Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Ivan Mlinarić

Tridimenzionalna vizualizacija avtomatiziranega skladišča v realnem času

DIPLOMSKO DELO
VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

doc. dr. Iztok Lebar Bajec
MENTOR

Ljubljana, 2013

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00524/2013
Datum: 02.09.2013

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

Kandidat: **IVAN MLINARIČ**

Naslov: **TRIDIMENZIONALNA VIZUALIZACIJA AVTOMATIZIRANEGA SKLADIŠČA V REALNEM ČASU**

REAL-TIME THREE-DIMENSIONAL VISUALISATION OF AN AUTOMATED WAREHOUSE

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

Tematika naloge:

Dinamičnost dogajanja v avtomatiziranih visoko-regalnih skladiščih je za tiste, ki jim je dovoljen prvi pogled vanje prava paša za oči. Že sam prostor skladišča je lahko zastrašujoč, regali dosegajo višine do 40 m, skladišča pa se upravlja samodejno in vrvež, ki se pri tem dogaja ni zanemarljiv.

V diplomski nalogi preučite princip delovanja avtomatiziranih visoko-regalnih skladišč, skladiščni informacijski sistem podjetja Epilog d.o.o. ter za slednjega zasnujte aplikacijo, ki bo prikazovala aktualno dogajanje v skladišču v tridimenzionalnem prostoru.

Mentor: doc. dr. Iztok Lebar Bajec

Dekan: prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Namesto te strani se vstavi *original izdane teme diplomskega dela s podpisom mentorja in dekana ter žigom fakultete, ki ga diplomant dvigne v študentskem referatu, preden odda izdelek v vezavo!*

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani izjavljam, da sem avtor dela, da slednje ne vsebuje materiala, ki bi ga kdorkoli predhodno že objavil ali oddal v obravnavo za pridobitev naziva na univerzi ali drugem visokošolskem zavodu, razen v primerih kjer so navedeni viri.
S svojim podpisom zagotavljam, da:

- sem delo izdelal samostojno pod mentorstvom doc. dr. Iztoka Lebarja Bajca,
- so elektronska oblika dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko in
- soglašam z javno objavo elektronske oblike dela v zbirki "Dela FRI".

— Ivan Mlinarić, Ljubljana, september 2013.

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Ivan Mlinarić

Tridimenzionalna vizualizacija avtomatiziranega skladišča v realnem času

POVZETEK

Diplomska naloga opisuje izdelavo aplikacije za vizualizacijo visokoregalnih skladišč za dolge materiale. V prvem delu so opisani realni sistemi in njihove lastnosti, ki odločilno vplivajo na samo izvedljivost vizualizacije. Nato je opisana tehnologija, na kateri aplikacija temelji. Sledi opis, na kakšen način so izdelani 3D-modeli objektov. Največji del diplomske naloge predstavlja opis same strukture kode in najpomembnejših delov, kot so podatkovni model, izrisovanje, entitetni razredi, krmilni razredi, konfiguracija itn.

Kljucne besede: 3D, grafika, vizualizacija, skladišče

University of Ljubljana
Faculty of Computer and Information Science

Ivan Mlinarić

Real-time three-dimensional visualisation of an automated warehouse

ABSTRACT

Thesis describes development of application for visualization of high bay storage warehouses for long material. At the beginning of the thesis are described real world systems and their properties, which decisively affect expediency and feasibility of building visualization upon them. In the second part it is described technology on which built application is based upon and how 3D models are built. The majority of the thesis describes structure and most important elements of application like model, rendering, entity classes, controllers, configuration, etc.

Key words: 3D, graphics, visualization, warehouse

ZAHVALA

Zahvaljujem se Damjanu Širci, direktorju podjetja Epilog d.o.o, ki mi je omogočil izdelavo same aplikacije v okviru praktičnega izobraževanja znotraj podjetja. Tadeju Vretiču se zahvaljujem za pomoč pri izdelavi 3D-objektov, Helgeju Försterju pa za izdelavo knjižice JPCT in za vso pomoč, ki mi jo je nudil prek foruma. Hvala tudi mentorju, doc. dr. Iztoku Lebarju Bajcu, moji družini, ki je dolga leta potrpežljivo čakala, da se lotim tudi pisanja tega dokumenta, in ne nazadnje hvala tudi "stricu Googlu", saj brez njega ne bi nikoli uspel spisati tega dokumenta v LaTeXu.

— Ivan Mlinarić, Ljubljana, september 2013.

KAZALO

1 Uvod

1.1 Izbira

Kakšen realen sistem bi bil zanimiv in hkrati tudi primeren za integracijo v realno-časovno računalniško 3D-vizualizacijo? Sistem mora biti v izvajanju kar se da veliko časa, če je le možno 24 ur na dan, sedem dni v tednu. Na primer: polnilna linija piva v pivovarni. Primeren sistem je tak, v katerem je dogajanje predvidljivo, premiki objektov sistema pa kar se da enostavni. Na primer: težko bi natančno vizualizirali valovanje morja v Luki Koper v realnem času, saj na valovanje vpliva preveč faktorjev. Samo valovanje pa je zapleteno in ni vedno natančno predvidljivo, medtem ko bi gibanje ladij v Koprskem zalivu že veliko lažje vizualizirali. Na voljo nam morajo biti podatki, ki jih potrebujemo za vizualizacijo. V prejšnjem primeru torej GPS-lokacije oz. podatki radarja o pozicijah, smeri in hitrosti ladij ter šifrant osnovnih podatkov o ladjah, kot so velikost, teža in podobno. Eden od glavnih ciljev vizualizacije je predvsem, da imamo v katerem koli trenutku z oddaljene lokacije vpogled v stanje sistema in v njegovo dogajanje, in sicer v obliki tridimenzionalne računalniške grafike. Izredno uporabno pa je tudi, če sama vizualizacija omogoča neposredno uporabniško interakcijo s sistemom in s tem hkrati tudi

takojšnji vpogled v rezultate ter spremembe, ki jih prinese uporabnikova interakcija s sistemom. Na primer: ko bi uporabnik lahko s klikom miške na ladjo povzročil, da se ta ustavi, spremeni hitrost, zavije itd. Sama interakcija ni nujno potrebna komponenta računalniške vizualizacije v realnem času. V primeru ko te komponente ni, lahko vizualizacija služi spremljanju ali predstavljanju vizualiziranega sistema. Interakcija, če je ta potrebna, lahko poteka kako drugače. Na primer: kontrolor plovbe opazi nevarnost trka dveh ladij prek vizualizacije, nato pa ladji opozori preko radijske zveze.

Na podlagi teh ugotovitev in razmišljanj se mi zdi vizualiziranje avtomatiziranega visokoregalnega skladišča primeren sistem. Avtomatizirana visokoregalna skladišča največkrat obratujejo nepretrgoma. V njih ni človeškega faktorja nepredvidljivosti, saj je celotno dogajanje robotizirano oz. avtomatizirano. Nabor gibov komponent skladišča je vnaprej znan in predstavljiv z matematičnimi funkcijami. Tako skladišče je največkrat tudi podprto z dobrim informacijskim sistemom, ki hrani vse oz. večino potrebnih informacij za vizualizacijo. Potrebujemo le dostop do njih.

1.2 Avtomatizirano visokoregalno skladišče

Sam princip delovanja avtomatiziranega skladišča se v osnovi popolnoma razlikuje od delovanja konvencionalnega ročnega skladišča. V ročnih skladiščih je osnova gibanje človeka proti uskladiščenemu materialu. Človek največkrat s pomočjo vozil, kot so viličarji, manipulira z materialom in ga premika po skladišču.

V avtomatiziranih skladiščih pa je osnova gibanje uskladiščenega materiala k človeku. Skladišča so popolnoma oz. v veliki meri avtomatizirana, kar pomeni, da delavcem v skladišču ni treba iskati in premikati materiala, temveč za to poskrbijo razne mehanske naprave in roboti, kot so dvigala, tračne enote, hidravlični vozički, tekoči trakovi, magnetna oprijemala itd. Ime (visokoregalna) izvira iz dejstva, da so največkrat več deset metrov visoka, saj za dvigala premagovanje višin ni nobena ovira. Velika prednost takega skladišča je, da se lahko zgradi na majhni površini s sorazmerno velikim faktorjem kapacitete, čeprav pa ni nič nenavadnega, če je poleg visokoregalnega skladišča tudi ročno.

Poznamo več tipov visokoregalnih skladišč, ki so ponavadi poimenovana glede na tip transportnih enot skladišča. Na primer: paletna visokoregalna skladišča, kasetna skladišča za dolgi material (palice, profili), kasetna skladišča za pločevino, kontejnerska

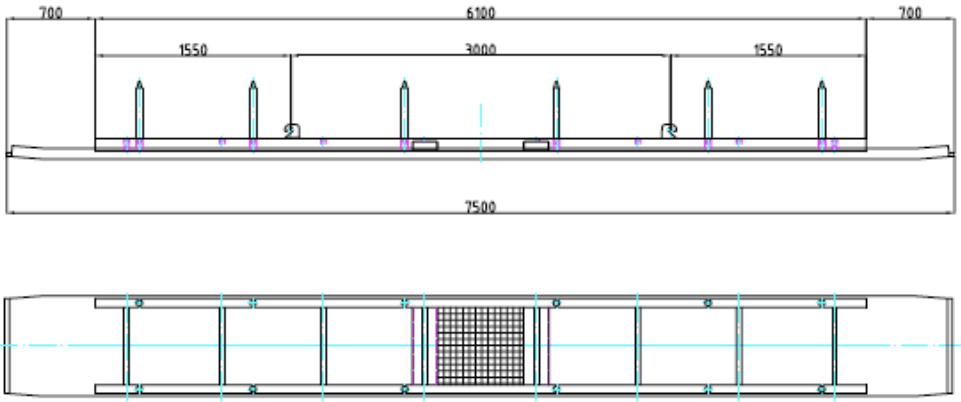
skladišča itd. Primer takega skladišča je na sliki 1.1.

Slika 1.1: Avtomatizirano visokoregalno kasetno skladišče za dolge materiale. Asas, Akyazi - Turčija.

1.3 Avtomatizirano visokoregalno kasetno skladišče za dolge materiale

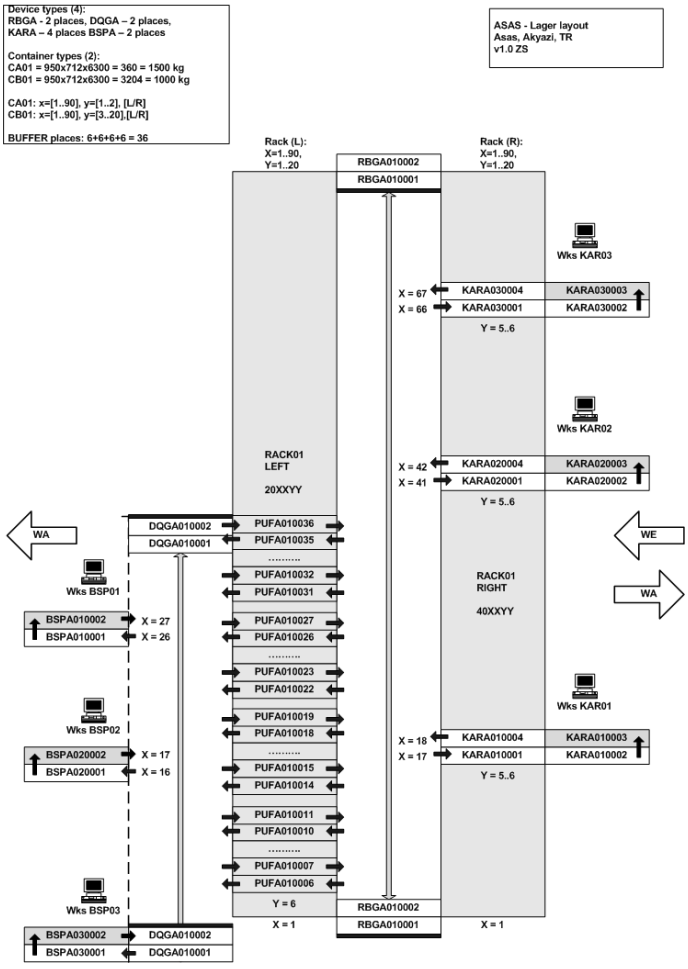
Kasetna skladišča kot vsebnike uporabljajo kasete. Osnovne kasete so kovinske enote, v katere je moč naložiti material, lahko pa tudi druge manjše tipe vsebnikov, kot so na primer leseni zaboji ali kovinske kasete manjših dimenzij, primerne za transport v logistiki. Osnovne kasete nikoli ne zapustijo sistema. V njih se na za to predvidenih mestih odlaga oz. se z njih odvzema material (oz. drug manjši tip vsebnika). Kasete so po navadi ozke in dolge, primerne za vstavljanje materiala, kot so kovinske palice, cevi, profili oken ipd. Slika 1.2 prikazuje načrt kasete in načrt vloženega kovinskega vsebnika.

Glavnina oz. največji del skladišča so po navadi regali in med njimi visokoregalna dvigala, ki pobirajo in prinašajo kasete v regal oz. iz njega. Regali so razdeljeni matrično na lokacije, v katerih so shranjene kasete. Regalna dvigala premikajo kasete na razne ostale naprave, ki kasete primikajo naprej do komisionirnih mest, lahko pa tudi kar direktno na naprave, ki so že same po sebi komisionirna mesta. Na komisionirnih mestih se izvede materialna operacija, kot so na primer odprema, žaganje, konfekcioniranje itd. Ta korak



Slika 1.2: Kaseta z vloženim vsebnikom (skip), dokumentacija podjetja Epilog d.o.o.

v procesu lahko izvede človek, lahko pa se tudi ta izvede popolnoma avtomatizirano s pomočjo naprav, kot so vakuumska ali magnetna oprijemala, robotske roke, avtomatske žage itd. Primer postavitve takega skladišča je prikazan na sliki 1.3. Na sliki so prikazane postavitve posameznih naprav in začasnih odložišč j z označbami posameznih prostorov, logične kooridinate naprav in lokacije PC-računalnikov. Na sliki so z večjimi puščicami označeni tudi procesi (*WE* - uskladiščenje materiala, *WA* - izskladiščenje materiala).



Slika 1.3: Skica postavitve naprav. Asas, Akyazi - Turčija.

1.4 Omejitve pri preslikavi v računalniško grafiko

Celotno skladišče takega tipa je sestavljeno iz ogromno velikih in majhnih delov. Ker pa moramo sistem izrisovati v realnem času, se moramo omejiti na glavne komponente in jih tudi poenostaviti oz. znižati njihovo kompleksnost oblike do te mere, da se jih da predstaviti s tolikšnim številom poligonov, ki jih današnja povprečna grafična kartica lahko izriše v realnem času. Točne lokacije posameznih komponent nam niso dostopne v vsakem trenutku, zato moramo premike interpolirati in jih le na določenih točkah sinhronizirati oz. korigirati, če je to potrebno. Vizualizacija ne sme vplivati na performance izvajanja dela v samem skladišču.

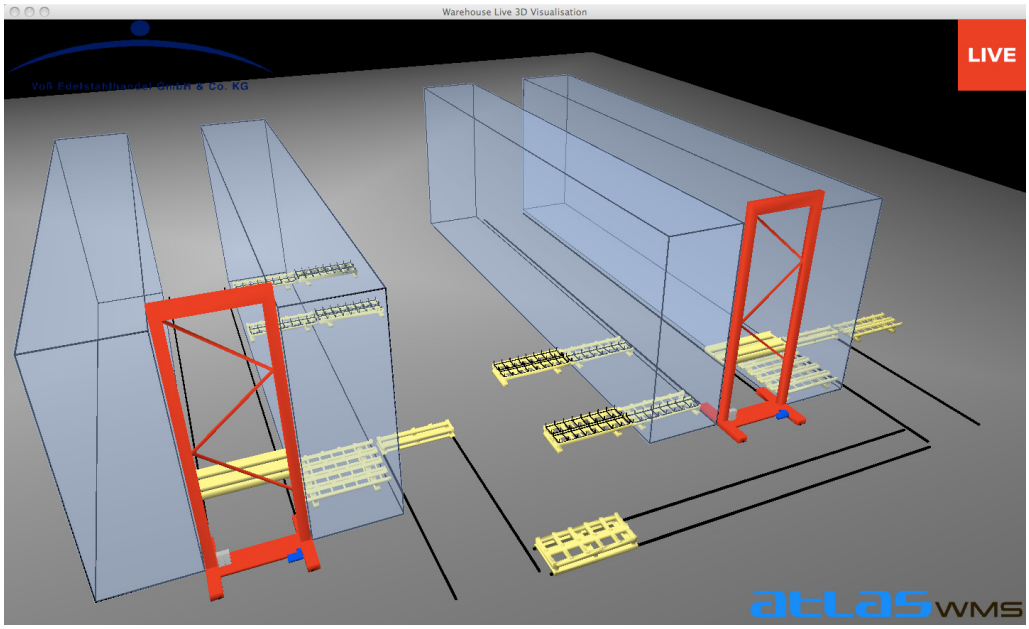
1.5 Izbor funkcionalnosti, predstavljenih v vizualizaciji

Vizualizacija podpira sledeč nabor naprav: *RBGA* - regalno dvigalo z eno mizo, *RBGB* - regalno dvigalo z dvema mizama, *KARA* - karosel s štirimi prostori, *KARB* - karosel s šestimi prostori, *VWLA* - tračni voziček z eno mizo, *VWLB* - tračni voziček z dvema mizama in *VWQ* - voziček s poljubnim številom začasnih odložišč. Možno je sestaviti poljubno nepremično napravo s sestavljanjem osnovnih mest.

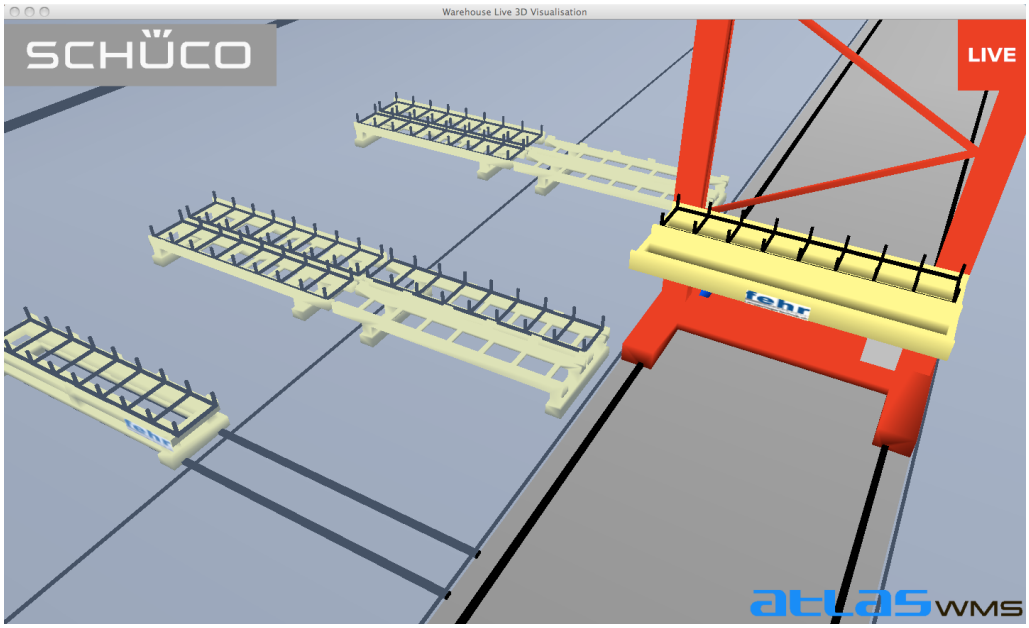
Podprto je prikazovanje vsebnikov (kaset). V skladiše je možno vstaviti poljuben statičen objekt, kot so table, ograje in podobno. Vizualizacija prikazuje vse premike naprav in kaset znotraj skladišča. Omogoča vizualiziranje večih skladišč hkrati. Regali so poenostavljeni transparentni kvadri, saj bi bilo vizualiziranje celotne matrike regalov prezahtevno za izrisovanje v realnem času in hkrati tudi nesmiselno, saj bi bil prikaz prenatrpan in bi zakrival pogled do naprav ter samega dogajanja. Podpira postavitev več pogledov (kamer).

Interakcija je implementirana prek tipkovnice, in sicer premikanje po prostoru, preklapljanje med pogledi in preklapljanje med skladišči. Na ekranu so prikazani logo podjetja lastnika skladišča, logo informacijskega sistema in logo *Live* oz. *V živo*, kadar je aplikacija uspešno povezana na podatkovno bazo.

Na sliki 1.4 je zajet posnetek zaslona aplikacije iz perspektive, ki zajema celotno skladišče. Na sliki 1.5 je zajet posnetek zaslona iz perspektive, ki prikazuje samo del skladišča, kjer je več dogajanja.



Slika 1.4: Posnetek zaslona aplikacije, ki zajema celotno skladišče.



Slika 1.5: Posnetek zaslona aplikacije, ki zajema del dogajanja.

1.6 Informacijski sistem AtlasWMS

Podjetje *Epilog d.o.o.*[1] je razvilo skladiščni informacijski sistem s prvotnim namenom podpore avtomatiziranih visokoregalnih skladišč za dolge materiale. Produkt je bil razvit v partnerstvu s švicarskim podjetjem *Fehr Lagerlogistik AG*[2], ki je eden največjih proizvajalcev tovrstnih skladišč v Evropi. Produkt je v več kot desetih letih razvoja postal še mnogo več kot to. Danes podpira tudi velika ročna skladišča, optimizacijo transportov, planiranje, optimiziranje žaganja, priklop na velik nabor raznoraznih avtomatskih naprav v skladiščih različnih proizvajalcev, kot so na primer avtomatizirane žagalne celice, avtomatska vakuumska dvigala itn.

Na sliki 1.6 vidimo zemljevid skladišč, ki jih upravlja AtlasWMS. Glavnina skladišč je v Švici in Nemčiji, nekaj skladišč pa je v Španiji, Franciji, na Finskem, Švedskem, Danskem, v Sloveniji, na Poljskem, v Turčiji in Veliki Britaniji.

Slika 1.6: Zemljevid skladišč, ki jih upravlja AtlasWMS

[1] http://www.epilog.net/
[2] http://www.lagerlogistik.ch/

2 Izbor tehnologije

2.1 Zahteve aplikacije

Aplikacija mora teči na operacijskih sistemih *Windows XP*, *OS X* in *Linux*. Vse nastavitve za skladišča morajo biti shranjene v konfiguracijski datoteki, kar bo tudi omogočilo hitro dodajanje novih skladišč. Dogajanje v skladišču lahko upravljajo različni tipi krmilnikov in tudi več njih hkrati. Na primer: za en del skladišča en krmilnik in za drug del skladišča drug. Na začetku morata biti dva tipa krmilnikov. *Demo* krmilnik, ki lahko prikazuje neko vnaprej določeno dogajanje. Tak krmilnik se lahko uporabi, kadar pri prezentaciji nimamo interneta. Prav tako se lahko uporabi za predstavitev neke vnaprej definirane situacije in za potrebe testiranja. *Live* krmilnik je namenjen za prikazovanje skladišča v realnem času z oddaljene lokacije. Dodatni možni krmilnik, ki bi bil lahko razvit v prihodnosti, je vizualiziranje skladišča na podlagi logov iz preteklosti, zato mora biti zasnova takoj zasnovana tako, da se ga bo dalo enostavno vstaviti. Zasnova aplikacije mora biti taka, da bo omogočala veliko odprtih možnosti za nadaljnji razvoj. 3D-modeli naprav morajo biti v enem od popularnih formatov, in sicer (*3ds* ali *obj*). Vizualizacija mora znati izpisovati teksture na 3D-modele in tudi direktno na ekran.

Aplikacija mora imeti tudi podporo za arhitekturo odjemalec - strežnik. Strežnik mora podatke brati iz vseh potrebnih podatkovnih baz različnih skladišč in jih posredovati do vseh trenutno odprtih instanc vizualizacije na različnih računalnikih. S tem bosta rešeni dve težavi. Dostop do podatkovnih baz avtomatiziranih skladišč ni javno dostopen, medtem ko mora biti sam strežnik aplikacije javno dostopen prek interneta, da se vizualizacija lahko poganja kjerkoli. Podatkovne baze skladišč so zato tudi minimalno obremenjene, če zaženemo veliko število klientov vizualizacije hkrati, saj server prebere zahtevane podatke samo enkrat v določenem intervalu in jih nato kopira in posreduje na zahtevo do vseh klientov. Vseeno mora imeti klient tudi možnost, da teče brez strežnika podatkov in lahko sam direktno dostopa do podatkovnih baz skladišč.

2.2 Opis izbranih tehnologij

Ker je bila ena od zahtev aplikacije, da mora teči na različnih platformah, je bila logična izbira programskega jezika *Java*. Prav tako je moje osebno poznavanje tega programskega jezika najboljše v primerjavi z vsemi ostalimi. Ker so bile zahteve po izrisovanju v realnem času, je bilo smiselno uporabiti neko eksterno javansko knjižico, ki podpira *OpenGL*[1]. *OpenGL* je v osnovi vmesnik med programi in grafičnimi karticami, ki je vse od uvedbe leta 1992 postal najbolj pogosto uporabljen ter podprt 2D- in 3D-grafični uporabniški programski vmesnik (API), s tem pa je prinesel na tisoče grafičinih aplikacij za široko paleto računalniških platform.[1]

Po drugi strani pa je bilo smiselno uporabiti eno od knjižic, ki se dejansko ukvarja z višjim nivojem procesiranja oziroma izrisovanja. Z drugimi besedami, knjižnica, ki podpira ukaze, s katerimi operiramo nad objekti, kot so svet, 3D-objekt, luč, tekstura in kamera. Operacije nad objekti pa so rotiranje, skaliranje, transformiranje in premikanje. Neposredno pa se ne ukvarjamo s samim izrisovanjem površin oziroma trikotnikov in nam ni treba programirati efektov, kot so osvetljevanje, senčenje in podobni.

Smiselno je bilo uporabiti eno od knjižic, ki je na višjem nivoju od uporabniškega programskega vmesnika okolja *OpenGL*. *Oracle* oziroma programski jezik Java podpira projekt *Java 3D*[2], ta pa ne temelji na *OpenGL* in je zato mnogo počasnejši ter tudi dokaj nizkonivojski. V času, začetka gradnje vizualizacije, sta bili na trgu dve glavni nizkonivojski ogrodji, ki sta temeljili na uporabniškem programskem vmesniku *OpenGL*.

[1] http://www.opengl.org/about/overview/
[2] http://java.sun.com/javase/technologies/desktop/java3d/

To sta *JOGL*[3] in *LWJGL*[4]. Nad njima pa je bilo izdelanih nekaj višjenivojskih ogrodij, med njimi najbolj razširjena *JPCT*[5] in *JMonkey*[6]. Obema je bilo takrat skupno, da sta bili v tem času še dokaj v povojih. Sama dokumentacija je bila slaba, prav tako pa je bilo zelo malo primerov, iz katerih se bi dalo izhajati. Če pa so že bili kakšni primeri, so bili ponavadi zelo osnovni ali pa so temeljili na kakšni igri in so bili za moj specifičen primer dokaj neuporabni. Odločil sem se za uporabo knjižice *JCPT*, saj je imel njen avtor zelo aktiven forum, s katerim sem si lahko veliko pomagal.

2.2.1 3D-pogon *JPCT*

JPCT knjižica podpira lastno razvito programsko izrisovanje in *OpenGL* izrisovanje s pomočjo knjižic *LWJGL* ali *JOGL*. Avtor se je odločil za možnost programskega izrisovanja, da je knjižico mogoče uporabiti tudi na napravah, ki nimajo grafične procesne enote (*GPU*). Knjižica ima vgrajen algoritem za zaznavanje trkov. Podpira nalaganje 3D-modelov formatov *3ds, obj, md2 in asc.* Možna je integracija v *Swing* ali *AWT*. Podpira točkovno osvetlitev in tudi ambientno, difuzno ter odsevno osvetlitev. Mogoče je vstavljati programsko primitivne oblike teles, kot so kocka, piramida, valj itn. Podpira enostavno ravnanje s pogledi oz. kamerami. Podpira efekt transparentnosti.

Naštejmo nekaj najpomembnejših razredov, ki sem jih uporabil.

- *FrameBuffer*, slika, v katero se izrisuje sama scena. *KeyMapper*, razred, preko katerega je možno implementirati interakcijo preko tipkovnice.
- *Config*, razred, preko katerega je možno nastaviti vse nastavitve samega pogona.
- Razreda *Texture* in *TextureManager* sta namenjena delu s teksturami.
- *World* razred je najpomembnejši del ogrodja, saj predstavlja samo sceno oz. svet, v katerega vstavljamo vse objekte in osvetlitve.
- Razred *Light* predstavlja entitetni razred za manipulacijo z osvetlitvami.
- Razred *Object3D* predstavlja posamezni 3D-objekt oz. model.
- Z razredom *Loader* je možno naložiti *Object3D* objekte iz datotek.

[3] https://jogamp.org/jogl/www/
[4] http://www.lwjgl.org/
[5] http://www.jpct.net/
[6] http://jmonkeyengine.com/

- *Camera* je entitetni razred, s katerim upravljamo s pogledom na sceno.
- *SimpleVector* predstavlja tridimenzionalni vektor, s katerim definiramo smer oz. orientacijo.
- Z razredom *Matrix* je možno izvajati bolj napredne operacije skaliranja, rotiranja in premikanja objektov.
- S pomočjo razreda *Primitives* pa je možno ustvariti poljubno enostavno telo, kot so kocka, krogla, piramida itd.

Avtor knjižice je Helge Förster, API knjižice pa se nahaja na naslovu *http://www.jpct.net/doc/*.

2.2.2 Scenski graf

Scenski graf je osnovna podatkovna struktura, ki se uporablja v vektorskem grafičnem urejanju. Predstavlja zbirko vozlišč v obliki drevesa ali grafa. Vsako vozlišče ima lahko veliko otrok, vendar po navadi samo enega starša. Če izvedemo operacijo nad določenim staršem, se ta operacija izvede nad vsemi otroki v drevesu. Vozlišča so grupirana in posamezne operacije izvajamo nad grupami.[2] Z uporabo knjižice *JPCT* scenski graf sestavimo tako, da naložimo posamezne objekte v razrede *Object3D*. Če je posamezni objekt sestavljen iz več različnih delov s katerimi želimo operirati posebej, jih shranimo vsakega v svoj razred *Object3D*. Te objekte med seboj povežemo v drevesno strukturo. Vsakega od objektov pa vstavimo v enega ali več razredov *World*. Z vstavljanjem v različne svetove lahko upravljamo, katere objekte izrisujemo v različnih situacijah. Na ta način lahko optimiramo izrisovanje, saj ogrodju ni treba ves čas preračunavati vseh objektov, ampak samo tiste, ki so v posameznem razredu *World*.

3 Izdelava 3D-modelov

3.1 Uporabljena orodja

3D-modeli objektov skladišča so bili zmodelirani z orodjem *3D Solid Works*[1]. Uporabljena je bila preizkusna verzija orodja. Orodje se je izkazalo kot odlično za izdelavo mehanskih strojnih objektov, saj je orodje namenjeno izdelavi modelov prav te vrste. Modeli so bili nato uvoženi in obdelani v orodju *Blender*[2], kjer so bile površinam objektov dodeljene teksture in materiali ter s tem sam videz. Modeli so bili nato izvoženi v formatu *3ds*.

3.2 Zgradba modelov

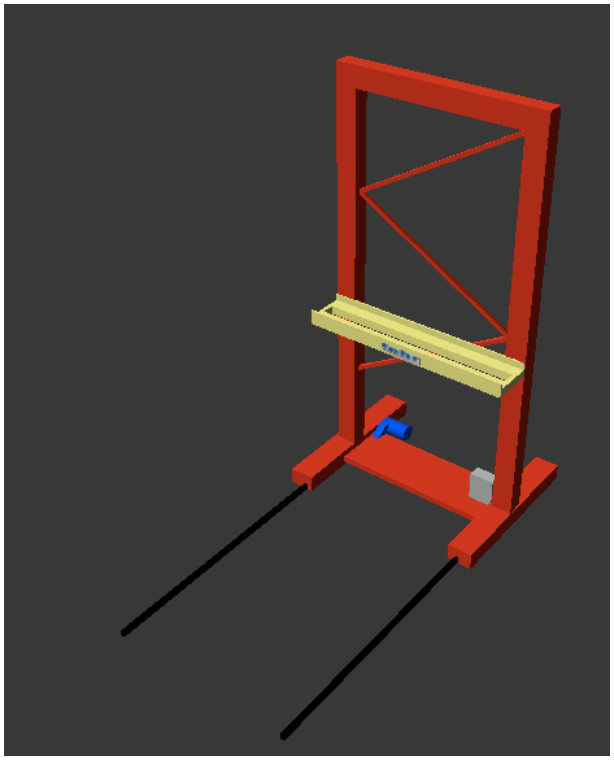
Osnovno načelo modeliranja objektov je grupiranje mrež (mesh-ov) za vsako gibljivo komponento naprave oziroma objekta posebej. Na primer: dvigalo z eno mizo (*RBGA* - Slika 3.1) ima tri za vizualizacijo zanimive komponente. Prva komponenta so statične tračnice, po katerih se dvigalo premika. Druga komponenta je sama konstrukcija oz. lok

[1] http://www.solidworks.com/sw/products/cad-software-3d-design.htm
[2] "http://www.blender.org/"

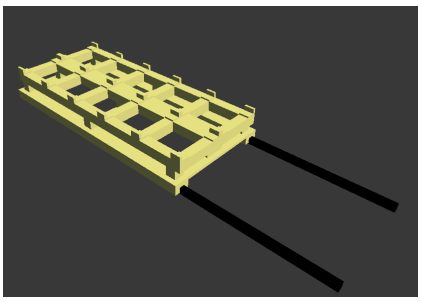
dvigala, ki se lahko premika horizontalno. Tretji del pa je miza, ki se premika vertikalno. Če bi želeli dodati še kakšno gibljivo komponento, bi jo zmodelirali in grupirali poligone posebej. Na primer: vlečna oprijemala mize, ki zagrabijo kaseto, preden jo premaknejo. Posamezni deli morajo biti pri modeliranju grupirani, ker jih kot take potem naložimo in operiramo z vsakim posebej znotraj 3D-ogrodja aplikacije.

Paziti je tudi treba, da naredimo čim več ravnih površin, s čimer se izognemo situaciji, kjer je posamezni objekt sestavljen iz velikega števila poligonov oziroma trikotnikov. Če izdelamo modele z veliko poligoni, potrebujemo bolj zmogljivo grafično kartico. Objekte moramo konsistentno postaviti v koordinatni prostor.

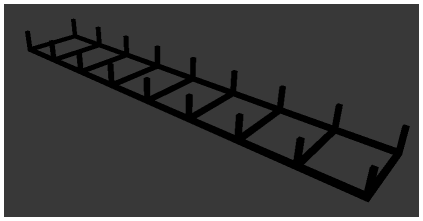
Na slikah 3.1, 3.2 in 3.3 vidimo nekaj modelov, izdelanih za potrebe aplikacije.



Slika 3.1: Kasetno dvigalo z eno mizo.



Slika 3.2: Tračni voziček z dvema mizama.



Slika 3.3: Kaseta za material.

3.3 Teksture

Ogrodje uporablja poleg materialov tudi teksture za izris površin modelov, zato je treba vse teksture, na katere se sklicujejo modeli, naložiti v ogrodje. Večina uporabljenih tekstur je enobarvnih, pri nekaterih pa sem v samo teksturo vstavil logo proizvajalca. Ogrodje zahteva, da so vse teksture dimenzij potence števila 2, na primer: 16 X 16, 128 X 128, 256 X 64.

Slika 3.4: Tekstura z logom proizvajalca

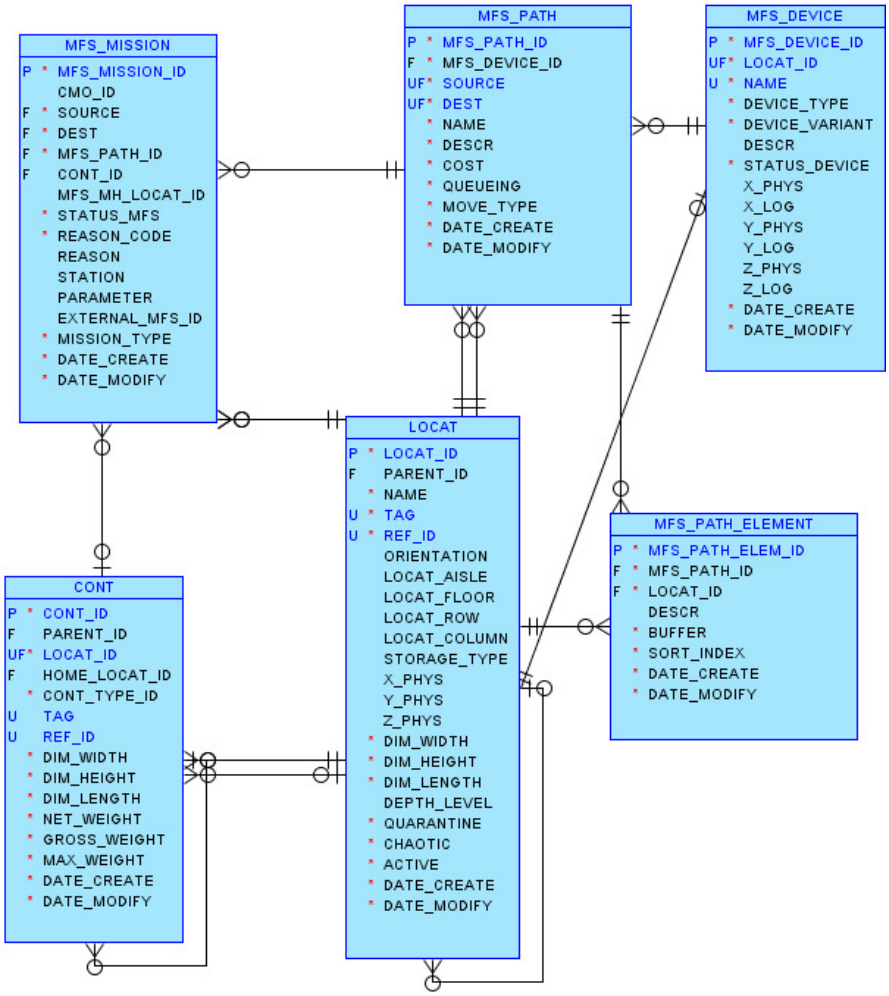
4 Zasnova programske kode

4.0.1 Podatkovni model

Na sliki 4.1 vidimo najpomembnejše tabele v podatkovni bazi, ki hranijo informacije, potrebne za vizualizacijo. To so *LOCAT*, ki hrani podatke o lokacijah, *CONT*, ki hrani podatke o kasetah in drugih vsebnikih, *MFS_DEVICE*, ki hrani podatke o napravah, *CMO*, ki hrani podatke o logističnih nalogih, *MFS_MISSION*, ki hrani podatke o logističnih nalogih znotraj posamezne naprave, *MFS_PATH*, ki hrani podatke o logističnih poteh, in *MFS_PATH_ELEMENT*, ki hrani podatke o posameznem elementu logistične poti.

4.0.2 Fizični entitetni elementi visokoregalnih skladišč

Osnovni entitetni elementi visokoregalnih skladišč, ki jih pozna in vodi informacijski sistem, imajo svojstveno politiko označevanja. Primer: krajša oblika oznake lokacije (*LOCAT*) v regalu bi bila *204010* oz. *(aabbcc)*. Znaka *aa* predstavljata desno oz. levo stran regala. *20* predstavlja levo stran, 40 pa desno. Zadnji del regalnega dvigala je nosilni lok dvigala, sprednji del pa je oz. so regalne mize. Na ta način lahko določimo, kako



Slika 4.1: Okrnjen podatkovni model aplikacije.

je skladišče obrnjeno in na kateri strani je leva oz. desna stran. Znaka *bb* predstavljata koordinato lokacije po osi X oz. kolono v matrici lokacij, znaka *cc* koordinato po osi Y oz. vrsto v matrici lokacij. Kasete (*CONT*) imajo identično označevanje, imajo pa ponavadi še predpono *C*, torej *C204010*.

Lokacije v skladišču so definirane v drevesni strukturi in razdeljene po tipih. Avtomatski del skladišča pozna dva tipa lokacij, in sicer lokacijo za shranjevanje (angl. *STORAGE*) in avtomatske lokacije naprav (angl. *CONVEYOR*). Kasete so prav tako različnih tipov, vendar sem se v vizualizaciji omejil in prikazujem vse kasete enako, čeprav so v resnici različnih višin, širin in tudi oblik. Posledično vizualizacija ni primerna za

odkrivanje napak v konfiguraciji informacijskega sistema.

Kasete in lokacije fizično niso označene. Te številke obstajajo le v informacijskem sistemu. Poznamo dva osnovna tipa visokoregalnih skladišč, *kaotično* in *nekaotično*. V kaotičnem skladišču dvigalo po opravljeni materialni operaciji kasete ne vrne nazaj na isto lokacijo v regalu, ampak jo po navadi da v najbližjo prosto lokacijo. V takih skladiščih so kasete močno pomešane in iz številke kasete ni moč razbrati lokacije, kjer naj bi bila. Velika večina skladišč je danes takšnih, ker so hitrejša, saj se pot, ki jo mora opraviti dvigalo, da prenaša kasete naokrog, zmanjša. V nekaotičnih skladiščih se kaseta po materialni operaciji vrača vedno nazaj na isto pozicijo. Taka konfiguracija predstavlja varnejšo obliko skladišča, saj v primeru napak hitro lahko lociramo, kje je kaj. Zaradi izredne zanesljivosti današnjih sistemov in prihrankov pri električni energiji ter času pri uporabi kaotičnih skladišč so ta uporabljena veliko pogosteje.

Naprave (*MFS_DEVICE*) so označene s šestimi znaki, na primer: *RBGB02* oz. *(aaabcc)*. Znaki *aaa* predstavljajo tip naprave, na primer regalno dvigalo, tračni voziček, hidravlično dvigalo, karosel ... Znak *b* predstavlja podtip naprave, na primer A, B, C, D ... Znaka *cc* predstavljata zaporedno številko naprave znotraj skladišča. Naprave so lahko v različnih statusih, in sicer: avtomatika, ugasnjena, ročni način, ročni način plus, sinhronizacija, napaka, brez povezave.

4.0.3 Logistične poti in nalogi

Logistični nalog (*CMO*) predstavlja eno zahtevo za premik kasete. Hrani podatke o kaseti, ki naj bo premaknjena, o startni in ciljni lokaciji, statusu in prioriteti. Logistični nalog je razdeljen na več logističnih nalogov znotraj posamezne naprave (*MFS_MISSION*). Na primer: kaseta *C202122*, ki je trenutno na lokaciji za shranjevanje v regalu, mora prispeti na komisionirno napravo *KARA01*. Logistični modul zgenerira najprej ukaz za regalno dvigalo, naj pobere kaseto s trenutne lokacije kasete na lokacijo mize *RBGA010001* regalnega dvigala, takoj za tem pa še ukaz, da to kaseto odloži na paritveno mesto ciljne naprave *KARA01*. Na koncu pošlje še ukaz napravi *KARA01*, naj premakne kaseto s paritvenega mesta *KARA010001* na materialno mesto *KARA010003*. Vsak od teh treh ukazov predstavlja logistični nalog znotraj posamezne naprave. Poleg kasete, naprave, startne in ciljne lokacije hrani še status, razlog, referenco do pripadajočega logističnega naloga in tudi referenco do logistične poti, na podlagi katere je bil zapis zgeneriran. Logistična pot (*MFS_PATH*) hrani zapise o vseh možnih poteh, na podlagi katerih lahko

modul za preračunavanje logističnih poti zgenerira nalog z lokacije *A* na lokacijo *B*. Vsak zapis hrani startno in ciljno lokacijo ter ceno premika. Vsak zapis logistične poti pa je sestavljen iz več elementov (*MFS_PATH_ELEMENT*), ki predstavljajo vrstni red lokacij, ki si sledijo na tej poti.

4.0.4 SQL poizvedbe

Vse potrebne poizvedbe so v *PL/SQL* paketu *app_vis3d_pkg*. Poizvedba 4.1 poišče vse kasete, ki so trenutno na napravah. To poizvedbo je treba izvesti posebej, da prikažemo tiste kasete, ki so že prispele na naprave, nimajo pa več nobenega naloga. Želene podatke je moč zajeti s pomočjo tabel *LOCAT, CONT in CONT_TYPE*, pri čemer nas zanima samo tip lokacije *CONVEYOR*

Druga poizvedba 4.2 vrača vse aktivne naloge, ki se trenutno izvajajo v skladišču. Na primer: naprava RBGA01 ima trenutno nalogo, da prestavi kaseto z lokacije 203021 v regalu na napravo KARA03 na mesto KARA030003. Ta nalog se bo izvedel s štirimi premiki kasete. Prvi premik bo prevzem (angl. *pick*) kasete z lokacije 203021 na dvigalo RBGA01 na mizo RBGA010001. Drugi premik bo oddaja (angl. *drop*) kasete z dvigala z mize RBGA0001 na napravo na prvo mesto KARA030001, tretji in četrti premik pa bosta sekvenčna premika kasete na mesti KARA030002 in KARA030003. Do želenih podatkov pridemo preko več tabel. Glavnina poizvedbe temelji nad tabelami *MFS_MISSION, MFS_PATH* in *MFS_PATH_ELEMENT*. V poizvedbi je pomemben del kode, ki preverja, ali je naslednja lokacija na logistični poti (*NEXT_MPE.SORT_INDEX = MPE.SORT_INDEX + 1*) prosta. Le v primeru, da je naslednja lokacija prosta, se premik začne izvajati, sicer je nalog v čakanju, dokler se lokacija ne sprosti. Te situacije se ne da razbrati direktno iz podatkov nalogov posamezne naprave, ker ima nalog enak status, kadar je naslednja lokacija prosta ali zasedena. Fizično naprava s pomočjo senzorjev sproži naslednji premik. Izvleček poizvedbe 4.3 prikazuje del, kjer se preverja pogoj, ali je lokacija, ki je naslednja v logistični poti, prosta oz. na njej ni nobene kasete (*CONT*)

Izvorna koda 4.1: Kasete na napravah

```
SELECT CONT.TAG AS CONT, LOCAT.TAG AS LOCAT,
            LT.CODE AS LOCAT_TYPE
  FROM CONT, LOCAT, LOCAT_TYPE LT
 WHERE CONT.LOCAT_ID = LOCAT.LOCAT_ID
```

```
       AND LOCAT.LOCAT_TYPE_ID = LT.LOCAT_TYPE_ID
       AND LT.CODE='CONVEYOR';
```

Izvorna koda 4.2: aktivni nalogi

```
SELECT MPE.MFS_PATH_ELEM_ID, CONT.TAG AS CONT, LOCAT.TAG AS SRC,
       NEXT_LOCAT.TAG AS DEST, MM.PARAMETER AS PARAM,
       LTS.CODE AS LOCAT_TYPE_SRC, LTD.CODE AS LOCAT_TYPE_DEST,
       MM.MFS_MISSION_ID, DEVICE.NAME AS DEVICE
  FROM MFS_MISSION MM, CONT, LOCAT, MFS_PATH MP, MFS_PATH_ELEMENT MPE,
       MFS_PATH_ELEMENT NEXT_MPE, LOCAT NEXT_LOCAT, LOCAT_TYPE LTS,
       LOCAT_TYPE LTD, MFS_DEVICE DEVICE
 WHERE MM.STATUS_MFS IN ('A','S')
   AND DEVICE.MFS_DEVICE_ID = MP.EXEC_DEV_ID
   AND LOCAT.LOCAT_TYPE_ID = LTS.LOCAT_TYPE_ID
   AND NEXT_LOCAT.LOCAT_TYPE_ID = LTD.LOCAT_TYPE_ID
   AND CONT.CONT_ID = MM.CONT_ID
   AND LOCAT.LOCAT_ID = CONT.LOCAT_ID
   AND MP.MFS_PATH_ID = MM.MFS_PATH_ID
   AND MP.MFS_PATH_ID = MPE.MFS_PATH_ID
   AND LOCAT.LOCAT_ID = MPE.LOCAT_ID
   AND NEXT_MPE.MFS_PATH_ID = MP.MFS_PATH_ID
   AND NEXT_MPE.SORT_INDEX = MPE.SORT_INDEX + 1
   AND NEXT_LOCAT.LOCAT_ID = NEXT_MPE.LOCAT_ID
   AND NOT EXISTS (SELECT '1'
                     FROM CONT
                    WHERE CONT.LOCAT_ID = NEXT_LOCAT.LOCAT_ID)
```

Izvorna koda 4.3: Del izvorne kode 4.2

```
   AND NOT EXISTS (SELECT '1'
                     FROM CONT
                    WHERE CONT.LOCAT_ID = NEXT_LOCAT.LOCAT_ID)
```

Obstaja še tretja, mnogo bolj kompleksna poizvedba, ki pa se uporablja samo v določenih primerih, ko podatki v podatkovni bazi sploh niso zadostni za pravilno zaznavanje možnih premikov. Takrat v posebno tabelo *VIS3D_MOVE_COND* naložimo še nekaj metapodatkov. S pomočjo teh metapodatkov poizvedba vrne premike, ki jih osnovna poizvedba ne zazna.

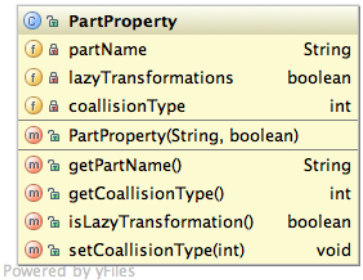
4.1 Nalaganje, manipuliranje in hranjenje objektov v grafičnem pogonu

Grafični pogon *JPCT* podpira nalaganje modelov iz datotek v pomnilnik programa preko razreda *Loader*. Nalaganje iz datoteke je razmeroma počasno, medtem ko je aktivacija prikaza modela, ki je že prednaložen v pomnilniku, zelo hitra. Datoteko mora grafični pogon najprej razčleniti in preformirati v objekt v pomnilniku, do katerega potem lahko hitro dostopa. 3D-objekti so sestavljeni iz več grupiranih poddelov, do katerih moramo dostopati posebej.

Za nalaganje 3D-modelov sem izdelal 3 razrede, ki skrbijo za operacije, potrebne za nalaganje modelov in tekstur.

4.1.1 PartProperty

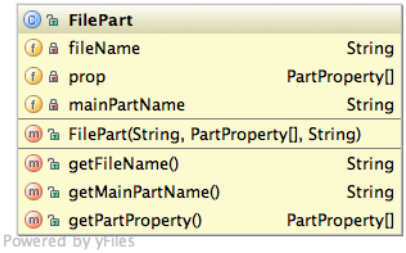
Razred *PartProperty*, katerega struktura je prikazana na sliki 4.2, je entitetni razred, ki vsebuje lastnosti posameznih grupiranih poddelov 3D-modela, na primer tračnice regalnega dvigala. Vsebuje oznako posameznega grupiranega poddela - *partName*. Je statičen oz. premičen - *lazyTransformations*. S tem poljem povemo ogrodju JPCT, da mu ni treba izvajati kalkulacij nad tem objektom. Vsebuje tu način, kako se bodo obravnavali trki med poddelom in kamero - *coallisionType*. Če se s kamero oz. pogledom približamo objektu, lahko nadaljujemo skozi njega ali pa okoli njega. Na primer: če se približamo regalu, želimo nadaljevati skozenj, medtem ko če se približamo neki napravi, tega ne želimo, saj bi potem videli napravo od znotraj, zato nadaljujemo okrog nje.



Slika 4.2: Zgradba entitetnega razreda *PartProperty.java*.

4.1.2 FilePart

Razred *FilePart*, katerega struktura je prikazana na sliki 4.3, je entitetni razred 3D-modela, ki pa vsebuje lastnosti za celoten 3D-model, na primer regalno dvigalo. Vsebuje ime datoteke, iz katere naj se naloži 3D-model - *fileName*, ime glavnega dela modela - *mainPartName* in tabelo z lastnostmi za vsak grupiran poddel modela, posebej opisanim z entitetnim razredom *PartProperty*. Razreda *FilePart* in *PartProperty* sta torej v razmerju 1 : N. Vsak objekt vizualizacije je sestavljen iz več poddelov. Eden izmed poddelov glavnega dela je glavni del, ki pa je lahko tudi celoten model, če je edini. Te informacije posredujemo razredu *ModelLoader*, da lahko pravilno naloži oz. razčleni datoteko *3ds*.



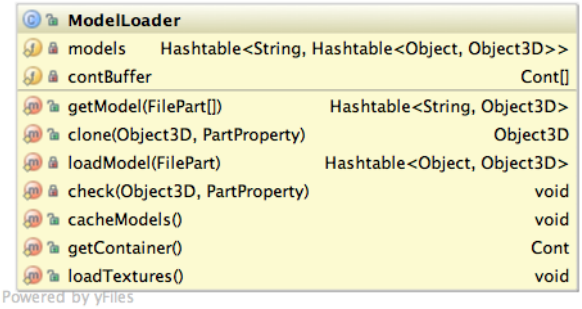
Slika 4.3: Zgradba entitetnega razreda *FilePart.java*.

4.1.3 ModelLoader

Razred *ModelLoader*, katerega struktura je prikazana na sliki 4.4, je namenjen branju 3D-modelov in tekstur iz datotek. Zbirka zbirk (*Hashtable of Hashtable of Object3D*) *models* je neke vrste predpomnilnik, ki hrani kopije seznamov poddelov objekta, ki so že bili zahtevani vsaj enkrat. Ti se razčlenijo in shranijo iz datoteke v pomnilnik preko metode *loadModel*.

Izvorna koda 4.4: ModelLoader.getModel()

```
public static Hashtable<String, Object3D> getModel(FilePart[] files, float
    scaleFactor)
{
   Hashtable<String, Object3D> cloneParts =
      new Hashtable<String, Object3D>();
   for (FilePart file : files)
   {
      if (models.get(file.getFileName()) == null)
```



Slika 4.4: Zgradba razreda *ModelLoader.java*.

```
        {
            models.put(file.getFileName(), loadModel(file, scaleFactor));
        }
        Hashtable<Object, Object3D> parts = models.get(file.getFileName());
        for (int j = 0; j < file.getPartProperty().length; j++)
        {
            PartProperty prop = file.getPartProperty()[j];
            Object3D part = parts.get(prop.getPartName());
            Object3D clone = clone(part, prop);
            cloneParts.put(prop.getPartName(), clone);
        }
    }
    return cloneParts;
}
```

Ko zahtevamo določen model prek metode *getModel*, ki je zapisana v izvorni kodi 4.4, se najprej zgradi objekt, če ta še ne obstaja, nato pa se posreduje dvojnik preko metode *clone* zahtevanega objekta. Gre za tipično implementacijo ustvarjalnega programskega vzorca za načrtovanje z imenom prototip (angl. *prototype*). Prototip vzorec se nanaša na ustvarjanje dvojnika, kadar želimo izboljšati performance. V tem primeru zagotavlja eno najboljših poti za ustvarjanje objekta. Ta vzorec se uporablja, ko je neposredno ustvarjanje določenega objekta drago.[3] V tem primeru je razčlenjevanje in nalaganje 3D-objekta iz datoteke draga operacija. Zato enkrat shranimo v predpomnilniku objekt in pri vsaki naslednji zahtevi vrnemo dvojnik objekta. Vsak 3D-model se naloži ob prvi uporabi v pomnilnik, kjer se ves čas hrani. Na ta način pohitrimo nalaganje, prav tako pa optimiziramo rabo pomnilnika. Dvojniki objektov si delijo tudi isto mrežo (angl. *mesh*).

Izvorna koda 4.5: ModelLoader.loadModel()

```
private static Hashtable<Object, Object3D> loadModel(FilePart filePart){
   Hashtable<Object, Object3D> parts = new Hashtable<Object, Object3D>();
   Object3D[] objectParts = Loader.load3DS("3ds" + Vis3D.FILE_SEPARATOR +
       filePart.getFileName(), 1);
   Object3D model = new Object3D(0);
   for (Object3D part : objectParts){
      part.rotateX((float) -Math.PI / 2);
      part.rotateMesh();
      part.setRotationMatrix(new Matrix());
      boolean found = false;
      for (int j = 0; j < filePart.getPartProperty().length; j++){
         PartProperty property = filePart.getPartProperty()[j];
         if (property.equals(filePart.getMainPartName()))
            continue;
         if (part.getName().startsWith(property.getPartName())){
            part.createTriangleStrips();
            parts.put(property.getPartName(), part);
            found = true;
            break;
         }
      }
      if (!found)
         model = Object3D.mergeObjects(model, part);
   }
   model.createTriangleStrips();
   parts.put(filePart.getMainPartName(), model);
   return parts;
}
```

Metoda *loadModel*, ki je zapisana v izvorni kodi 4.5, prebere in razčleni model iz datoteke tipa 3ds, ki mora biti v podmapi 3ds. Vse dele grupira po imenih, kot je definirano v entitetnem razredu FilePart 4.3. Če za kakšen del ni definicije, ga pridruži k glavnemu delu modela. Vsak del nato posebej shrani v tabelo hash, ki jo metoda tudi vrača. Preko te tabele hash lahko potem dostopamo do posameznih delov modela.

Niz *contBuffer* hrani N število objektov *Cont* (kaset). Kasete se na začetku prek metode *cacheModels* naložijo v pomnilnik in so na voljo kateri koli sceni oz. skladišču, da jih uporabi oz. zasede. Če skladišče zavzame kaseto, na njej postavi zastavico *used*

na *true* in jo s tem rezervira. Ko skladišče kaseto sprosti, spet postavi zastavico na *false*. Metoda *getContainer* vrača prvo nezasedeno kaseto iz niza *contBuffer*. Na ta način imamo ves čas v pomnilniku alocirane kasete, kar posledično pomeni, da se lahko hitro dodajajo in odstranjujejo iz skladišč.

Izvorna koda 4.6: ModelLoader.loadTextures()

```
public static void loadTextures() throws IOException
{
   TextureManager texMan = TextureManager.getInstance();
   File dir = new File("textures");
   String[] children = dir.list();
   for (String fileName : children){
      if (fileName.toUpperCase().endsWith("JPG"))
         texMan.addTexture(fileName.toUpperCase(), new Texture("textures" +
             Vis3D.FILE_SEPARATOR + fileName));
   }
}
```

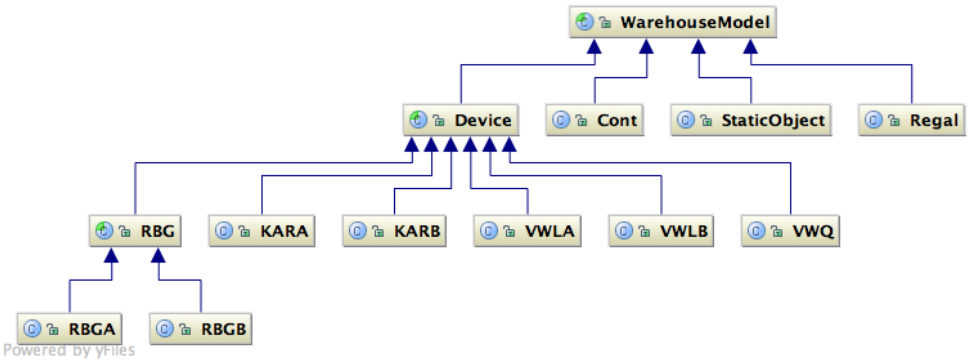
Metoda *loadTextures*, ki je zapisana v izvorni kodi 4.6, naloži vse teksture iz datotek v pomnilnik oz. razred *TextureManager*. Prebere vse datoteke tipa jpg. Teksture se nato uporabljajo na samih 3D-modelih ali jih izrisujemo direktno na zaslon. Omejitve pri teksturah so v tem, da morata biti višina in širina potenca števila 2. V nasprotnem jih grafični pogon avtomatsko poveča do najbližje dovoljene vrednosti, kar povzroči popačenje in zaradi tega so nesorazmerne glede na višino in širino.

Izvorna koda 4.7: Vis3DUtil.scaleImage()

```
public static Image scaleImage(BufferedImage image, double factorW,
    double factorH)
{
   BufferedImage scaled = new BufferedImage(image.getWidth(), image.
       getHeight(), BufferedImage.TYPE_INT_ARGB);
   int width = (int) (factorW * image.getWidth());
   int height = (int) (factorH * image.getHeight());
   Image img = image.getScaledInstance(width, height, BufferedImage.
       SCALE_SMOOTH);
   scaled.getGraphics().drawImage(img, 0, 0, width, height, null);
   return scaled;
}
```

Teksture za prikaz logov na ekranu so pripravljene za resolucijo HD 1080 p. Če uporabljamo drugačno resolucijo, se te avtomatsko prilagodijo s pomočjo metode *scaleImage*, ki je zapisana v izvorni kodi 4.7. Metoda ustrezno zmanjša dimenzije loga, ne da bi spremenila dimenzijo same teksture. To je rešeno tako, da najprej naredi prazno teksturo oz. sliko s transparentnim ozadjem ustreznih dimenzij, nato pa v desni zgornji kot vstavi zmanjšan logo, prilagojen trenutni resoluciji prikaza.

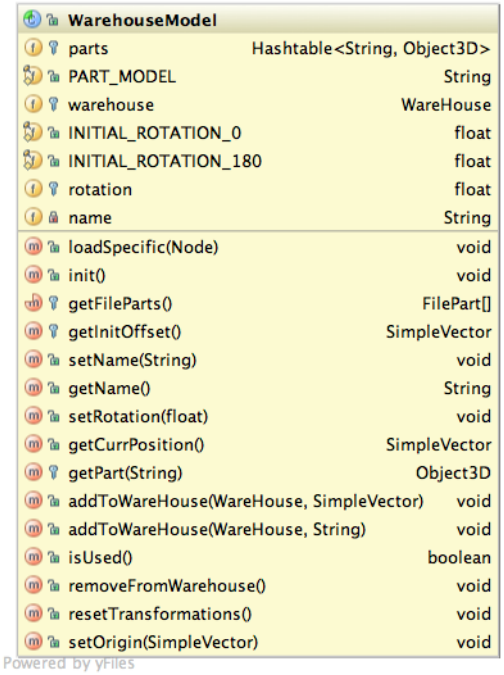
4.2 Entitetni razredi 3D-modelov skladišča



Slika 4.5: UML-diagram entitetnih razredov skladišča.

Entitetni razredi elementov skladišča so drevesno organizirani oz. dedovani tako, kot prikazuje slika 4.5. Osnovni razred, iz katerega so dedovani vsi ostali, je abstraktni razred *WarehouseModel*. Struktura razreda je prikazana na sliki 4.6. Vsebuje tabelo vseh posameznih poddelov modela skladišča - *parts*, do katerih se lahko dostopa prek metode *getPart*, hrani ime objekta - *name*, rotacijo - *rotation*, referenco skladišča, v katerem je - *warehouse*, vsebuje metode za dodeljevanje ali odstranjevanje skladišču - *addToWarehouse* in *removeFromWarehouse*, metodo za izračun trenutne pozicije v skladišču - *getCurrPosition*, metodo za setiranje začetne pozicije - *setOrigin*, metodo za vrnitev elementa na začetno stanje - *resetTransfomrations* ter metodo *loadSpecific*, ki iz konfiguracijske datoteke *xml* prebere sebi specifične parametre.

Na drugem nivoju v drevesni strukturi elementov na sliki 4.5 so razredi *Device*, *Cont*, *StaticObject* in Regal. *Regal* je transparentni kvader s poudarjenimi robovi, ki predstavlja matrično konstrukcijo lokacij, kamor se shranjujejo kasete. *StaticObject* je namenjen za



Slika 4.6: Zgradba entitetnega razreda *WarehouseModel.java*.

generično vstavljanje poljubnega statičnega objekta, s katerim ne bomo nič manipulirali, razen vstavili na neko fiksno lokacijo. S tem razredom lahko vstavimo objekte, kot so table, ograje ipd. Razred *Cont* predstavlja kasete v skladišču.

Razred *Device* je osnovni razred, iz katerega so podedovane vse naprave skladišč. To so regalna dvigala - *RBGA* in *RBGB* -, tračni vozički - *VWLA*, *VWLB* in *VWQ* - krožni tekoči trakovi - *KARA* in *KARB*. Regalno dvigalo je dedovano še na en nivo nižje, in sicer iz povsem praktičnega razloga, ker je 80% funkcionalnosti obeh dvigal identičnih. Osnovne funkcionalnosti, ki jih razred *Device* pokriva, so dodeljevanje hitrosti premikanja po oseh X, Y in Z, dodajanje in odstranjevanje kaset ter osnovno premikanje naprav.

Združevanje posameznih delov znotraj grafičnega pogona je rešeno z relacijo otrok-starš. Ko dodajamo poddele posameznega modela v sceno oz. v skladišče, jih povežemo v drevesno strukturo. Ko naredimo posamezno operacijo nad nekim delom, se ta operacija izvede tudi nad vsemi poddeli, ki so povezani nižje v drevesu. To je zelo uporabna funkcionalnost, saj na ta način lahko veliko laže premikamo, rotiramo in skaliramo 3D-telesa. Na primer: kot je prikazano v izvorni kodi 4.8, je RBG-dvigalo sestavljeno iz treh

poddelov, tračnice *PART_TRACKS*, loka *PART_BOW* ter mize *PART_TABLE*. Najprej se postavijo v sceno *warehosue* tračnice, ki so statične. Kot otrok se tračnicam doda lok in nato miza loku. Ko ob štartu pozicioniramo RBG-dvigalo v skladišče, je dovolj, da pozicioniramo tračnice, s čimer se bosta pravilno pozicionirala tudi lok in miza. Dvigalo se horizontalno premika po tračnicah. Zato premikamo lok, s čimer se bo hkrati premikala tudi miza. Miza se edina premika vertikalno. Ker nima nobenega otroka, ko premikamo mizo, ne premikamo z njo ničesar drugega.

Ko objekt postavimo v sceno in pokličemo metodo *build*, ogrodje na novo preračuna rotacijski pivot in center objekta. To delovanje je tukaj nezaželeno, zato rotacijski pivot in center objekta shranimo v spremenljivki in ju po klicu metode *build* ponovno nastavimo na prvotne vrednosti.

Izvorna koda 4.8: RBG.addToWareHouse()

```
public void addToWareHouse(WareHouse warehouse, SimpleVector origin){
   super.addToWareHouse(warehouse, origin);
   Object3D partTracks = getPart(PART_TRACKS);
   Object3D partBow = getPart(PART_BOW);
   Object3D partTable = getPart(PART_TABLE);
   partTracks.setOrigin(origin);
   partBow.setOrigin(origin);
   partTable.setOrigin(origin);
   warehouse.addObject(partTable);
   warehouse.addObject(partTracks);
   warehouse.addObject(partBow);
   partTracks.addChild(partBow);
   partBow.addChild(partTable);
   SimpleVector rotPivot = partTracks.getRotationPivot();
   SimpleVector center = partTracks.getCenter();
   partTracks.build();
   partBow.build();
   partTable.build();
   partTracks.setCenter(center);
   partTracks.setRotationPivot(rotPivot);
   partBow.setCenter(center);
   partBow.setRotationPivot(rotPivot);
}
```

4.3 Entitetni podatkovni razredi

4.3.1 MoveContMission

Najpomembnejši entitetni podatkovni razred je *MoveContMission*, ki vsebuje vse potrebne informacije o posameznih premikih kaset v skladišču. V osnovi gre za preslikavo podatkov, ki so v tabeli *MFS_MISSION* s specifičnimi podatki, ki jih potrebuje in preračunava vizualizacija za lastne potrebe. Ta razred direktno uporablja scena oz. skladišče. Vsak premik v skladišču je določen z logističnimi nalogi.

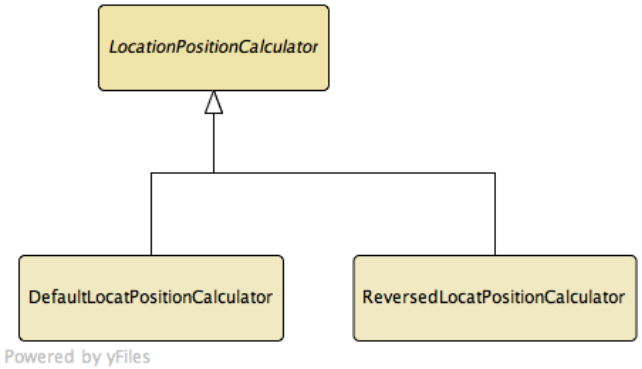
Premik je lahko treh vrst. Lahko je običajen premik v okviru ene naprave - *move*. Drugi je prevzemni premik - *pick*, kar pomeni, da mora naprava kaseto prevzeti, bodisi z neke lokacije v regalu bodisi z neke druge naprave. Tretji tip je odpremni premik - *drop*, kar pomeni, da mora naprava odložiti kaseto v regal ali na drugo napravo. Razred vsebuje tudi informacije, kot so oznaka naprave, oznaka kasete, startna točka, ciljna točka, trenutna točka, identifikator logističnega naloga, eventualne naprave, s katerimi mora komunicirati pri predajah (prevzem ali odprema) kaset, ter ostali potrebni meta podatki.

MissionBuffer je ovojni razred do niza, ki vsebuje vse trenutno aktivne premike za vse naprave v enem skladišču. Vsako skladišče oz. scena ima eno instanco razreda *MissionBuffer*, iz katerega posamezne naprave dostopajo do nalogov, ki so jim namenjeni. Vsaka naprava ima v vrsti lahko več nalogov za premik. Na primer: regalno dvigalo takoj prejme dva naloga, enega za prevzem kasete in drugega za odpremo kasete.

4.3.2 LogicalLocation

LogicalLocation vsebuje podatke o eni lokaciji. To so: oznaka lokacije oz. alias, tip lokacije (imamo dva tipa lokacije, lokacija v regalu *storage* in lokacija na napravi *conveyor*) ter vektor oz. točka, ki jo ta lokacija predstavlja v koordinatnem sistemu scene.

Na sliki 4.7 je prikazana UML-struktura razredov za izračunavanje oz. generiranje razredov *LogicalLocation* na podlagi fizičnih lokacij v podatkovni bazi. Povsod v kodi se uporablja abstraktni razred *LocationPositionCalculator*, na katerega delovanje pa vedno lahko vplivamo s specifično implementacijo.



Slika 4.7: UML-graf kalkulatorjev za izračun točke lokacije.

4.3.3 ContData in MissionData

Razreda *ContData* in *MissionData* sta enostavna objekta *bean* in se uporabljata za hranjene v javansko obliko transformirane oblike podatkov iz podatkovne baze. Podatke v tej obliki dobi krmilnik, ki jih nato transformira v obliko, primerno za posamezno sceno - *MoveContMission*.

4.4 Krmilni razredi

Gibanje objektov oz. dogajanje v skladiščih je izvedeno s pomočjo krmilnih razredov. Zgradba programske kode je zasnovana tako, da lahko skladišče krmili tudi več krmilnih razredov hkrati. Na primer: dvigala krmili en krmilni razred, premike viličarjev pa drugi.

Izdelana sta dva različna krmilnika. Prvi je *DemoWarehouseController* - demo krmilnik, ki prikazuje neko vnaprej definirano situacijo. S pomočjo tega krmilnika je moč prikazati neko zanimivo situacijo, lahko pa se uporabi za namene testiranja aplikacije.

Drugi veliko bolj zanimiv krmilnik je *LiveWarehouseController*. Ta krmilnik se povezuje na produkcijsko bazo skladišča in zajema žive podatke. Na podlagi teh podatkov prikazuje trenutno dogajanje v skladišču. Podatke zajema preko poizvedb *SQL* približno vsaki 2 sekundi. Poizvedbe se izvajajo povprečno pol sekunde, krmilnik pa še dodatno spi približno sekundo in pol. Bolj pogost zajem podatkov ni smiseln, saj se ne spreminjajo tako pogosto in tudi ne želimo preveč obremenjevati produkcijske podatkovne baze. Poizvedbi *SQL* vračata dva bloka podatkov. Prva vrača vse kasete, ki so na napravah, druga pa vse aktivne premike.

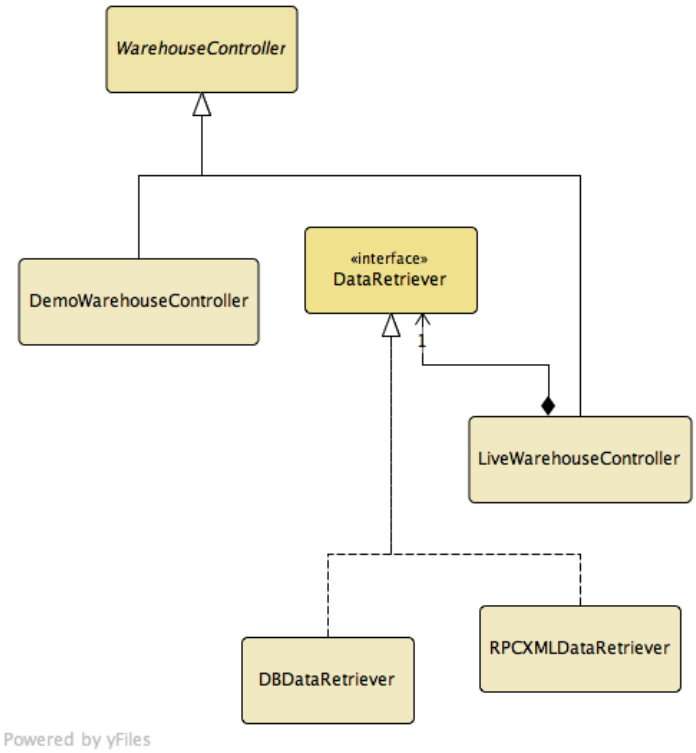
V prvem poskusu izdelave krmilnika so bile kasete, ki so že na napravah, pozicionirane po skladišču samo ob inicializaciji scene in so bile nato premikane izključno prek nalogov, vendar se je izkazalo, da to ni pravi način, saj so v avtomatiziranem skladišču možni tudi določeni ročni posegi. Na primer: v primeru mehanske ali kakšne druge okvare ali napake delavec prestavi neko kaseto ročno z dvigala v regal. Sama naprava je obveščena o tem premiku, kajti delavec mora obvezno v informacijskem sistemu označiti take premike, vendar se za tak premik ne kreira nalog. Podobna situacija se zgodi, če izgubimo povezavo za nekaj časa in ko se povezava vrne, so se kasete lahko že premaknile drugam. Take in podobe situacije so me prisilile, da sem zadevo zastavil tako, da se v vsaki iteraciji preberejo tudi vse kasete, ki so že na napravah in nimajo nobenega naloga. V primeru odstopanj se kasete v vizualizaciji sinhronizirajo z realnim stanjem, in sicer tako, da kaseta enostavno v trenutku skoči z enega mesta na drugega. Take situacije se po navadi dogajajo dokaj poredko, zato to ni moteče.

Krmilnik, ki zajema podatke iz produkcijskega sistema, bi teoretično lahko bil tudi obremenilen za podatkovno bazo, predvsem če poganjamo veliko instanc vizualizacije hkrati. V izogib temu problemu in hkrati tudi zaradi dejstva, da so podatkovne baze takih skladišč v privatnih omrežjih podjetij in zato nedostopne, sem ta krmilnik razbil na več delov. Za zajem podatkov skrbi poseben razred *DataRetriever*, ki pa ima dve izpeljanki. Prva je, da podatke zajema direktno kar sam klient, druga pa, da podatke zajema prek podatkovnega strežnika *WebVis3DServer*.

Slika 4.8 prikazuje arhitekturno zasnovo programske kode krmilnikov. Podatkovni strežnik teče na oddaljeni lokaciji znotraj podjetja Epilog d.o.o., ki je avtor informacijskega sistema skladišč in njegov vzdrževalec, kar pomeni, da ima tudi urejen dostop do teh podatkovnih baz. Strežnik zajema podatke in jih na zahtevo klientov posreduje poljubnemu številu instanc vizualizacije. Ko neka instanca vizualizacije zahteva podatke za neko skladišče, jih strežnik začne zbirati prek razreda *DataServer* in jih hrani ter osvežuje lokalno v spominu. Klienti na vsake pol sekunde zahtevajo nove podatke in če so se ti v vmesnem času spremenili, jih strežnik posreduje. Če strežnik že vsaj tri minute ni dobil zahteve po podatkih za določeno skladišče, tudi sam preneha poganjati poizvedbi *SQL* na produkcijski podatkovni bazi, dokler vnovič ne dobi zahteve po teh podatkih.

Zgradba aplikacije je odprta tudi za izdelavo novih krmilnikov, kot so vizualiziranje situacije na podlagi logov naprav iz preteklosti ali na primer simulacijski krmilnik, preko katerega bi lahko povezali kakšno simulacijsko orodje, s katerim bi merili pretočnost

skladišča in iskali ozka grla.



Slika 4.8: UML-graf kotrolnih razredov.

4.5 Konfiguracija in nastavitve

Ena od zahtev aplikacije je enostavno in hitro dodajanje novih skladišč. Če imamo v programski kodi že definirane oz. skodirane vse naprave, je dodajanje takega novega skladišča le stvar konfiguracije. Če pa nam manjka še kakšna naprava, je treba sprogramirati samo en nov razred, ki bo vseboval logiko te nove manjkajoče naprave. Celotna konfiguracija skladišč in prav tako ostale globalne nastavitve vizualizacije so definirane v datoteki *config.xml*. Izvorna koda 4.9 predstavlja primer te datoteke z definicijo enega skladišča. V datoteko je možno dodati N poljubnih skladišč. Omejeni smo le z velikostjo pomnilnika, ki ga bodo posledično zasedli modeli v skladiščih. Datoteka se prebere in razčleni s pomočjo razreda *WarehouseXMLLoader*. Za razčlenjevanje *config.xml* datoteke je uporabljen parser *DOM*.

V datoteki *config.xml* so shranjene nastavitve, dimenzije izrisa - *screen width* in *screen height*, hitrost izrisovanja (število slik na sekundo) - *screen fps*, katerakoli nastavitev iz razreda *com.threed.jpct.Config*, kot so v izvorni kodi 4.9 uporabljene - *fadeoutLight* - bledenje svetlobe glede na oddaljenost od izvora, *linearDiv* - intenzivnost bledenja svetlobe glede na razdaljo, *lightDiscardDistance* - maksimalni domet svetlobe, *farPlane* - maksimalna oddaljenost objektov od kamere, da se ti še izrisujejo, *collideOffset* - oddaljenost kamere od objekta, kjer se začne preračunavati detekcija trkov, *maxPolysVisible* - maksimalno število poligonov, ki se izrisujejo v posamezni sceni, *glFullscreen* - zagon aplikacije v celozaslonskem načinu.

Dimenzije in število izrisov na sekundo lahko nastavimo v konfiguracijski datoteki. To je pomembno, da lahko zadevo prilagodimo glede na strojno opremo, ki nam je na voljo.

Vozlišča *warehouse* so korenska vozlišča posameznega skladišča. Za posamezno skladišče lahko definiramo *controller*: krmilnik in specifične nastavitve zanj (v priloženi izvorni kodi 4.9 so definirani *LiveWarehouseController* z dobaviteljem podatkov server - klient (*data_retriever = RPCXMLDataRetriever*)), *logoName* - logo datoteka, ki se izrisuje v desni zgornji kot aplikacije, *locat_position_calculator* - razred, ki preračunava fizične koordinate elementov skladišča v koordinatni sistem vizualizacije, *light* - točkovne luči (lokacija in itenziteta), *camera* - poglede oz. kamere (lokacija in smer).

Vozlišče *object* je korensko vozlišče statičnega objekta. V izvorni kodi 4.9 sta definirana dva regala (*REGAL01-L* levi in *REGAL01-R* desni). Za posamezni objekt lahko definiramo: *class* - razred, *name* - ime objekta, *position_type* - tip pozicioniranja (poznamo dva tipa *ABSOLUTE* absolutno pozicioniranje, kjer podamo koordinate *BY_LOCAT* glede na ime lokacije, na podlagi katere *locat_position_calculator* preračuna absolutne koordinate) ter *custom_property* - dodatne lastnosti, ki so specifične za posamezni tip statičnega objekta.

Vozlišče *device* je korensko vozlišče naprave. Za posamezno napravo lahko definiramo popolnoma enake nastavitve kot za vozlišče *object*. Dodatno lahko definiramo še *orientation* - orientacijo in *rotation* - rotacijo. Orientacija ima dve vrednosti, in sicer *NORMAL_ORIENTATION* - normalno in *MIRROR_ORIENTATION* - zrcalno. Če gre za zrcalno orientacijo, je izhodno mesto na napravi vhodno in vhodno-izhodno. Rotacija je lahko *0* ali *180*. Če nastavimo vrednost *180*, se naprava obrne za 180 stopinj.

Izvorna koda 4.9: Izvleček konfiguracijske datoteke config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
  <screen width="1920" height="1080" fps="35"/>
  <config name="fadeoutLight" value="false"/>
  <config name="linearDiv" value="500"/>
  <config name="lightDiscardDistance" value="25000"/>
  <config name="farPlane" value="20000"/>
  <config name="collideOffset" value="150"/>
  <config name="maxPolysVisible" value="30000"/>
  <config name="glFullscreen" value="true"/>
  <property name="logoName" value="ATLASWMS.GIF"/>
  <warehouse name="INOR">
    <controller class="net.epilog.vis3d.controller.LiveWarehouseController">
      <data_retriever
        class="net.epilog.vis3d.controller.RPCXMLDataRetriever"
        connName="VIS3D-INOR"
        serverURL="http://devel.epilog.net:10099/xmlrpc">
      </data_retriever>
    </controller>
    <property name="logoName" value="INOX.GIF"/>
    <locat_position_calculator class="net.epilog.vis3d.controller.calc.
        ReversedLocatPositionCalculator"/>
    <object class="Regal" name="REGAL01-L" position_type="BY_LOCAT" location
        ="01200100010001">
      <custom_property name="regalHeigth" value="27"/>
      <custom_property name="regalLength" value="29"/>
    </object>
    <object class="Regal" name="REGAL01-R" position_type="BY_LOCAT" location
        ="01400100010001">
      <custom_property name="regalHeigth" value="27"/>
      <custom_property name="regalLength" value="29"/>
    </object>
    <device class="RBGA" name="RBGA01" rotation="180" position_type="
        ABSOLUTE" xPos="3400" yPos="-20" zPos="500" speedFactorX="1.3">
      <custom_property name="trackLength" value="34"/>
      <custom_property name="rbgHigth" value="27"/>
    </device>
    <device class="KARB" name="KARB01" orientation="NORMAL_ORIENTATION"
        position_type="BY_LOCAT" location="01200100260001"/>
    <device class="KARA" name="KARA02" orientation="NORMAL_ORIENTATION"
```

```
        position_type="BY_LOCAT" location="01200100200001"/>
    <device class="KARA" name="KARA03" orientation="MIRROR_ORIENTATION"
        rotation="180" position_type="BY_LOCAT" location="01400100260001"/>
    <light itensity="38" xPos="-1500" yPos="-2000" zPos="1500"/>
    <light itensity="28" xPos="5000" yPos="-2000" zPos="1500"/>
    <camera>
      <position x="4657.3247" y="-1346.6215" z="-654.9646"/>
      <direction x="-0.8707916" y="0.38020563" z="0.3117141"/>
    </camera>
    <camera>
      <position x="3638.4824" y="-955.1207" z="590.7592"/>
      <direction x="-0.7897637" y="0.46511623" z="-0.39992523"/>
    </camera>
  </warehouse>
```

4.6 Renderiranje in animacija

Renderiranje je realizirano v glavnem razredu *Vis3D*, ki ima tudi metodo *main*. Renderiranje je izvedeno s statičnim številom izrisov na sekundo. Da dosežemo točno število izrisov na sekundo, po izrisu vsake slike postavimo nit v spanje za določen čas.

V enačbi

$$\frac{1000000000}{F} - T_1 = T_2 \tag{4.1}$$

je T_1 dejanski čas izrisa v nanosekundah, F zahtevano število izrisov na sekundo, T_2 pa čas spanja niti v nanosekundah.

Po sami začetni postavitvi objektov v sceno je animacija implementirana z izključno enim ukazom v samem ogrodju in to je *translate* - premakni. Ni bilo niti potrebe po rotiranju. Algoritem objekte nekaj desetkrat v sekundi premika v želeni smeri za določeno število točk, s čimer dosežemo efekt premikanja teles v prostoru.

Za bolj realističen videz animacije sem vgradil efekta počasnejši začetek in počasnejši konec (angl. *slow in slow out*). Gibanje človeškega telesa in tudi večine drugih predmetov potrebuje na začetku in na koncu čas, da pospeši in upočasni. Iz tega razloga je animacija videti bolj realistično, če se objekti na začetku in na koncu premikajo počasneje kot v sredini premika.[4] Tudi v resnici se naprave v skladišču tako premikajo. Dvigalo začne s premikom počasi in nato prvih nekaj metrov linearno pospešuje, dokler ne doseže maksimalne hitrosti, nato pa večino poti opravi z maksimalno hitrostjo in potem zadnjih

nekaj metrov linearno upočasnjuje, dokler se ne ustavi na želeni točki.

5 Ugotovitve

5.1 Sklepne ugotovitve

3D-vizualiziranje razmeroma enostavnih realnih sistemov v Javi se je izkazalo po eni strani kot dokaj lahko izvedljivo, pri čemer ne potrebujemo zelo globokih matematičnih poznavanj samih principov renderiranja in vizualiziranja. Zadošča nam že grobo poznavanje principov. To nam omogočajo predvsem objektne javanske knjižice, ki v obliki tako imenovanih črnih škatel skrijejo kompleksne izračune same vizualizacije. Navzven pa ponujajo programerju enostavnejše operacije, ki so nekako logične že tudi laično usposobljenemu programerju. Sam se namreč nikoli nisem posebej izobraževal o tej temi.

Po drugi strani pa je izdelava vseeno zelo težavna, saj je bilo v času, ko sem izdeloval program, zelo malo dokumentacije in primerov, s katerimi bi si lahko pomagal. Zato je šlo tukaj v veliki meri tudi za raziskovalno nalogo, kjer je bilo potrebnega veliko zamudnega preizkušanja, preden je bilo možno priti do želenih rezultatov.

V sami nalogi sem se ukvarjal tudi z drugimi problemi, ki pa niso bili vezani na sam problem, kot je recimo sinhronizacija niti in dostopov do podatkov v zbirkah. Java kot programski jezik ima nekaj slabosti, ki pri rešavanju take naloge zadevo še dodatno

otežijo. Aplikacija se v komercialne namene ni pokazala kot zanimiva po funkcionalnosti, saj upravljanje skladišča prek 3D-grafike ni najbolj enostaven način. Aplikacija se je dobro izkazala kot predstavitev dejanskega fizičnega produkta visokoregalnega skladišča in je bila kot taka koriščena na večih logističnih sejmih po Evropi. Prav tako je aplikacija v uporabi kot dekorativni prezentacijski element v podjetju Epilog d.o.o., kjer ves čas teče na velikem LCD-zaslonu v sejni sobi.

5.2 Nadaljnji razvoj aplikacije

Za nadaljnji razvoj je še veliko prostora, prav tako je samo ogrodje mogoče ponovno uporabiti za podobne projekte. Vprašanje pa je v sami smotrnosti nadaljnjega razvoja s komercialnega stališča.

V letošnjem letu, je predviden razvoj na funkcionalnosti prikaza vožnje viličarjev po ročnih delih skladišča. Ta funkcija bo implementirana s pomočjo tehnologije avstrijskega podjetja *Zeno Track GmbH*[1], ki s pomočjo računalniškega vida v primerno opremljenem skladišču preračunava točne koordinate viličarjev in s pomočjo senzorja na vilicah viličarja sporoča, ali ima viličar kaj naloženo. Viličarji so opremljeni s kamerami, na vsakih nekaj metrov pa so v skladišču na tleh nameščene oznake.

[1] http://www.zenotrack.com/

Literatura

[1] Richard S Wright, Jr. et al., *OpenGL SuperBible: Comprehensive Tutorial and Reference*. Addison-Wesley Professional, 5th Edition, 2010.

[2] Allen Sherrod, *Data Structures and Algorithms for Game Developers*. Charles River Media, 1st Edition, 2007.

[3] Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1st Edition, 1994.

[4] Isaac Kerlow, *The Art of 3D Computer Animation and Effects*. Wiley, 4th Revised and enlarged Edition, 2009.