mikrofonov

Štirje mikrofoni so razporejeni po celi širini ohišja (prikazano na shemi 3.6). Poleg zajema zvoka v pravem stereo načinu, to omogoča tudi določanje smeri vira zvoka v prostoru pred Kinectom. S pomočjo razlik v amplitudi sprejetega valovanja in predpostavke, da se govorec nahaja pred (in ne za) napravo, je mogoče enolično izračunati, iz katere smeri glas prihaja.

Ker programska orodja podpirajo tudi glasovne ukaze in razpoznavanje govora, je ugotavljanje smeri zvoka pomembno predvsem s stališča, Kinect ob zaznavanju več kot enega uporabnika ugotovi, kateri trenutno govori in reagira temu primerno.



Slika 3.8: IR projektor in IR kamera. Shema prikazuje isto projicirano točko, ki glede na različno oddaljenost predmeta pade v kamero pod različnim kotom.

Motor za nagib ohišja in senzor nagiba

Motorček v podstavku (prikazan na shemi 3.6) je mogoče programsko krmiliti in tako prilagajati nagib naprave v smeri gor-dol. S prilagajanjem nagiba lahko celotno napravo usmerimo tako, da gleda bolj navzgor, če je postavljena na nižjo podlago, ali pa bolj navzdol, če je postavljena kje višje. Senzor nagiba je neodvisen od motorčka in nagib zaznava s pomočjo merilca pospeška, ki meri smer sile teže. Mehanizem, ki lahko napravo nagne za največ +-27°, je namenjen zgolj občasnim popravkom kota in ne stalni uporabi [40].

Slika 3.9: Demonstracijski program "Green Screen", ki uporabi Kinect za postavitev osebe pred navidezno ozadje.

3.5.2 Programska oprema

Poleg uradnega nabora gonilnikov, SDK-ja in Toolkita, ki jih razvija in v rednih intervalih posodablja Microsoft, je na voljo še nekaj odprtih neuradnih gonilnikov in orodij, ki pa so pogosto nekoliko zastarela in ne podpirajo tako širokega spektra funkcionalnosti.

Glede na to, da uradna izdaja izmed vseh operacijskih sistemov podpira samo Windows, obstaja precejšnje število razvijalcev predvsem v okolju Linux, ki druge možnosti kot da se zatečejo k neuradnim izdajam sploh nimajo. Dve popularnejši orodjarni sta OpenKinect/LibFreeNect in OpenNI.

OpenKinect/LibFreeNect [33] v celoti uporablja povsem svoje rešitve in je tako na vseh nivojih popolnoma neodvisen od Microsoftove podpore. Tako omogoča delo s Kinectom tudi na operacijskih sistemih Linux in OSX. Težava je, da ne podpira skoraj nobenih naprednejših funkcij. Omogoča

namreč neposredno pridobivanje podatkov s senzorja, kamere in mikrofonov, ter njihovo osnovno obdelavo (globinska slika), nima pa podpore za nadaljnjo obdelavo teh podatkov, kot je iskanja skeleta uporabnika, razbiranje gest (npr. odprtih, zaprtih rok, odrivov) in lociranje kota izvora zvoka na podlagi mikrofonov. Poleg tega projekt v zadnjem času ni aktiven in od lanskega leta ni doživel nobene posodobitve.

OpenNI [34] po drugi strani podpira skoraj vse funkcionalnosti uradnega programskega paketa, poleg tega pa zanj obstaja tudi cela vrsta dodatkov, ki določene vidike še dodatno razširijo. Poleg Kinecta podpira tudi druge prostorske senzorje na operacijskih sistemih Windows, Linux in OSX. Glavna težava je, da od verzije 2.0 naprej pri uporabi Kinceta zahteva uradne Microsoftove gonilnike in SDK, ki pa delujejo samo v okolju Windows. Tako težave uporabnosti na drugih operacijskih sistemih ne odpravlja, ampak odgovornost zanjo zgolj prelaga na Microsoft.

To pomeni, da je uporaba operacijskega sistema Windows in Microsoftovih razvojnih orodij v resnici skoraj nujna za učinkovito rabo Kinecta in vseh njegovih zmogljivosti. V nadaljevanju si bomo ogledali, kaj nam kot razvijalcem ponujata uradna Microsoftova SDK in Toolkit.

Microsoft Kinect SDK 1.8

SDK (Software Development Kit) [28] za Kinect je zbirka programskih pripomočkov, ki nam omogoča učinkovit dostop do podatkov, ki jih naprava generira, vsebuje pa tudi večje število orodij in funkcij, s katerimi lahko podatke obdelamo in iz njih izluščimo uporabne informacije. Uporabljamo ga lahko preko programskih jezikov C/C++ ali C#. V zadnji izdaji (1.8) pa so dodali še podporo za dostop do vmesnika z jezikom HTML5. Funkcije v njem služijo v glavnem trem opravilom:

- inicializaciji,
- prejemanju podatkov,
- obdelavi podatkov.

Kjer je treba dodati, da sta prejemanje in obdelava podatkov pogosto precej tesno povezana.

Na začetku moramo poklicati nekaj funkcij, da vklopimo senzor in zaženemo cikel zajemanja in procesiranja podatkov na Kinectu, pri čemer z uporabo kombinacije zastavic določimo, katere dele vmesnika želimo uporabljati. Če na primer želimo globinski senzor s prepoznavanjem igralcev in zajem zvoka, dvignemo zastavici: `NUI_INITIALIZE_FLAG_USES_DEPTH_AND_PLAYER_INDEX` in `NUI_INITIALIZE_FLAG_USES_AUDIO`. Podoben pristop se uporablja tudi pri večini ostalih funkcij. V naslednji fazi inicializacije same tokove zaženemo, kar zahteva še nekaj podrobnih nastavitev, potem pa lahko kadarkoli zahtevamo trenutno stanje senzorja. Tokovi v večini primerov nosijo že nekoliko obdelane podatke. Na primer – globinsko sliko nam izračuna že ogrodje v ozadju in se nam ni potrebno ukvarjati s triangulacijo in računanjem oddaljenosti. Prav tako s "poslušanjem" toka skeleta igralcev lahko neposredno dobimo kartezične koordinate njihovih posameznih sklepov v metrih glede na oddaljenost od senzorja (ki ima koordinate [0, 0, 0], globinska os pa poteka v pravokotni smeri v prostor pred njim).

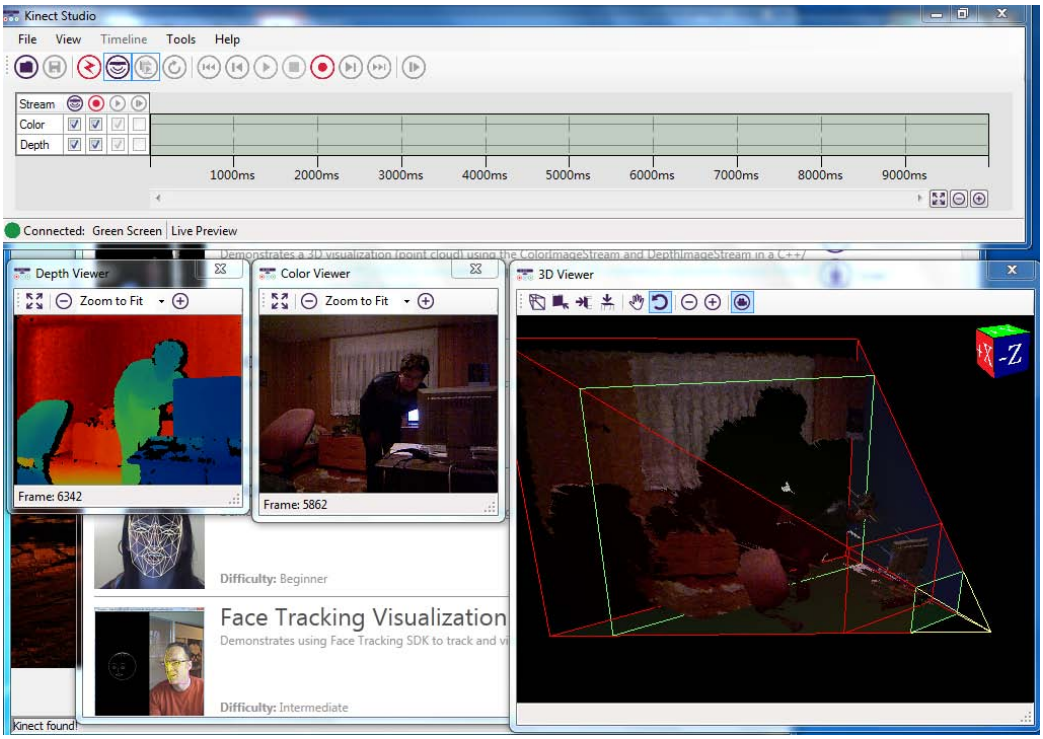
Zajem podatkov tako tipično poteka v zanki, v kateri v vsakem obhodu najprej zahtevamo nov okvir podatkov z želenega toka (recimo toka globinskih slik ali toka skeletnih koordinat igralcev), potem pa ga posredujemo delu programa, ki ga potrebuje.

Microsoft Kinect Toolkit 1.8

Posebno orodjarno dodatnih funkcionalnosti, ki vsebuje tudi zbirko primerov uporabe z dodano izvorno kodo, so pri Microsoftu prvič izdali v maju 2012 ob verziji SDK 1.5, torej približno ob istem času, ko je izšla tudi različica Kinecta za Windows. Za uporabo Toolkita seveda potrebujemo tudi ustrezno verzijo SDK-ja.

Verzija 1.5 je vsebovala le dodatke, ki so poleg običajne rabe omogočali sledenje obraznih potez in že omenjene primere z izvorno kodo za razvijalce. Vključevala je tudi prvo verzijo posebnega programa (Kinect Studio [27],

prikazan na sliki 3.10), ki je namenjen snemanju in pregledovanju podatkov v



Slika 3.10: Kinect Studio z globinsko sliko, barvno sliko in kombiniranim 3D pogledom.

realnem času, ki jih Kinect trenutno zajema in pošilja določeni aplikaciji. To orodje je uporabno predvsem z vidika razhroščevanja.

Verzija 1.6 razen nekaterih popravkov in posodobitev ni prinesla občutnih novosti, z verzijo 1.7 v marcu 2013 pa so dodali še dva pomembna dodatka: t.i. Fusion in podporo interakcijam.

Fusion, ki je namenjen predvsem skeniranju in modeliranju predmetov ali prostora, ki ga Kinect zaznava. Z njegovo pomočjo lahko tako s precej veliko natančnostjo izdelamo 3D modele resničnih predmetov z njihovo barvo vred. Postopek je robusten in daje dobre rezultate tudi na zapletenih predmetih, kot so recimo ljudje ali – bolj specifično – njihovi obrazi.

Podpora interakcijam pa omogoča predvsem zaznavo odprtih ali zaprtih

dlani ter določene geste, kot je recimo odriv/potisk (angl. *push*). Namenjena je predvsem navigaciji po raznih 2D površinah, kot so meniji ali zemljevidi in omogoča enostavnejšo in učinkovitejšo implementacijo nadomestka za miško. Na primer, pred verzijo 1.7 je uporabnik gumb v meniju pritisnil tako, da je na njem samo dovolj dolgo držal virtualno kazalko, ki je sledila njegovi roki. Zdaj pa lahko kazalko enostavno pripelje nad gumb in ga precej bolj naravno "potisne" v globino. Poleg tega tudi detekcija zaprtih dlani omogoča veliko različnih in predvsem bolj intuitivnih načinov za implementacijo interakcij.

Verzija 1.8 je spet prinesla samo določene nadgradnje ter popravke že obstoječih funkcionalnosti in primerov.

3.6 Vuzix Wrap 920AR

Očala za navidezno in obogateno resničnost v zadnjem času doživljajo zelo hiter razvoj. Tehnologija tankih in majhnih zaslonov z visoko resolucijo, digitalnih videokamer in senzorjev premikanja se je v zadnjem času občutno izboljšala, obenem pa tudi zelo pocenila. Tako so očala za navidezno in obogateno resničnost postala v vedno bogatejši paleti dostopna širši javnosti z rešitvami, kot so Oculus Rift[11], Meta[12] ter izdelki podjetij Virtual Realities[13] in Vuzix[14].

Pojma navidezna resničnost in obogatena resničnost sta si med sabo podobna, pa vendar obstaja med njima očitna razlika. Pri virtualni resničnosti je namreč slika, ki jo (tipično s pomočjo očal) predvajamo uporabniku v celoti računalniško generirana, pri obogateni resničnosti gre pa za dodajanje navideznih elementov realnemu svetu. To praviloma dosežemo tako, da s pomočjo dveh umerjenih kamer zajemamo sliko ozadja, s pomočjo računalniške grafike pred to ozadje dodamo še navidezne objekte. Obstaja pa tudi druga pot, pri kateri na prosojno steklo očal, ki omogoča pogled skoznje, prikazujemo

[11] Oculus Rift - uradna stran. Dostopno na: http://www.oculusvr.com/
[12] Meta VR - uradna stran. Dostopno na: http://www.metavr.com/
[13] Virtual Realities - uradna stran. Dostopno na: http://www.vrealities.com/
[14] Vuzix - uradna stran. Dostopno na: http://www.vuzix.com/home/

dodatno sliko, ki tako na videz postane del resničnosti.

Za razvoj naše aplikacije so nam bila na voljo očala za obogateno resničnost Wrap 920AR podjetja Vuzix [45] (prikazana na sliki 3.11). Za

Slika 3.11: Očala za obogoateno resničnost Vuzix Wrap 920AR, v ospredjiu lahko vidimo obe kameri za zajem stereo slike.

njihovo učinkovito delovanje so ključni trije deli, ki jih bomo podrobneje predstavili v nadaljevanju:

- Dva zaslončka (po eden za vsako oko) na notranji strani očal.
- Dve kameri (po ena za vsako oko) na zunanji strani očal.
- Senzor nagiba očal.

Poleg teh komponent očala omogočajo tudi predvajanje zvoka s pomočjo vgrajenega sistema slušalk, ampak ta vidik za ustvarjanje vtisa obogatene resničnosti ni ključen, niti ga ne uporabljamo v rešitvi.

Z vidika programske podpore smo v zvezi z delovanjem očal uporabili dve orodjarni. Vuzix SDK, ki predvsem omogoča kalibracijo in branje podatkov s

senzorjev o orientaciji očal, ter OpenCV, ki močno olajša in pospleši zajem slike s kamer.

3.6.1 Strojna oprema

Par prikazovalnih zaslončkov

Na notranji strani očal se pred vsakim očesom nahaja po en LCD zaslonček v ločljivosti 800x600 (vidno na sliki 3.12). Dostopna sta preko klasičnega

Slika 3.12: Vuzix Wrap 920AR - pogled očal z notranje strani. Na sliki vidimo leči za oba zaslončka, dve 2,5mm vtičnici za slušalki, na desni strani pa tudi senzor za ugotavljanje rotacije očal.

VGA priključka, preko katerega ju računalnik zazna in obravnava kot običajen zaslon. V osnovnem načinu delovanja oba prikazujeta enako sliko (kloniranje). S pomočjo kontrolne ploščice (prikazana na sliki 3.13) ali programskega vmesnika pa lahko delovanje spremenimo tudi tako, da vsak prikazujeta svojo polovico slike (kar sicer pomeni, da se njuna ločljivost v vodoravni smeri zmanjša na polovico). Na ta način lahko zelo enostavno na levem

Slika 3.13: Kontrolna ploščica za AR očala – omogoča nastavitve zaslončkov in načina steroskopskega predvajanja.

očesu prikažemo drugačno sliko kot na desnem, kar je nujno za učinkovito ustvarjanje vtisa globine.

Obstajajo tudi druge opcije, s katerimi lahko različno sliko na obeh zaslonih dosežemo s tem, da na levem zaslonu prikazujemo vse lihe sličice, na desnem pa vse sode, ampak ta način ni programsko podprt in ga je v uporabniških aplikacijah zelo težko pravilno uporabiti. Možno je uporabiti tudi klasični način prikaza različnih barv na posameznem zaslončku, kar pa neizogibno nekoliko popači barve.

Par kamer

Na sprednji strani očal se nahajata dve vzporedni kameri, po ena pred vsakim očesom. Razdalja med njunima središčema znaša 60mm, njun zorni kot pa je razmeroma ozek in v navpični smeri znaša 30°.

Obe skupaj na računalnik priključimo s pomočjo enega USB-priključka, ki obenem skrbi tudi za napajanje obeh kamer in osvetlitve zaslonov. Računalnik ju po priklopu zazna kot povsem običajni USB-spletni kameri z maksimalno

ločljivostjo 640x480 in ju tako tudi obravnava. Za zajem slike torej ne potrebujemo nobenih specifičnih orodij.

Senzor nagiba očal

Lahko ga poimenujemo tudi senzor nagiba glave (angl. *head-tracker*). To je v resnici ločena napravica, ki je priklopljena na notranjo stran očal (vidna na sliki 3.13). Vsebuje senzorje magnetnega polja, žiroskope in merilce pospeška, ki omogočajo, da lahko v kateremkoli trenutku ugotovimo orientacijo očal v prostoru. Ta podatek je namreč nujen za pravilno in dinamično poravnavo virtualne slike z realnim ozadjem.

Vsi podatki s senzorjev so dostopni s pomočjo programskega vmesnika Vuzix SDK.

3.6.2 Programska oprema

Vuzix SDK

Programski vmesnik Vuzix SDK [46] vsebuje v glavnem funkcije za delo s senzorjem. Torej za njegovo inicializacijo, kalibracijo in zajem trenutnih podatkov. Poleg njih obstaja še nekaj funkcij za nastavitev stereo načina in izpis podatkov o verziji strojne in programske opreme. Ogrodje ne podpira zajema slike s kamer, ki pravzaprav delujeta kot popolnoma neodvisni običajni spletni kameri.

Najpomembnejša funkcija je `IWRGetTracking`, ki nam vrne podatke o trenutni orientaciji očal, kadar jo pokličemo. Podatke zapiše v tri spremenljivke, ki določajo rotacijo okrog posamezne osi in jih lahko enostavno uporabimo za nadaljnje procesiranje.

OpenCV

OpenCV [32] je ena najbolj znanih in razširjenih odprtokodnih knjižnic za rabo v zvezi z računalniškim vidom. Poleg Windowsov podpira namreč še

Linux in Mac OSX, v določeni malo okrnjeni različici pa tudi mobilne sisteme, kot sta Android in IOS.

Vsebuje celo vrsto orodij za zajem videa, obdelavo posameznih slik, filtriranje, iskanje robov, značilk in razlik ter še številnih drugih opravil v povezavi z računalniškim vidom.

V našem primeru smo jo potrebovali izključno za zajem slik s kamer, kar je mogoče z njeno pomočjo doseči na učinkovit način. Zajem tako ni odvisen od operacijskega sistema, kot bi bil, če bi za ta namen na primer neposredno uporabili Windowsovo knjižnico DirectShow, ki je poleg tega veliko zapletenejša za uporabo.

Poglavje 4

Rešitev

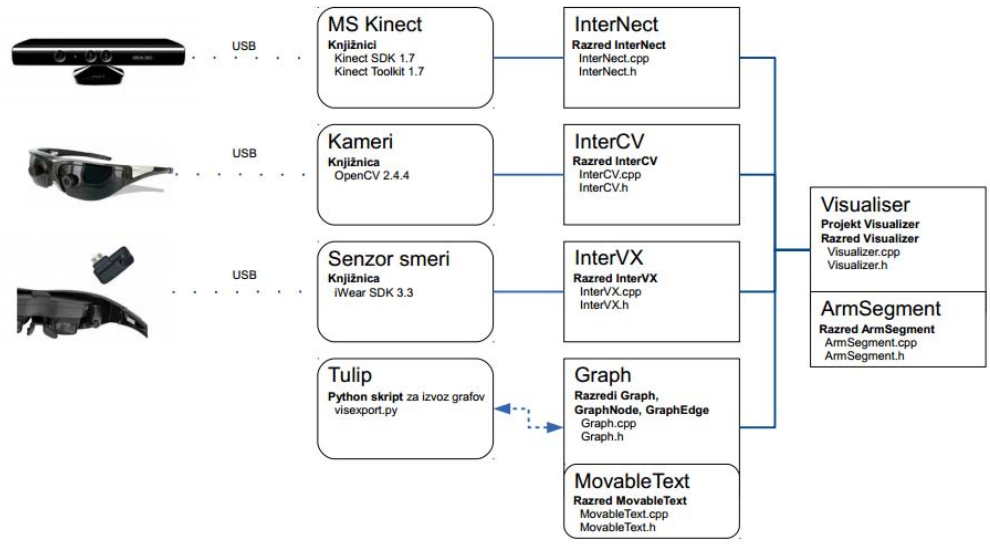
Na podlagi raziskanih in izbranih tehnologij ter zastavljenih ciljev smo sestavili rešitev. Med izdelavo smo ves čas stremeli tudi k čim večji modularnosti in neodvisnosti posameznih delov aplikacije med sabo. S tem smo dosegli, da bi ob posodobitvi knjižnic, zamenjavi določenih strojnih komponent, prestavitvi rešitve na drug operacijski sistem ali nadaljevanju in nadgrajevanju obstoječega dela lahko zamenjali samo posamezne kose programske kode, kjer bi bile spremembe potrebne, preostanek pa bi lahko ostal nespremenjen.

Rešitvi smo nadeli ime "Visualizer" in je sestavljena iz naslednjih glavnih komponent:

- temeljne grafična aplikacija v ogrodju OGRE,
- modula za zajem podatkov s Kinecta (InterNect),
- modula za zajem slike s kamer (InterCV),
- modula za zajem podatkov z Vuzix senzorja za orientacijo očal (InterVX),
- modula za uvoz in prikaz grafov ter njihove oblike.

Poleg tega smo razvili tudi skripto v Pythonu, ki preko Tulipovega grafičnega okolja trenutni graf izvozi v obliko, primerno za uvoz in prikazova-

nje v našem programu. Shemo zgradbe celotne rešitve si lahko ogledamo na sliki 4.1.



Slika 4.1: Shema komponent projekta, datotek, uporabe knjižnic in strojne opreme.

Razvili in testirali smo jo osebnem računalniku v naslednji konfiguraciji:

- Procesor: AMD Athlon 64 X2 3800+ (takt 2,01GHz)
- Grafična kartica: ATI Radeon HD 4870 (grafični pomnilnik 1GB)
- Pomnilnik: 2GB DDR2
- Kapaciteta diska: 80GB
- Operacijski sistem: Windows 7 (32bit različica)

Poleg tega smo zaradi težav z napajanjem USB-vtičnic dodali še razširitveno USB-kartico s petimi dodatnimi priključki.

Ker rešitev združuje in uporablja tudi večje število zunanjih knjižnic (OGRE, Kinect SDK, Kinect Toolbox, Vuzix SDK in OpenCV), te pa so

odvisne še od nekaterih drugih, ki jih namestimo kar z njimi vred v kompletu, postopek prevajanja projekta traja kar precej časa.

Pospešimo ga lahko najlaže z uporabo t.i. vnaprej prevedenih zaglavnih datotek (angl. *precompiled headers*), kar smo uporabili tudi v našem projektu. S to funkcijo nam prevajalnik omogoča, da določene datoteke z glavami (datoteke s končnico .h) in njihove odvisnosti, ki jih je v večjih knjižnicah, kot sta OGRE in OpenCV, zares ogromno, ni potrebno prevajati s preostankom projekta vsakič znova, ampak samo ob morebitnih spremembah. Način njihove uporabe bomo na kratko pojasnili v poglavju 4.1.1.

V nadaljevanju si bomo najprej ogledali nekaj podrobnosti glede uporabe Visual Studia 2010 ter prevajanju in uporabi knjižnic v povezavi z njim, potem pa se bomo posvetili posameznim modulom našega programa, pojasnili njihovo delovanje in pri vsakem posebej tudi izpostavili pomembnejše dele kode, ki so bili ključnega pomena za doseganje zastavljenih ciljev.

4.1 Uporaba MS Visual Studia

Visual C++ 2010 Express je izrazito projektno naravnano okolje, kar je za razvoj večjih in nekoliko zapletenejših aplikacij, kot je naša zelo smiselno, saj nas sili k določeni stopnji urejenosti.

Razvoj rešitve v Visual Studiu 2010 poteka v naslednjih korakih:

1. Ustvarimo novo rešitev (angl. *solution*).
2. Ustvarimo nov projekt znotraj rešitve.
3. Ustvarimo ali dodamo datoteke z izvorno kodo.
4. Konfiguriramo uporabo vnaprej prevednih zaglavnih datotek.
5. Namestimo in uvozimo knjižnice.

Zadnje tri točke si ne sledijo nujno v strogem vrstnem redu, ampak se pogosto odvijajo hkrati. Knjižnice tipično dodajamo takrat, ko začnemo

razvijati del rešitve, kjer bomo določene dele potrebovali, na konfiguracijo vnaprej prevedenih zaglavnih datotek pa moramo paziti sproti, ko v projekt dodajamo nove knjižnice in datoteke z izvorno kodo.

4.1.1 Vnaprej prevedene zaglavne datoteke

Vnaprej prevedene zaglavne datoteke (angl. *precompiled headers*), so mehanizem, ki nam omogoča, da v fazi prevajanja določene dele izvorne kode ali zaglavnih datotek prevedemo enkrat, nato pa jih ob naslednjih prevodih samo uporabimo in tako računalniku prihranimo zamudno delo. Na delovanje končne rešitve postopek nima nikakršnega vpliva, nam kot razvijalcu pa lahko prihrani veliko količino časa čakanja ob prevajanju, saj v primeru, ko uporabljamo večje število (velikih) knjižnic, postopek prevajanja njihovih zaglavnh datotek lahko traja občutno dlje, kot pa prevajanje celotne izvorne kode projekta samega.

4.1.2 Uporaba zunanjih knjižnic

V naši rešitvi v veliki meri uporabljamo zunanje knjižnice. Pri uporabi moramo biti pozorni predvsem na dva vidika. To sta prevajanje knjižnice za določeno okolje in arhitekturo ter uporabo knjižnice v končnem projektu. Prvemu se lahko pogosto izognemo, saj so v veliki večini primerov tudi odprtokodne knjižnice za tako običajno kombinacijo, kot je Visual C++ 2010 na operacijskem sistemu Windows, že prevedene, drugi pa v vsakem primeru pride v poštev.

Prevajanje knjižnic iz njihove izvorne kode ponavadi poteka v dveh korakih. Najprej izvorno kodo (z ustreznimi "makefile" datotekami) vstavimo v program CMake[1], ki na njihovi podlagi zna ustvariti projektne datoteke za izbrani prevajalnik (v našem primeru Visual Studio 2010). Nato nam ostane le še, da te projektne datoteke odpremo in sprožimo proces prevajanja. Rezultat so knjižnice za statično povezovanje (s končnico .lib, tipično v mapi

[1] CMake – uradna stran. Dostopno na: http://www.cmake.org/

"lib") in knjižnice za dinamično povezovanje (s končnico .dll v mapi "bin")

Uporaba knjižnic je enostavna. Poskrbeti moramo samo, da Visual Studio 2010 (ali drugo podobno okolje) ve, kje naj išče knjižnice za statično povezovanje, zaglavne datoteke in dodati mapo, kjer se nahajajo .dll datoteke na sistemsko pot (ali pa jih prenesti v isto mapo, kjer se nahaja naša glavna .exe datoteka).

4.2 Temeljna aplikacija OGRE

Osnova za naš projekt je standardna aplikacija v okolju OGRE. OGRE nam na voljo da precejšnjo količino abstraktnih razredov, ki jih lahko v svoji implementaciji podedujemo, jim izpolnimo potrebne metode, nato pa v funkciji `main` samo izdelamo nov objekt na podlagi tega razreda in poženemo določeno metodo (v našem primeru poimenovana `go`), ki nato poskrbi za zagon vseh potrebnih podsistemov in registrira našo aplikacijo/objekt v vseh želenih vlogah.

Temelj našega projekta se tako nahaja v datoteki "Visualizer.cpp", s pripadajočo zaglavno datoteko "Visualizer.h". Obe datoteki skupaj določata en sam razred, ki po dogovoru vsebuje metodo `go`, poleg tega pa deduje še celo vrsto abstraktnih razredov, ki v okviru okolja OGRE lahko opravljajo različne vloge in zato zahtevajo implementacijo specifičnih metod. V nadaljevanju si bomo najprej na kratko ogledali zaporedje ključnih opravil v funkciji `go`, nato pa še vloge podedovanih razredov in naloge, ki jih njihove metode opravljajo.

4.2.1 Metoda `go`

Metoda `bool Visualizer::go(void)`, ki jo poženemo v funkciji `main` takoj po tem, ko izdelamo glavni objekt aplikacije, opravi naslednje naloge v sledečem zaporedju:

1. Izdela objekt `mRoot` na podlagi razreda `Ogre::Root`. Ta objekt predstavlja najnižjo osnovno ogrodje OGRE aplikacije.

2. Iz konfiguracijske datoteke prebere nastavitve, ki jih uporabniku pred nadaljevanjem tudi predstavi v začetnem oknu za nastavitve in mu omogoči spremembe glede grafične konfiguracije (npr. pogon za upodabljanje, ločljivost, celozaslonski način, izbira naprave in zaslona).

3. Na podlagi prebranih nastavitev naloži dodatne vire in izdela okno, v katerem bo upodabljanje potekalo.

4. Izdela upravljalnik scene (angl. *scene manager*), ki omogoča enostavno delo z grafom scene in dodatnimi elementi, ki vplivajo na videz objektov v vidnem polju (na primer luči).

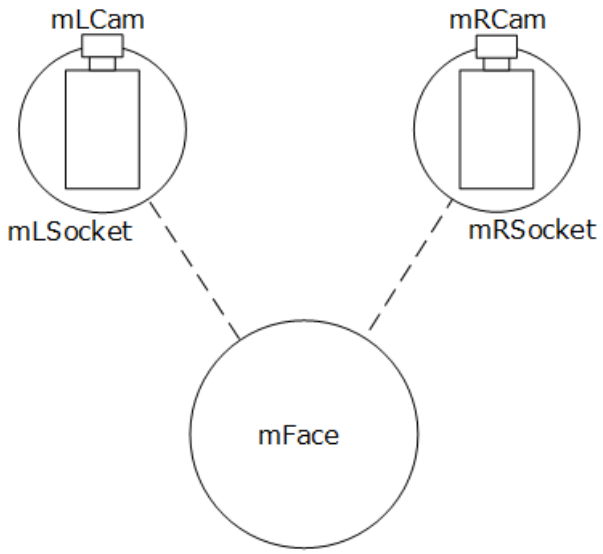
5. Temu sledi koda, specifična za naš projekt, ki izdela začetno sceno in zažene vse potrebne knjižnice.

6. Razred registriramo še v vseh vlogah, za katere smo ga z izvedbo abstraktnih metod usposobili.

7. S klicem metode `mRoot->startRendering()` zaženemo zanko za upodabljanje.

Z vidika pomembnosti za našo rešitev, se najzanimivejša koda nahaja v delu, ki ga označuje peta točka zgornjega seznama.

Ker uporabljamo stereoskopski vid, je potrebno tudi pogled v virtualno sceno izvesti z uporabo dveh kamer, ki pa morata biti med sabo ves čas povezani na enak način, kot sta pravi kameri na očalih. Zato smo izdelali del grafa scene s tremi vozlišči. Dve izmed njih z imeni `mSocketL` in `mSocketR` nosita vsako svojo kamero, obe pa sta v odnosu "otroka" povezani na vozlišče z imenom `mFace`, ki igra vlogo virtualne uporabnikove glave, čeprav nima svoje vizualne predstavitve. Ta zgradba je prikazana na diagramu 4.2. Njena glavna prednost je, da nam omogoča izvajati rotacijo in premikanje obeh kamer hkrati na naraven način tako, da vse operacije izvajamo samo nad vozliščem `mFace`, kameri pa se premikata z njim. Na tem mestu smo se morali tudi odločiti, kako bomo določili merske enote v virtualnem svetu v razmerju



Slika 4.2: Shema povezanih objektov za zajem slike v virtualnem svet.

do realnega. Določili smo, da je ena enota v virtualnem svetu enaka enemu centimetru v realnosti. Te odločitve smo se morali nato držati povsod v nadaljevanju.

Prav tako zaradi stereoskopskega vida na tem mestu zaženemo dve površini za izris (angl. *viewport*) eno ob drugi, ki ju nato povežemo vsako s svojo pripadajočo kamero. Pripravimo tudi pravokotni 2D plošči za izris ozadja, njuni teksturi, ki ju nato med izvajanjem dinamično posodabljamo in materiala, s pomočjo katerih teksturi dodelimo želenima površinama. Pripravimo in zaženemo tudi vse ostale objekte, ki jih uporabljamo za zajem podatkov s kamer, senzorja in Kinecta. Poleg tega pa še objekte za uvoz in izrisovanje grafa ter izris navideznih rok. Objekte namenjene izrisu na tem mestu razporedimo tudi v različne čakalne vrste, ki jih potrebujemo predvsem zaradi določanja obnašanja šablon (angl. *stencils*).

4.2.2 Izvedba abstraktnega razreda `Ogre::FrameListener`

Za dedovanje tega abstraktnega razreda smo morali implementirati eno samo metodo `virtual bool frameRenderingQueued(const Ogre::FrameEvent& Evt)`.

Metoda je zelo pomembna, saj jo ogrodje pokliče po enkrat za vsak cikel upodabljanja. V njej tako pred upodabljanjem posamezne sličice tako program opravi naslednja opravila:

1. Prebere podatke s senzorjev.
2. Posodobi teksturi ozadja glede na sliki z obeh kamer.
3. Na podlagi podatkov izvede transformacije ustreznih elementov na sceni.
4. Preveri, če se katera od rok dotika katerega od vozlišč in sproži izpis ustreznega besedila.
5. Skrije desno ozadje, prikaže levo ozadje in izriše sliko z leve kamere.
6. Skrije levo ozadje, prikaže desno ozadje in izriše sliko z desne kamere.

Tehnično je najzanimivejša tretja točka z zgornjega seznama. Vključuje namreč premike glave, premike rok in premike grafa glede na zaznane interakcije.

Lokacijo glave program zazna s pomočjo Kinecta. Glede na to, da smo si za koordinatno izhodišče izbrali kar položaj Kinecta, moramo pridobljene koordinate samo še pretvoriti v ustrezne enote. Rotacijo glave nam priskrbi modul za branje rotacijskih senzorjev na Vuzix očalih (InterVX).

Lokacijo in obnašanje rok, prav tako kot pri glavi, program zaznava s Kinectom. Natančnejši postopek zajema obojega je predstavljen v poglavju 4.4. Tukaj glede na zajete podatke popravimo lokaciji navideznih rok, nato pa glede na zaznane interakcije in razliko od prejšnjega stanja po potrebi transformiramo še graf.

4.2.3 Izvedba abstraktnega razreda `Ogre::RenderQueue Listener`

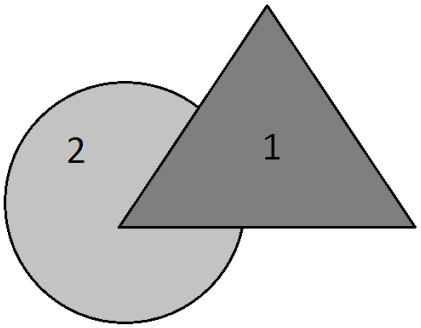
Da naš razred zadošča pogojem dedovanja mora implementirati štiri metode:

- `virtual void preRenderQueues();`
- `virtual void postRenderQueues();`
- `virtual void renderQueueStarted (uint8 queueGroupId, const String &invocation, bool &skipThisInvocation);`
- `virtual void renderQueueEnded (uint8 queueGroupId, const String &invocation, bool &repeatThisInvocation);`

Vse štiri metode določajo opravila, ki jih ogrodje pokliče glede na čakalne vrste za upodabljanje (angl. *render queues*). Z njihovo pomočjo lahko namreč vse vidne predmete na sceni razporedimo v eno od 100 čakalnih vrst (0 ima najvišjo prioriteto, 99 pa najnižjo) in jim tako v grobem določimo zaporedje, po katerem jih sistem izriše. To sicer nima neposredne povezave z globino (če je predmet št. 1 na sceni bližje od predmeta št. 2 tako, da ga deloma prekriva (prikazano na sliki 4.3), ga bo prekrival neodvisno od tega ali je razporejen v vrsto z nižjo ali višjo prioriteto od predmeta št. 2, samo tisti del predmeta št. 2, ki ni prekrit, se bo v enem primeru na površini za izris pojavil prej, v drugem pa kasneje).

Prvi dve metodi na zgornjem seznamu se tako izvedeta pred začetkom oziroma po koncu zaporednega izrisovalnega sprehoda po vrstah. Preostali dve pa pred začetkom oziroma po koncu obdelave vsake posamezne vrste. S prvim parametrom (`queueGroupId`) določita tudi, katera vrsta je pri tem klicu v obdelavi.

Predvsem z drugima dvema si močno pomagamo pri delu s šablonami, katere podrobneje predstavimo v poglavju 4.3. Na tem mestu namreč lahko določimo, kakšen vpliv na šablonski medpomnilnik (angl. *stencil buffer*) imajo izrisani predmeti določene skupine, kot tudi kakšen vpliv naj ima



Slika 4.3: Predmet 1 je izrisan pred predmetom 2 glede na globino, ne glede na zaporedje izrisovanja.

šablonski medpomnilnik na izrisovanje predmetov iz določene skupine. S tem pripomočkom v našem primeru na primer ustvarimo nekakšen učinek "zelenega zaslona" in določimo mesta, kjer želimo izrisati ozadje.

4.2.4 Izvedba ostalih abstraktnih razredov

Metode iz ostalih abstraktnih razredov so povezane predvsem z osnovnimi funkcijami za branje vhoda s tipkovnice in miške ter za hitro izvedbo uporabniškega vmesnika z enostavnimi okni in besedilnimi polji. Te metode smo tako uporabljali na povsem običajen način in se zato v njihove podrobnosti ne bomo spuščali.

4.3 Uporaba šablon

Pri upodabljanju scene poleg zaslonskega medpomnilnika (angl. *frame buffer* ali *color buffer*) sistem za upodabljanje uporablja še dva medpomnilnika.

Globinski medpomnilnik (angl. *depth buffer* ali *z-buffer*) nosi za vsako

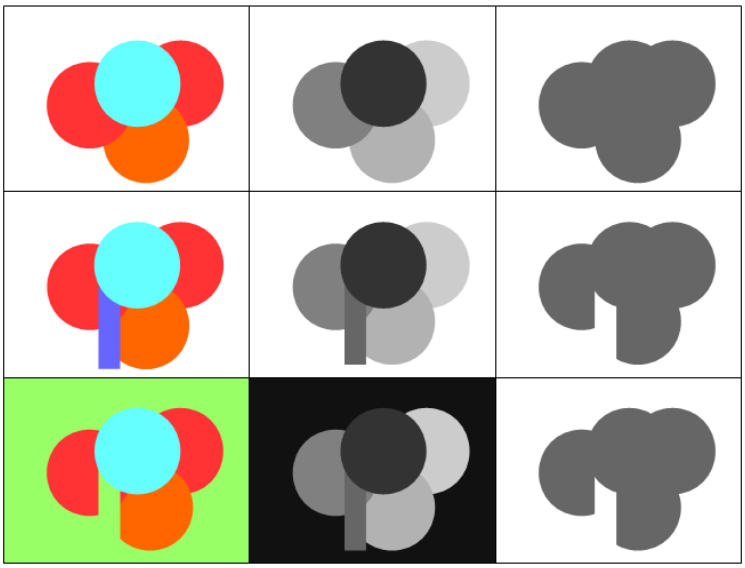
točko zaslonskega medpomnilnika še vrednost o globini objekta, ki ga tista točka predstavlja 4.4. Z njegovo pomočjo si lahko tako sistem za upodabljanje močno pomaga pri odločanju, ali je določen predmet, ki ga mora še izrisati dovolj v ospredju, da je v resnici viden na tisti točki slike.

Šablonski medpomnilnik (angl. *stencil buffer*) pa je popolnoma prepuščen programerjevi rabi, a je vseeno tudi strojno podprt celo z višjo prioriteto, kot globinski medpomnilnik. Tipično lahko vsaka točka zaslonskega medpomnilnika nosi po eno 8-bitno vrednost. Med izrisom določenih predmetov lahko tako najprej ukažemo, naj se na mestih, kjer se tisti predmet izriše, vrednost šablonskega medpomnilnika na določen način spremeni (na primer nastavi na določeno vrednost, poveča za 1 ali sešteje s konstanto), povsod drugje pa naj ostane enaka. Pri izrisu drugih predmetov pa lahko te vrednosti upoštevamo in ukažemo, naj se predmet izriše samo na mestih, kjer šablonski medpomnilnik ustreza določenim pogojem (enakost z nič, konstanto, večje ali manjše). Ker so vse te operacije strojno podprte na grafični kartici, so zelo hitre in ne povzročijo nobene opazne razlike v hitrosti delovanja.

V naši rešitvi šablone uporabljamo za reševanje problema prikazovanja rok pred virtualnimi objekti. Problem je namreč v tem, da pri naivni implementaciji sistema za obogateno resničnost, sliko s kamer uporabniku enostavno ves čas prikazujemo za ozadje virtualne scene, računalniško generirane predmete pa (lahko z določeno mero prosojnosti) postavimo predenj. To deluje čisto solidno, dokler se na ozadju pojavijo predmeti, ki so bliže od predmetov v navidezni sceni in bi jih zato povsem očitno morali prekrivati. V našem primeru v to kategorijo velikokrat sodijo uporabnikove roke, saj bi – glede na to, da so pogosto bliže njegovih oči, kot pa graf – morale le tega prekrivati. Tako delovanje je pomembno tudi z vidika ustvarjanja pravilnega občutka za globino in "dotikanja" posameznih vozlišč.

Zato smo problem rešili na sledeč način: Na izpraznjena zaslonski in šablonski medpomnilnik (neodvisno od njegovih vrednosti) v prvi fazi narišemo graf in vse točke šablonskega medpomnilnika na mestih, kjer to storimo, napolnimo z vrednostjo 1. V naslednji fazi na zaslonski medpomnilnik (prav

tako neodvisno od vrednosti šablonskega, vendar pa odvisno od globinskega medpomnilnika) glede na podatke pridobljene s Kinecta izrišemo modele rok, pripadajoče vrednosti v šablonskem medpomnilniku pa postavimo nazaj na 0. Tako imamo zdaj v šablonskem medpomnilniku z vrednostmi 1 označeno senco grafa, na kateri "manjkajo" deli ki jih prekrivajo roke. V zadnjem koraku pa povsem v ospredju izrišemo še ozadje z vklopljenim pogojem, da izris želimo samo na mestih, kjer je vrednost šablonskega medpomnilnika enaka 0. Rezultat je, da tista mesta ozadja, kjer so roke vidne pred grafom ustrezno tudi izrišemo. Shematski prikaz postopka lahko vidimo tudi na sliki 4.4



Slika 4.4: Prikaz upodabljanja in stanja medpomnilnikov. Na levi vidimo stanja v zaslonskem medpomnilniku, na sredi v globinskem medpomnilniku, na desni pa v šablonskem medpomnilniku. Najprej se izrišejo elementi z virtualne scene, ki nastavijo tudi vrednosti šablonskega medpomnilnika na 1. V drugem koraku se izrišejo elementi, skozi katere želimo, da se vidi ozadje (roke), zato na šablonskem medpomnilniku vrednosti postavimo nazaj na 0. V zadnjem koraku izrišemo ozadje pred vsemi ostalimi elementi, vendar le na mestih, kjer je šablonski medpomnilnik enak 0.

4.4 Moduli za zajem podatkov

Kot smo zapisali že na začetku poglavja 4, za zajem podatkov z naprav uporabljamo tri samostojne module, ki smo jih na podlagi specifičnih knjižnic izdelali sami. To so InterNect za zajem podatkov s Kinecta, InterCV za zajem podatkov s kamere in InterVX za zajem podatkov s senzorja rotacije na očalih.

Vsi trije moduli za zajem zunanjih podatkov imajo v osnovi zelo podobno zgradbo, zato tudi njihova uporaba v glavnem programu poteka po zelo podobnem principu. Ključno pa se seveda razlikujejo v podatkih, ki jih vračajo in načinu, na podlagi katerega jih pridobivajo. Vsakega od njih smo implementirali v svojem razredu, določenim v svojem paru datotek z izvorno kodo in zaglavjem. V nadaljevanju se bomo najprej posvetili skupnim značilnostim, nato pa si pri vsakem posebej ogledali še nekaj zanimivejših specifik in problemov, ki jih rešujejo.

Življenje vsakega od modulov za zajem podatkov poteka v naslednji obliki:

V glavnem programu (v metodi `go` razreda `Visualizer` v datoteki "Visualizer.cpp") na podlagi razreda najprej zgradimo svoj objekt. Nato pokličemo njegovo metodo `Init`, ki poskrbi za začetne nastavitve in zagon pripadajoče strojne opreme, s katere želimo zajemati podatke. Sam zajem nato začnemo z metodo `Start`, ki ustvari svojo nit in ji dodeli izvajanje kode določene v posebni funkciji, ki se nahaja zunaj razreda (v isti datoteki), kljub vsemu pa je njeno delovanje vezano nanj preko njenega parametra, ki vsebuje kazalec na dotični objekt. S pomočjo tega kazalca v določenih intervalih ves čas dokler metoda `IsRunning` vrača `true` poganja metodo `Update` in tako neodvisno od upodabljanja ves čas zajema podatke in jih posodablja.

Do teh podatkov lahko nato kadarkoli v zelo kratkem času dostopimo z ustrezno "get" metodo in jih uporabimo. Glede na to, da aplikacija tako teče v več nitih, moramo poskrbeti tudi za preprečevanje hkratnega dostopa. To na enostaven način dosežemo s konstruktom `mutex`, ki nam omogoča zaklepanje in tako povzroči, da se na kritičnem območju kode, ki izvaja pisanje ali branje podatkov, nahaja samo ena nit naenkrat. Metodo `Update` lahko seveda

pokličemo s katerekoli niti. To nam da tudi možnost, da na primer pred začetkom zajema videa preberemo posamezno sličico, ali da katerikoli modul enostavno uporabljamo v isti niti kot jo uporablja preostanek programa. Za zaključek zajema uporabimo metodo `Stop`, ki ustavi tek posebne niti, nato pa še sprostimo uporabo strojne opreme z metodo `UnInit`. Čisto na koncu objektov za zajem tudi ne smemo pozabiti uničiti s funkcijo `delete`.

Te lastnosti so vsem trem modulom skupne. Razlikujejo se tako predvsem v implementaciji metod `Init` in `Update`, ki sta tudi najdaljši in najbolj zapleteni.

4.4.1 Modul za zajem podatkov s Kinecta (InterNect)

S Kinecta zajemamo tako podatke o skeletu uporabnika, kot tudi o tem, ali ima v določenem trenutku katero od rok stisnjeno v pest.

Podatke o skeletu lahko dobimo na precej enostaven način. Vsa orodjarna za delo s skeleti uporabnikov je namreč v Kinect SDK orodjarni že dolgo časa. Tako je potrebno v postopku inicializacije samo najti delujoč priklopljen Kinect senzor, tako da se s števcem sprehodimo po vseh možnih indeksih senzorjev (od 0 do vrednosti, ki nam jo vrne `NuiGetSensorCount` na podlagi tega s pomočjo funkcije `NuiCreateSensorByIndex` ustvariti nov objekt razreda `INuiSensor` z zastavico `NUI_INITIALIZE_FLAG_USES_SKELETON`. Nato samo še pripravimo strukture za hrambo zajetih podatkov in v metodi `Update` vsakič, ko želimo svež okvir podatkov o uporabnikovem skeletu pokličemo funkcijo `NuiSkeletonGetNextFrame`.

Funkcija preko kazalca, ki ji ga podamo v parametru, napolni tabelo 4D vektorjev, v kateri vsak od njih nosi x, y in z pozicijo posameznega sklepa skeleta v prostoru, četrti parameter `w` pa ima lahko samo vrednosti 0 ali 1, ki določata vidnost sklepa. Tako ga lahko uporabimo kot pogoj za zanesljivost dobljenih podatkov.

Glede na to, da je podpora interakcijam (kamor sodi tudi zaznavanje stisnjenih/iztegnjenih dlani) na voljo šele od verzije 1.7 kot del orodjarne Kinect Toolkit, je tudi njena uporaba nekoliko zahtevnejša. Čeprav jo na koncu upo-

rabljamo povsem podobno, kot zgoraj opisani tok, moramo poleg posebnega toka interakcij predhodno sami zagnati še tok okvirov za skelet in tok okvirov za globinske slike s številčenjem igralcev (zastavica `NUI_INITIALIZE_FLAG_USES_DEPTH_AND_PLAYER_INDEX`) nato pa vsak zajet okvir z ostalih dveh tokov ročno posredovati toku za interakcije, ki par obdela in s krajšim zamikom vrne okvir z interakcijami. Potrebuje tudi implementacijo posebnega razreda `INuiInteractionClient` s funkcijo `GetInteractionInfoAtLocation`, s katero lahko omejimo, kateri kos 2D zaslona povzroči, da se interakcije sploh sprožijo. V našem primeru seveda želimo prijem zaznavati povsod, zato neodvisno od lokacije na 2D "interakcijskem platnu" samo vedno nastavimo parameter `IsGripTarget = True` in vrnemo znak za uspešno obdelavo. S tem v tok okvirov interakcij pošljemo nov okvir s podatki, ki ga v glavnem razredu v funkciji `Update` spet ujamemo in na njegovi podlagi priredimo vrednosti atributov, ki označujeta trenutno stanje stisnjene leve ali desne dlani (`mLGrip` in `mRGrip`).

Do zajetih podatkov lahko tako kadarkoli dostopamo z "get" metodami, za lokacije sklepov pa poleg trenutnih lokacij hranimo in omogočamo tudi dostop do prejšnjega stanja, saj potrebujemo podatke o premikih dlani za ustrezno transformiranje grafa.

4.4.2 Modul za zajem podatkov s kamer (InterCV)

Za dostop do obeh kamer na očalih uporabljamo knjižnico OpenCV. Ta nam poleg mnogih drugih funkcionalnosti, ki jih sicer v naši rešitvi ne uporabljamo, lahko bi pa koristile pri možnih nadgradnjah, ponuja tudi enostaven vmesnik za zajem slike s katerihkoli naprav, ki jih sistem prepozna in zna uporabljati kot kamere. V okolju Windows ta postopek poteka preko ogrodja DirectShow.

Zajem slike v metodi `Init` tako zaženemo s pomočjo funkcije `cvCapture FromCAM`, ki ji podamo indeks naprave preko katere želimo zajemati (0 pomeni privzeto napravo, v našem primeru levo oko, 1 pa desno). Funkcija v primeru uspeha vrne kazalec na strukturo, preko katere se nato lahko sklicujemo na to napravo, s pomočjo funkcije `cvGetCaptureProperty` pa lahko pridobimo

razne lastnosti o formatu in trenutnem načinu zajemanja. Če ne uspe, pa vrne vrednost 0 (oziroma kazalec `null`). OpenCV podpira dostop do več naprav hkrati. Tako lahko v našem primeru samo ustvarimo dva objekta razreda `InterCV`, ju na začetku nastavimo z različnimi indeksi naprav za zajemanje, in uporabljamo povsem neodvisno.

Do slikovnih podatkov pridemo na precej enostaven način. V metodi `Update` kličemo funkcijo `cvQueryFrame`, ki vrne kazalec na strukturo `IplImage`, ki vsebuje polje surovih RGB vrednosti za posamezne slikovne elemente. Pri tem naj poudarimo, da vrnjeni podatki niso poravnani na 32 bitov, kot to pričakuje OGRE za uporabo v teksturah, ampak na 24. Zato je potrebno, da jih za nadaljnjo uporabo najprej ustrezno pretvorimo. To storimo kar takoj po zajemu slike v zanki, ki je oblikovana tako, da ne popravlja vsakega slikovnega elementa posamično, ampak z nekaj logičnimi triki in zamikanjem bitov v enem obhodu obdela po štiri naenkrat. To njen tek namreč občutno pohitri.

V metodi `GetTexture` tako pokličemo samo funkcijo `memcpy`, ki ustrezne podatke samo hitro kopira v določen medpomnilnik brez obdelave vsakega slikovnega elementa posebej.

4.4.3 Modul za zajem podatkov s senzorjev nagiba na očalih (InterVX)

Ta modul se opira na knjižnico IWearSDK. V metodi `Init` pokliče samo funkcijo `IWROpenTracker`, ki zažene zajem podatkov, nato pa še s klicem `IWRZeroSet` ponastavi trenutno orientacijo očal na začetno stanje `brez nagiba`. To funkcijo lahko pokličemo tudi kasneje, ko ima uporabnik očala že na glavi, s pritiskom na preslednico. Na ta način lahko poravnamo smeri pogledov v navideznem in realnem svetu.

V metodi `Update` kličemo samo `IWRGetTracking`, ki nam vrne rotacije okrog vseh treh osi, te pa nato shranimo v ustrezne atribute, do katerih lahko v vsakem trenutku dostopamo preko "get" metod iz glavnega programa.

Poleg golega zajema modul omogoča tudi določeno mero glajenja z eno-

stavnim uteženim povprečenjem. Z uporabo parametrov `ponderv` in `ponderc` v metodi `Init` lahko nastavljamo, kako dolgo zgodovino meritev naj si sistem zabeleži in kako močno vsaka od teh meritev vpliva na končni rezultat. Na ta način odpravimo določeno mero tresenja, ki se pojavi zaradi šuma v senzorjih. Glede na to, da je branje podatkov s senzorjev hitro, ta postopek ne vpliva opazno na zmanjšano odzivnost pri obračanju glave. Vsaj če ne uporabimo pretirano dolgega seznama uteži.

4.5 Modul za uvoz in prikaz grafov

Graf, ki smo ga predhodno oblikovali ali izdelali v orodju Tulip moramo, če ga želimo opazovati s pomočjo naše rešitve, uvoziti in prikazati. Format in način izvoza sta natančneje predstavljena v poglavju 4.6, na tem mestu pa se bomo ogledali samo modul za uvoz in izris grafa na podlagi teh podatkov.

V zvezi z grafi smo v projektu implementirali razred `Graph`, ki je določen v datotekah `Graph.cpp` in `Graph.h`. Poleg tega v isti datoteki definiramo še dva razreda `GraphNode` in `GraphEdge`, ki nista namenjena neposredni rabi, ampak kot gradnika, ki ju uporablja `Graph`. `GraphNode` poleg tega uporablja še zunanji razred za izpis besedila v 3D prostoru `MovableText` [29].

Razred `Graph` vsebuje dva konstruktorja. Eden je namenjen predvsem testiranju prikaza grafov brez potrebnega uvoza zunanjih podatkov in generira naključen graf na podlagi želenega števila vozlišč in povezav, tako da vozlišča naključno razporedi v prostoru med uporabnikom in Kinectom ter jih med seboj poveže na naključen način. Drugi pa na podlagi datoteke, ki definira število vozlišč, njihove lastnosti, pozicije in povezave med njimi generira točno določen graf, ki smo ga prej izdelali s pomočjo orodja Tulip. V vsakem primeru konstruktor najprej izdela seznam vozlišč, jim priredi ustrezne lastnosti in jih pripne na skupno scensko vozlišče, nato izdela še seznam povezav in jim določi pripadajoča vozlišča.

Poleg tega vsebuje še metode, kot so: `Touch`, ki preveri, če se dane koordinate nahajajo znotraj katerega od vozlišč in za vsak zadetek pokliče ustrezno

metodo iz razreda `GraphNode`, `ResetTouch`, ki ponastavi vse spremembe, ki jih je povzročila metoda `Touch` in `Move`, ki graf transformira na podlagi dveh vektorjev določenih s štirimi točkami, ki opisujejo premik obeh rok.

Na ta način lahko iz glavne aplikacije v primeru, če zaznamo vsaj eno stisnjeno dlan, na podlagi teh podatkov samo pokličemo metodo `Move` in graf se premakne na ustrezen način. Razliko med eno in obema stisnjenima dlanema obravnavamo že v glavnem razredu. Če je stisnjena samo ena dlan, drugo roko ignoriramo, oziroma upoštevamo, kot da se druga roka premika na identičen način kot prva in parametre metode `Move` zato napolnimo na ustrezen način. Tako graf samo premaknemo na drugo lokacijo (translacija), ne spreminjamo mu pa velikosti (skaliranje) ali rotacije. Če zaznamo, da sta stisnjeni obe dlani, pa omogočamo poln nabor transformacij, saj se v tem primeru roki gibljeta neodvisno ena od druge in lahko graf tudi obračamo ali mu spreminjamo velikost (pri tem velikost vozlišč ostaja enaka, spreminjajo se zgolj razdalje med njimi.

4.6 Izvoz podatkov iz okolja Tulip

Kot smo zapisali že v poglavju 3.3.7, je Tulip v svoji osnovi programska orodjarna, ki jo lahko uporabljamo v jeziku Python ali C/C++ povsem neodvisno od uporabniškega vmesnika. Ta je tako le pripomoček za olajševanje dela in za bolj intuitivno rabo njegovih funkcionalnosti, ponuja pa nam tudi dostop do interpreterja v Pythonu, s pomočjo katerega lahko neposredno s programskimi ukazi manipuliramo graf, spremembe pa v realnem času opazujemo na grafičnem prikazu. Python je seveda splošni programski jezik in z njegovo pomočjo lahko na precej enostaven način želene podatke preberemo, nato pa jih z običajnimi ukazi za delo s tokovi in datotekami zapišemo v (besedilno) datoteko.

S tem namenom smo napisali tudi vnaprej pripravljeno skripto, ki prebere vse želene podatke o grafu, nato pa jih zapiše v sledečem formatu:

- V prvo vrstico zapiše število vozlišč n (na podlagi tega podatka ob

branju ugotovimo, koliko sledečih vrstic označuje opise vozlišč).

- V naslednjih n vrsticah zapiše podatke o posameznih vozliščih v formatu: `[x,y,z] [r,g,b,a] [sx,sy,sz] [komentar]`. (Prve tri vrednosti povedo lokacijo vozlišča, druge štiri barvo in prosojnost, tretje tri velikost in raztegnjenost krogle v določenih smereh, sledi pa jim še napis, ki se pojavi, če se vozlišča v prikazovalniku "dotaknemo". Tega v Tulipu določimo z lastnostjo `viewLabel`)
- V naslednji vrstici zapiše število povezav m
- V naslednjih m vrsticah zapiše pare števil, ki povedo indeksa začetnega in končnega vozlišča posamezne povezave.

Datoteko takega formata lahko po potrebi brez težav ročno popravimo, pa tudi enostavno preberemo s pregledovalnikom, saj je format podatkov enolično določen in ga neodvisno od omrežja beremo povsem na enak način. Format ni standarden predvsem iz razloga kompleksnosti zapisa. Za naše potrebe namreč zadostujejo precej omejeni podatki o zgradbi omrežja, ki jih na zelo enostaven način lahko beremo in urejamo tudi ročno.

Ob morebitni nadgradnji projekta, kjer bi prikaz grafa podpiral tudi kakšne zapletenejše atribute, pa bi bilo smiselno dodati sistemu še komponento za uvoz in branje raznih standardnih formatov zapisa.

Poglavje 5

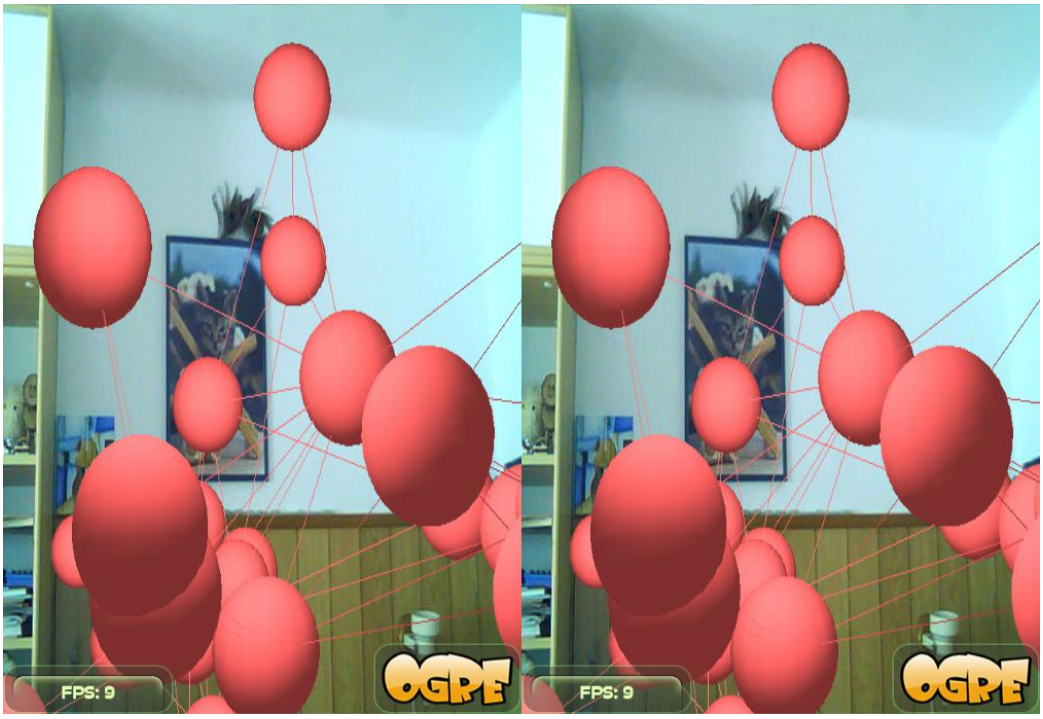
Zaključki

S pomočjo danih orodij smo v predstavljeni rešitvi zastavljene cilje dosegli. Ugotovili smo, da je tehnologija obogatene resničnosti za prikazovanje in raziskovanje omrežij primerna, vendar poleg elegantnih in intuitivnih rešitev pregleda grafa, prinese tudi nekatere zaplete, ki jih je potrebno razrešiti. Opazili smo tudi določene pomanjkljivosti, ki jih z dano strojno opremo ni bilo mogoče odpraviti. Ob testiranju in uporabi končnega izdelka smo dobili nekaj idej za mogoče izboljšave in nadaljevanje dela v prihodnosti. Zaslonski posnetek naše rešitve lahko vidimo na sliki 5.1.

Uporabnik lahko graf opazuje stereoskopsko v treh dimenzijah in na realnem ozadju, lahko se premika v prostoru, pri čemer se graf obnaša na zelo podoben način, kot če bi bil del realnega sveta, torej miruje glede na ozadje. Z grafom lahko uporabnik interaktivno upravlja na način, da ga s pomočjo gest poljubno transformira in z dotikom povzroči prikaz besedilnih podatkov posameznega vozlišča.

Rešitev je poleg tega napisana izrazito modularno, z ločenimi razredi, ki so dostopni preko vmesnikov in jih je mogoče prilagajati ali zamenjati brez neposrednega poseganja v jedro aplikacije. Koda je jasno razdeljena in sistematično komentirana, kar bi morebitni nadaljnji razvoj precej olajšalo.

Obstaja pa tudi nekaj pomanjkljivosti. Najočitnejša težava, ki je nismo uspeli dokončno rešiti, je glede povratne informacije dotika vozlišča in postavljanja predmetov iz realnega sveta (specifično – uporabnikovih rok) pred



Slika 5.1: Zaslonska slika rešitve, ki jo implementira naša aplikacija "Visualizer"

virtualne predmete vizualizacije. Obe težavi se povezujeta, saj je glede na to, da nismo imeli na razpolago nobenega orodja, s katerim bi lahko uporabniku dali vtis fizičnega dotika virtualnih predmetov, še toliko pomembneje, da to jasno prikažemo vsaj vizualno. Pomembno je torej, da uporabnik vsaj vidi, kdaj se njegova roka potopi v navidezni predmet. Ugotovili smo, da samo stereoskopski vtis enake globine roke in vozlišča za dobro predstavo o globini ne zadošča, če je ta ves čas narisana na ozadju za vozliščem. Zato smo se odločili, da glede na podatke o lokaciji rok te izrišemo še v navideznem svetu, nato pa na dele, kjer bi jih aplikacija izrisala, izrišemo ozadje (sliko uporabnikovih rok). Ideja sicer deluje, lahko pa zaradi neizvedljive popolne poravnanosti svetov pride do nekaj-centimetrskega odstopanja, kar v praksi pomeni, da na je na področju, kjer pričakujemo roko, na sliki realnega sveta v resnici ozadje ob roki, roka sama pa je prekrita z grafom. To je moteče

še toliko bolj, ker roke v virtualnem svetu izrišemo dokaj grobo s pomočjo valjev in krogel. Zato smo mnenja, da bi bilo ta problem mogoče v veliki meri vsaj olajšati s pomočjo optičnega zaznavanja rok s slike ozadja ter njihovega izrezovanja, ali pa izrezovanja oblike glede na surove globinske podatke s Kinecta. Obe opciji smo proučili in ugotovitve zapisali v poglavju 5.2.

Ves čas nas je nekoliko motilo tudi dejstvo, da sta zaslončka na očalih Wrap 920AR majhna in je zato vidno polje zelo ozko, kar precej zmanjša kvaliteto uporabniške izkušnje. Za opazovanje večjih grafov, ali delov, ki se ne nahajajo neposredno pred uporabnikom, mora ta nujno zasukati glavo, kar pomeni, da s pogleda izgubi preostanek grafa, ki ga je opazoval pred tem, premikanje oči pa sploh ni smiselno.

Težave nam je povzročal tudi način priklopa očal na osebni računalnik, kjer smo razvijali rešitev. Glavna težava pri tem leži v dejstvu, da se tako osvetlitev za zaslončka, kot tudi obe kameri napajata preko istega USB-vmesnika, kar pomeni precejšnjo obremenitev na posamezni vtičnici računalnika. Uporaba na prenosnem računalniku vsaj ob delovanju s pomočjo baterije sploh ni mogoča, obe kameri naenkrat pa smo še na stacionarnem računalniku zaradi tega razloga lahko uporabljali samo v primeru, če smo vse ostale vtičnice v isti skupini pustili neuporabljene. Identifikacija te težave je bila zaradi na videz nepredvidljivega obnašanja zelo težavna za razrešitev in je zahtevala veliko časa in frustracije, da smo jo končno uspeli rešiti in najti delujočo konfiguracijo uporabljenih in neuporabljenih vtičnic. Poleg tega Kinect z uradnimi gonilniki ne podpira določenih USB-delilnikov (angl. *USB Hub*) in ga prav tako nismo mogli uporabljati na prenosnem računalniku, kjer je izbira in količina USB-vtičnic zelo omejena.

Edina strojna komponenta, ki ni povzročala nobenih težav in je delovala v vseh primerih, je bil tako senzor za zaznavanje nagiba glave na očalih. Delovanje Kinecta in senzorja nagiba glave pa sta nas po drugi strani s svojo natančnostjo kar nekoliko presenetila. Če roki (ali drug predmet) postavimo med Kinect in glavo, jo bo ta sicer prenehal natančneje zaznavati in bo zato nekoliko "poskakovala", ampak temu se s čisto tehničnega vidika enostavno

ni mogoče izogniti. Sicer po naših izkušnjah v večini primerov oba senzorja delujeta presenetljivo dobro.

5.1 Ideje za nadaljevanje dela

Za praktično in bolj množično uporabo naše rešitve bi bilo smiselno izdelati predvsem prijaznejši uporabniški vmesnik, ki bi omogočal enostavnejšo izbiro nastavitev ter uvažanje grafov. Z vidika prijaznosti do uporabnika in lažje namestitve bi bilo potrebno tudi zbrati dinamične komponente vseh knjižnic, ki jih rešitev potrebuje in izdelati zbirko ali namestitveni program, s katerim bi poskrbeli, da so vse potrebne odvisnosti pred prvim zagonom zadoščene.

Glede na to, da gre za orodje akademske in brezplačne narave, bi bilo zanimivo poleg okolja Windows podpreti tudi druge sistemske platforme, predvsem Linux. Ta problem ravno zaradi že omenjene odvisnosti nekaterih uradnih knjižnic od okolja Windows ne bi bil lahek, bi bil pa ob globlji raziskavi alternativnih opcij in neuradnih orodij vseeno mogoč, predvsem v prihodnosti, ko bo po vsej verjetnosti alternativ vedno več. Poleg tega smo za vse dele, kjer je bilo to brez dodatnih zapletov mogoče, že uporabili odprtokodne in večplatformne rešitve, kot sta OGRE in OpenCV.

Za bolj splošno rabo bi bilo zelo smiselno podpreti tudi strojno opremo drugih proizvajalcev z drugačno podporo, predvsem očala, saj trenutna zaradi ozkega zornega kota in nerodne ožičene povezave s podaljški (prikazane na sliki 5.2), pa tudi zaradi splošne dostopnosti in cene niso ravno idealna izbira. Alternativno bi bilo zanimivo preskusiti očala Oculus Rift v kombinaciji z modulom s kamerami za podporo obogatene resničnosti.

Za razširitev funkcionalnosti, bi lahko dodali tudi možnost urejanja grafov, oziroma opcijo dinamičnega popravljanje oblike. Pri tem bi si lahko pomagali tudi z neposredno rabo programskega vmesnika knjižnice Tulip.

Pri dodajanju novih funkcionalnosti bi bilo potrebno skrbeti tudi za izboljšanje učinkovitosti. Predvsem pri prikazovanju in hkratnem urejanju velikih in zapletenih grafov, bi se vsaj na starejših in manj zmogljivih sistemih

Slika 5.2: Očala Vuzix Wrap 920AR skupaj z USB-podaljški in "oprtnico" za nošenje kabla.

lahko pojavila težava nekoliko slabše odzivnosti. To težavo bi lahko rešili tudi z uporabo novejših in zmogljivejših sistemov.

5.2 Izboljšanje prikaza rok

Kot smo omenili že v poglavju 5, je ena izmed najbolj motečih težav ravno v prikazu uporabnikovih rok na realističen način. Ta vidik bi bilo vsekakor smiselno izboljšati, saj bi veliko pripomogel k boljši uporabniški izkušnji. V tej smeri smo že raziskali dve opciji, ju pa zaradi njunega obsega nismo uspeli izvesti.

Prva ideja bi bila uporaba kinecta za določanje obrisa roke. Težava pri tem je predvsem ta, da Kinect na sceno gleda s približno nasprotne strani glede na uporabnikov pogled in ne zajame povsem enake oblike, kot jo vidi uporabnik. Poleg tega bi slika s kamer in obris roke na podlagi zaznanih podatkov s Kinecta ostala vsaj nekoliko neporavnana, kar bi hitro lahko pomenilo, da bi namesto roke na ustreznem mestu prikazali "luknjo skozi

navidezne predmete do ozadja" v obliki roke, dejanska slika roke na ozadju pa bi bila prekrita z virtualnimi predmeti na sceni. S podobno idejo so se ukvarjali avtorji članka [9], katerih rešitev prav tako temelji na globinskih podatkih, le da jih ti pridobijo s pomočjo disparitete stereoskopskih slik z očal.

Druga ideja je uporaba določenih rešitev računalniškega vida za zaznavanje rok neposredno s kamerami. V zvezi s tem smo si ogledali rešitev avtorja dr. Andola X. Lija [19]. Na ta način bi se izognili težavi s poravnanostjo in nepravilnega prikaza, saj bi lahko robove rok določili zgolj na podlagi slike s kamer z očal. Težava je, da sistem v tem primeru postane močno odvisen od kvalitete slike ozadja, tekstur, razlik v obliki roke posameznika, kota pogleda na roko, osvetlitve, pa tudi zaznavi stisnjene pesti, ki je manj značilna, kot odprta dlan z ločenimi prsti. Ob idealnih okoliščinah bi tako lahko vseeno dosegli boljšo poravnanost prikazanih rok pred navideznimi predmeti, problem pa bi nastal v poravnanosti med lokacijami, ki jih vrne Kinect in prikazanima rokama. S podobnimi idejami se je ukvarjal avtor članka [7], ki prav tako pride do zaključka o veliki odvisnosti uspešnoti delovanja sistema od kvalitete slik in najdenih robov.

V vsakem primeru bi bile v tej smeri potrebne še nadaljnje raziskave.

Literatura

[1] G. L. Alexanderson, "Euler and Königsberg Bridges: A Historical View", *Bulletin (New Series) of the American Mathematical Society*, letnik 43, številka 4, 2006, str. 567-573.

[2] K. Appel, W. Haken, "Every Planar Map is Four Colorable", *Bulletin of the American Mathematical Society*, letnik 82, številka 5, 1976.

[3] R. T. Azuma, "A Survey of Augmented Reality", *Presence: Teleoperators and Virtual Environments*, letnik 6, številka 4, avgust 1997, str. 355-358.

[4] R. Azuma, Y. Baillot, R. Behringer, S. Feiner, S. Julier, B. MacIntyre, "Recent Advances in Augmented Reality", *Computer Graphics and Applications*, letnik 21, številka 6, december 2001, str. 34-47.

[5] V. Batagelj, A. Mrvar, "Pajek: A Program for Large Network Analysis", *Connections*, letnik 21, številka 2, 1998, str. 47-57.

[6] D. Belcher, "Using Augmented Reality for Visualising Complex Graphs in Three Dimensions", *Mixed and Augmented Reality*, 2003.

[7] M.-O. Berger, "Resolving occlusion in augmented reality: a contour based approach without 3D reconstruction", *Computer Vision and Pattern Recognition*, zbornik, 1997, str. 91-96.

[8] S. P. Borgatti, M. G. Everett, L. C. Freeman. *Ucinet for Windows: Software for Social Network Analysis*. Cambridge University Press, 2005.

[9] D. E. Breen, R. T. Whitaker, E. Rose, M. Tuceryan, "Interactive Occlusion and Automatic Object Placement for Augmented Reality", *Computer Graphics Forum*, letnik 15, številka 3, 1996, str. 11-22.

[10] M. Fourment, M. R. Gillings, "A Comparison of Common Programming Languages Used in Bioinformatics", *BMC Bioinformatics*, letnik 9, številka 1, 2008.

[11] R. Furlan, "Build Your Own Google Glass", *Spectrum*, 2012, str. 20-21.

[12] F. Harary. *Graph Theory*. Addison-Wesley, 1969.

[13] R. Ihaka, R. Gentleman, "R: A Language for Data Analysis and Graphics", *Journal of Computational and Graphical Statistics*, letnik 5, številka 3, 1996.

[14] R. Liebo, H. Kaufmann, "Visualisation of Complex Function Graphs in Augmented Reality", *magistrsko delo*, 2006.

[15] I. Milne, G. Rowe, "OGRE: Three-Dimensional Program Visualization for Novice Programmers", *Education and Information Technologies*, 2004, letnik 9, številka 3, str. 219-237.

[16] S Pedroni, N Rappin. *Jython Essentials*. OReilly, 2002.

[17] T. Starner, S. Mann, B. Rhodes, J. Levine, J. Healey, D. Kirsch, R. Picard, A. Pentland, "Augmented Reality Through Wearable Computing", *Presence: Teleoperators and Virtual Environments*, letnik 6, številka 4, avgust 1997, str. 386-398.

[18] L. Šubelj, N. Blagus, Š. Furlan, B. Klemenc, A. Kumer, D. Lavbič, A. Zrnec, S. Žitnik, M. Bajec, "Analiza kompleksnih omrežij: osnovni pojmi in primeri uporabe v praksi", *zbornik konference informatike v javni upravi*, 2010.

[19] (2013) Andol, orodje za razpoznavanje gest. Dostopno na: http://www.andol.info/hci/2059.htm

[20] (2013) Bullet Physics - uradna stran. Dostopno na: http://bulletphysics.org/wordpress/

[21] (2013) Gameplay3D - uradna stran. Dostopno na: http://gameplay3d.org/

[22] (2013) Gephi – referenca programskega vmesnika. Dostopno na: https://gephi.org/developers/

[23] (2013) Gephi - uradna stran. Dostopno na: https://gephi.org/

[24] (2013) Graphviz – uradna stran. Dostopno na: http://www.graphviz.org/

[25] (2013) GUESS – uradna stran. Dostopno na: http://graphexploration.cond.org/

[26] (2013) IGraph – uradna stran. Dostopno na: http://igraph.sourceforge.net/

[27] (2013) Kinect Studio. Dostopno na: http://msdn.microsoft.com/en-us/library/hh855389.aspx

[28] (2013) Kinect za razvijalce. Dostopno na: http://www.microsoft.com/en-us/kinectforwindowsdev/Start.aspx

[29] (2013) MovableText. Dostopno na: http://www.ogre3d.org/tikiwiki/tiki-index.php?page=MovableText

[30] (2013) OGRE - uradna stran. Dostopno na: http://www.ogre3d.org/

[31] (2013) OGRE – zmogljivosti. Dostopno na: http://www.ogre3d.org/about/features

[32] (2013) OpenCV – uradna stran. Dostopno na: http://opencv.org/

[33] (2013) OpenKinect - uradna stran. Dostopno na: http://openkinect.org/wiki/Main_Page

[34] (2013) OpenNI - uradna stran. Dostopno na: http://www.openni.org/

[35] (2013) Pajek - wiki. Dostopno na: http://pajek.imfm.si//doku.php

[36] (2013) Primer stereoskopskega posnetka. Dostopno na: http://kottke.org/photos/stereo/index.html

[37] (2013) Primer točk IR svetlobe, ki jih projicira kinect. Dostopno na: http://azttm.wordpress.com/2011/04/03/kinect-pattern-uncovered/

[38] (2013) Project Glass. Dostopno na: http://www.google.com/glass/start/

[39] (2013) SDL – uradna stran. Dostopno na: http://www.libsdl.org/

[40] (2013) Sestava senzorja Kinect in specifikacije. Dostopno na: http://msdn.microsoft.com/en-us/library/jj131033.aspx

[41] (2013) Seznam projektov, ki uporabljajo knjižnico OGRE. Dostopno na: http://www.ogre3d.org/tikiwiki/tiki-index.php?page=Projects

[42] (2013) Shematski prikaz senzorja Kinect. Dostopno na: http://msdn.microsoft.com/en-us/library/jj131033.aspx

[43] (2013) Tulip - uradna stran. Dostopno na: http://tulip.labri.fr/TulipDrupal/

[44] (2013) Unity 3D – uradna stran. Dostopno na: http://unity3d.com/

[45] (2013) Vuzix WrapAR - uradna stran. Dostopno na: http://www.vuzix.com/augmented-reality/products_wrap920ar.html

[46] (2013) Vuzix za razvijalce. Dostopno na: http://www.vuzix.com/support/developer_program.html

[47] (2013) Zoomgraph - uradna stran. Dostopno na: http://www.hpl.hp.com/research/idl/projects/graphs/zoom.html