Universiteit Leiden
Opleiding Informatica

Ontology Viewer

Van proof-of-concept naar layered software

Name: Patric Stout
Studentnr: S0403431
Date: 27/08/2015
Supervisor: Fons Verbeek

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Inhoudsopgave

1 Introductie

1.1 Ontologie

Een ontologie is een beschrijving van objecten en hun onderlinge relaties. Denk hierbij bijvoorbeeld aan relaties zoals “is kind van” en “is verwant tot”. Zo’n collectie van informatie wordt bijvoorbeeld opgeslagen in het Web Ontology Language (OWL) formaat. Hierin wordt op een generieke manier opgeslagen welke objecten welke relatie met elkaar hebben.
Omdat een enkel object meerdere relaties kan hebben met een ander object, is dit niet een enkele graaf. Het is meer vergelijkbaar met een multidimensionale graaf, waar afhankelijk van je dimensie je een andere soort graaf ziet. Zo kun je geïnteresseerd zijn in de “parent/children” relatie, of in de “sibling” relatie, of “compares to” relatie, etc. Door deze verschillende manieren naar dezelfde informatie te kijken, ontstaat een wild bos aan informatie. Waar het OWL formaat dat probeert te standaardiseren in een opslag formaat, zijn er ook applicaties die dat proberen te weergeven. De zogenoemde “semantic reasoners” kunnen zulke bestanden inlezen, en daar een zekere reasoning over doen.
Een veel gebruikte programma omtrent het verwerken van zulke bestanden is bijvoorbeeld Protégé. Deze semantische editor staat toe een ontologie te maken en te bekijken. Via semantische reasoners kan hier dan weer van meerdere kanten naar dezelfde informatie gekeken worden.

1.2 Visualisatie

Alhoewel Protégé grotendeels een tekstuele aanpak voor het bekijken van de informatie heeft, is er ook onderzoek hoe zulke complexe informatie op een visuele manier getoond kan worden. In het proefschrift van Jullie B. Dmitrieva, Aspects of Ontology Visualization and Integration wordt er op meerdere manieren van visualisatie in gegaan, zoals bijvoorbeeld via een Poincare (op een 2D disk het plaatsen van punten) of containment (op een bol plaatsen van nodes, waar de kinderen in de cirkel van de node zitten waar ze toe behoren).

1.3 Doelstelling

Dit werk is gebaseerd op het boven genoemde proefschrift. In dit proefschrift wordt er bezig gehouden met het zoeken en vinden van visualisatie methodes. Deze zijn geschreven in Java, waar de nadruk vooral ligt op het experimenteren van verschillende methoden. In dit paper gaan we deze software verder uitbreiden met een gestructureerde software aanpak, waarbij we proberen een generiek stuk software te maken waar gemakkelijk verschillende visualisatie methoden voor geschreven kunnen worden, en gemakkelijk geschakeld kan worden tussen dezen. De nadruk ligt hierbij op goed onderhoudbare software, die makkelijk te begrijpen is en uitgebreid kan worden. Ook ligt ter doelstelling dat de software over meerdere jaren heen bruikbaar is en blijft.

2 Software ontwerp

2.1 Layers

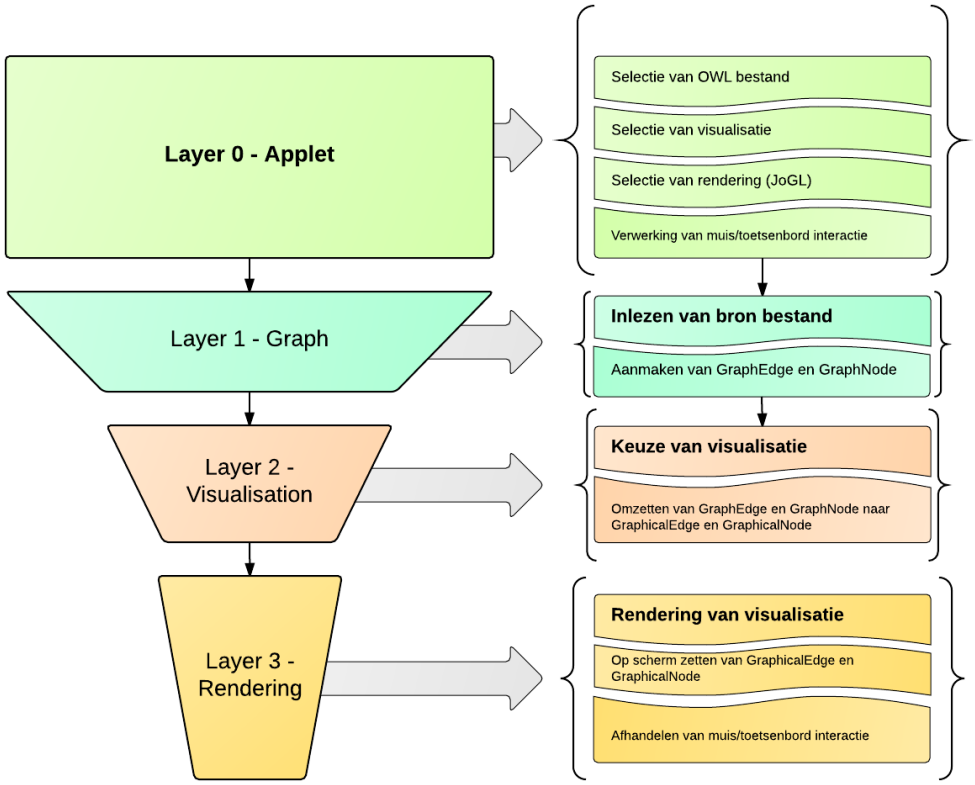
De ontologie viewer kan in meerdere layers worden opgedeeld. Elk van deze layers heeft één specifieke functie. Binnen een layer kunnen meerdere implementaties bestaan waar vrij tussen geschakeld kan worden afhankelijk van de input van buitenaf (bijvoorbeeld het bestand-formaat, of de gebruiker die een andere visualisatie kiest). Voor deze implementatie is gekozen voor een drie-tal layers, zoals weergeven in figuur 1.

1. Graph
   Het uitlezen van een ontologie formaat naar een graaf. Onderdeel hiervan is de reasoning van de ontologie, waardoor er een DCG (Directed Cycle Graph) over blijft. In alle visualisaties wordt het circulaire gedeelte van de graaf weergeven met een stippellijn of volledig weggelaten. Kortom, vanaf een visualisatie oogpunt kunnen we praten over een DAG (Directed Acyclic Graph).

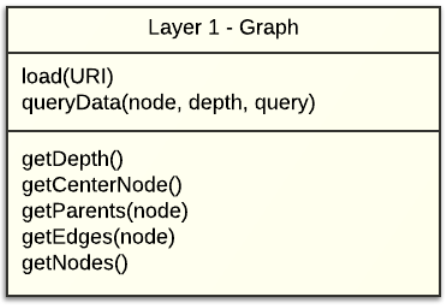
2. Visualisation
   De verschillende manieren om een DCG op het scherm te zetten. Bijvoorbeeld via een poincare graph of door surface packing op een bol.

3. Rendering
   De rendering engine om alles op het scherm te krijgen. Dit zullen we doen met JoGL, een 3D Library rondom OpenGL.

Door de introductie van software layers kun je een strict policy opstellen waar layer violations niet zijn toegestaan. Normaal gesproken loopt de informatie van een hogere layer terug naar een lagere, waar informatie niet de andere kant op stroomt, behalve op voor afgesproken plekken. Deze manier van communicatie wordt vastgelegd in een API, die per layer gedefinieerd is. Toegang van de ene layer naar de andere buiten deze API is niet toegestaan.
Het inlezen van informatie gebeurt in Layer 1. Het enige waar deze layer toegang tot heeft is de buitenwereld (bijvoorbeeld een bestand om in te lezen). Vervolgens geeft hij de DCG door naar de volgende layer, waar deze gevisualiseerd wordt. Denk hierbij aan het zetten van x/y coördinaten van elke node in de graaf. Als laatste komt deze aangepaste DCG aan bij de rendering layer, die de visualisatie op het scherm zet. Deze laatste laag doet (en kan) niets met de eerste laag, en informatie gaat altijd van hogere lagen naar lagere; nooit andersom.
Door deze strikte ondersplitsing van informatie is het erg gemakkelijk lokaal te experimenteren met, bijvoorbeeld, een nieuwe visualisatie of rendering. Ook verhoogd het de leesbaarheid van de software, omdat duidelijk is hoe verschillende lagen met elkaar communiceren. Mocht een implementatie van een layer niet (goed) functioneren, kan er altijd voor een andere gekozen worden.
Hier bovenop zit nog een layer 0, die de Java Applet layer is. Deze zorgt ervoor



Figuur 1: De layers van de ontology viewer



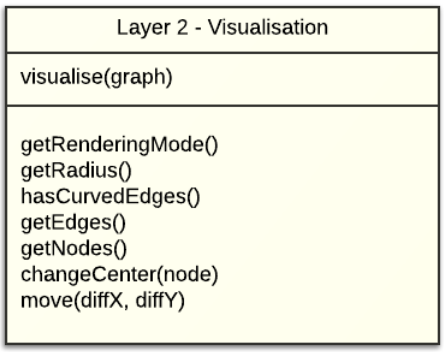
Figuur 2: Layer 1 - Graph

dat de applet opgestart kan worden en dat de layers aangeroepen worden. Layer 0 verzorgt de volgorde van aanroep tussen de layers en zorgt ervoor dat ze met de correcte instellingen aangeroepen worden.

2.1.1 Layer 1 - Graph

De Graph layer houdt zich bezig met het inlezen van een graaf, en zet deze om in een DCG. De API voor deze layer heeft twee kanten: een tweetal functies die aangeroepen worden vanaf Layer 0, en een 5-tal functies die worden aangeroepen van Layer 2.
Layer 0, de Applet zelf, kan een URI inladen en een query uitvoeren op de graaf. Deze worden omgezet naar GraphEdges en GraphNodes. Een GraphEdge is altijd een verbinding tussen twee GraphNodes, en kan van zichzelf informatie bevatten (zoals de relatie tussen de twee nodes, de naam, etc). Een GraphNode heeft enkel alleen een naam, en een indicatie of hij zelf nog kinderen heeft of niet.

In figuur 2 is de structuur van de API te zien. Het bovenste blok is de API voor layer 0, het onderste voor layer 2.

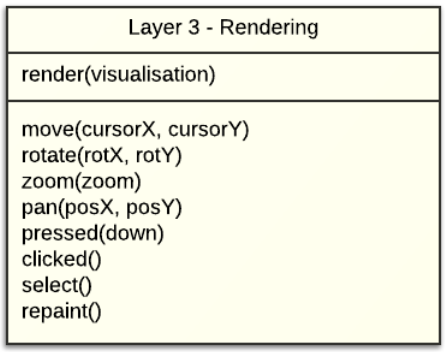


Figuur 3: Layer 2 - Visualisation

2.1.2 Layer 2 - Visualisation

De Visualisation layer houdt zich bezig met het verwerken van de DCG naar iets wat de Rendering layer op het scherm kan weergeven. Ook deze API heeft twee kanten: een enkele functie die vanaf Layer 0 kan worden aangeroepen, en 7 functies die vanaf Layer 3 kunnen worden aangeroepen.
Layer 0, de Applet, kan aangeven dat een bepaalde Layer 1 instantie gevisualiseerd moet worden. In dit proces worden de GraphEdges en GraphNodes van Layer 1 omgezet in GraphicalEdges en GraphicalNodes. Dit zijn dezelfde edges en nodes, maar met extra informatie, zoals de locatie op het scherm (relatief vanaf het midden), de label en het niveau.
Belangrijk in deze API is dat Layer 3 kan zeggen dat de node in het midden van de graaf veranderd is. In sommige visualisaties verandert hiermee ook gelijk de structuur van de visualisatie. Ook kan het verslepen van de visualisatie effect hebben waar nodes terecht komen. Aan de andere kant vertelt deze API aan Layer 3 wat voor een visualisatie hij is. getRenderMode() bijvoorbeeld kan terug geven of het om een 2D rendering gaat of om een 3D. In de API van deze layer zit dus alle belangrijke informatie over om wat voor visualisatie het gaat, en de Rendering layer pakt deze alleen op en vertaalt dit in een rendering.

In figuur 3 is de structuur van de API te zien. Het bovenste blok is de API voor layer 0, het onderste voor layer 3.



Figuur 4: Layer 3 - Rendering

2.1.3 Layer 3 - Rendering

De Rendering layer houdt zich bezig met het naar het scherm brengen van de visualisatie. Hij kan worden aangeroepen door Layer 0, of gebruikers interacties. Dit laatste gebeurd via een Dispatcher, die doorgeeft welke muis- en/of toetsenbordacties de gebruiker heeft gedaan. Aan de hand hiervan kan de rendering worden gedraaid, ingezoomd, etc.

In figuur 3 is de structuur van de API te zien. Het bovenste blok is de API voor layer 0, het onderste voor de dispatcher.

2.2 Modulair

Door de introductie van layers is de software sterk modulair. Geen enkele module hangt af van een andere module, niet in zijn eigen layer of in een andere layer. Dus ondanks dat er gewisseld kan worden van rendering engine, het zal niets veranderen aan de visualisatie implementaties. Maar ook is het erg gemakkelijk te schakelen tussen verschillende visualisatie methoden.
Uiteindelijk zal bij het uitvoeren van de applicatie uiteraard een keuze gemaakt moeten worden welke module uit welke layer gebruikt moet worden. Per layer moet er wel eentje gekozen worden voor dat de software correct kan functioneren.

2.3 Documentatie

Elke nieuwe functie in de API is voorzien van Javadoc. Javadoc is een standaard manier om van een functie of variabelen te beschrijven wat deze doet, welke parameters er verwacht worden, en wat zijn resultaat zal zijn. Door vervolgens Javadoc over de software heen te runnen, wordt er vol automatisch documentatie gemaakt in bijvoorbeeld HTML formaat. Dit maakt het navigeren binnen de code erg makkelijk, en geeft een goed overzicht voor andere mensen die bij-

voorbeeld een eigenlijk module willen maken voor een bepaalde laag.
Verder is het erg belangrijk om in de code commentaar toe te voegen waar de code van zichzelf niet triviaal uitlegt. Zo is het bijvoorbeeld overbodig om commentaar toe te voegen bij een statement zoals dit:

```
x += node.getX();
```

Het statement zelf legt prima uit wat hij doet. Daarentegen, als we naar dit voorbeeld kijken:

```
/* See how many edges we have; ignore edges to the parent */
int kids = edges.size();
for (GraphEdge edge : edges) {
    if (edge.getNodeTarget() == parent) {
        kids--;
    }
}
```

Zulke statements hebben zeer zeker documentatie nodig, anders zal het erg onduidelijk zijn waarom we er in bepaalde gevallen nog eentje van af te trekken.

De bovenste twee, in combinatie van zo klein mogelijke functie te gebruik, zorgt voor sterk leesbare software. Dit heeft als voordeel dat als andere mensen willen werken aan de software, alles duidelijk en vindbaar is.

3 Implementatie

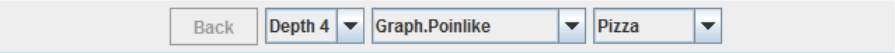
3.1 Voorgaand werk

Al het werk is gebaseerd op het proefschrift van Jullie B. Dmitrieva, Aspects of Ontology Visualization and Integration. Vooraf is goed onderzocht wat de code doet, om te kunnen begrijpen wat er nodig is. Omdat het proefschrift vooral proof-of-concept code beschrijft, is de code vooral gebouwd rond het concept: doen wat nodig is om het werkend te krijgen. Hiermee was het initiële vrij lastig om in beeld te krijgen wat nodig was en wat overbodig was.
Meerdere versie van hetzelfde bestand bestaan met kleine modificaties, om verschillende manieren van visualisatie te testen. Ongedocumenteerde waarden, code dat met commentaar uitgezet is, if-statements die nooit uitgevoerd kunnen worden, etc. Allemaal wat je verwacht met code die gebruikt wordt om te testen welke visualisatie methoden werken en welke niet.
Als 3D library wordt er gebruik gemaakt van Java3D. Helaas is Java3D niet langer ondersteund, en zal dus vervangen moeten worden. Als vervanger is gekozen voor JoGL. Maar net zoals met Java3D kan het zijn dat deze library in de toekomst niet meer onderhouden en/of ondersteund wordt. Dit is de reden dat layer 3 (rendering) bestaat.
Soms wordt er in 1 bestand meerdere visualisaties gedefinieerd, en soms met meerdere bestanden. Om hier 1 lijn in te trekken is layer 2 (visualisatie), die afdwingt dat 1 bestand 1 visualisatie geeft.
Hardcoded per visualisatie wordt een ontology ingeladen, en vaak op een net wat andere manier via de reasoner geïnterpreteerd. Dit geeft de duidelijke reden voor het bestaan van layer 1 (graph inlezen); het voorkomen dat we op meerdere plekken ongeveer hetzelfde doen, en dus visualisaties van net wat andere data kunnen krijgen.

3.2 Realisatie

Het maken van layer 1 (graph inlezen) is vrij vanzelfsprekend. De API definieert een aantal functies, zoals het inlezen van een bron bestand en het uitvoeren van een query op dit bestand. In het geval van OWL wordt er eerst een OWL bestand ingelezen, waarna met een query de reasoner aangestuurd wordt om daar een DCG van te maken.
Voor dit paper is gekozen om zowel het OWL formaat te ondersteunen met behulp van HermiT voor de reasoning, als CSV, wat een platte tekst representatie is van een DCG. Het laatst genoemde formaat ondersteunt geen reasoner.
Voor de 3D library is gekozen voor JoGL. Dit vooral omdat het op het moment van het schrijven de nummer één is voor 3D in Java. Het voorgaande werk was gemaakt met Java3D dat al een tijdje deprecated is, en nog lastig is aan de praat te krijgen en houden. Gelukkig was het vertalen van de statements vrij triviaal, omdat beide library GL “spreken”.
De API die wordt blootgesteld is vooral geconcentreerd op het interactieve gedeelte, met andere woorden, het gebruik van de muis en toetsenbord. De rende-

ring weet hoe hij visualisaties moet weergeven, en de centrale functie is daarom ook render(visualisation), die het weergeven van alles in 3D voor zijn rekening neemt.
Als laatste layer hebben we de visualisaties. De API definieert zich rond een enkele functie: visualise(graph), waarmee de gegeven graaf in nodes wordt opgedeeld die de rendering kan weergeven in de gevraagde modus. Na onderzoek blijkt de visualisatie zich in drie subcategorieën onder te verdelen: containement, graph en sphere. Graph is de makkelijkste: het gaat hier om disk, poincare en poinlike. Hier worden de nodes van de graaf op een 2D cirkel geplaatst. Deze cirkel is in 3 dimensies te roteren, waardoor er duidelijk in rondgekeken kan worden. Sphere doet hetzelfde als graph, maar dan op een bol. Hierbij wordt hyperbolic en stereographic ondersteunt. Als laatste, en de meeste belangrijke, is containement. Hier wordt door middel van semantische zoom informatie weergeven op een bol door middel van kleine cilinders. Het grootste verschil tussen de laatste en andere twee is de semantische zoom. De eerste twee laten altijd alle informatie zien, en kan door middel van zoomen dingen groter gemaakt worden. De laatste laat maar een selectie van de informatie zien, en door middel van zoomen kan daar meer of minder van zichtbaar worden gemaakt.
Het implementeren van de containment was verreweg het moeilijkste. Waar de andere twee aardig goed beschreven waren (zowel in het proefschrift als in de code), was de containment nog erg in test-fase. Het was ook erg lastig om het bestaande werk aan de praat te krijgen op een andere computer. Na veel proberen, is er uiteindelijk gekozen om alles van de grond af aan te schrijven, met als naslag het bestaande werk. Dit leidde gelukkig tot duidelijke resultaten en uiteindelijk in twee werkende versies: plain en scaled. In de eerste variant is zijn alle cirkels, ongeacht hun informatie, even groot. In de tweede variant is de cirkel bij geschaald aan de hand van de informatie die hij weer in zich heeft. Dit geeft dus direct visuele feedback over de informatie dichtheid.
Het plaatsen van de cirkels voor de containment methode is in het bijzonder erg lastig om goed te krijgen. Omdat we praten over een bol van 360 graden, waarbij boven op de bol kleinere halve bollen worden gelegd voor de visualisatie, is de wiskunde daarachter erg complex. Na vele implementaties, navragen, nazoeken en weggooien van code, is het eindelijk gelukt om een goede implementatie van de wiskunde te doen. Hierbij is dus wel gekozen de code zelf te implementeren, waardoor het resultaat afwijkt dan wat het origineel werk voor schreef.
Er is geprobeerd om zoveel mogelijk te blijven bij het originele werk, maar was het soms noodzakelijk af te wijken. Het lezen van OWL bestanden is volledig opnieuw geschreven, zo ook het gebruik van JoGL (tegenover Java3D). Visualisaties voor graph en sphere zijn nagenoeg 1 op 1 overgenomen, maar stevig voorzien van commentaar om de leesbaarheid te verhogen. Ook zijn ze gegoten in een API vorm, wat af en toe zorgde dat code op andere plekken terecht kwam dan origineel voorgesteld. In de meeste gevallen zorgt dit voor snellere code (iets wordt maar één keer uitgevoerd in plaats van elke keer), maar in enkele gevallen wordt code nu ook vaker uitgevoerd (omdat er minder gebruik gemaakt wordt van het “cachen” van waarden). Dit leidde echter niet tot meetbare negatieve prestaties, voornamelijk omdat JoGL veel sneller is dan Java3D.

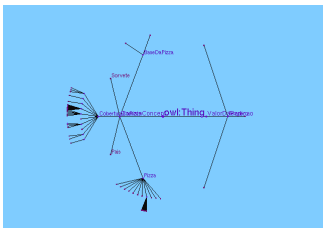


Figuur 5: De interface van de applicatie

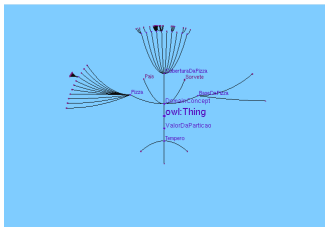
Voor containment is, zoals eerder aangegeven, afgeweken van het origineel werk. Veel code daarin bleek experimenteel, en soms zelfs ongefundeerd. Gelukkig is het resultaat vergelijkbaar, alhoewel subtiele interacties verloren zijn gegaan.

3.3 Interacties

De interactie van de applicatie is een mengeling van het werk van het proefschrift en interacties die voortvloeien uit de introductie van het layer systeem. In het voorgaande werk was het nodig om een andere Java Applet op te starten om bepaalde visualisaties te kiezen, en kon er niet geschakeld worden tussen deze vanuit hetzelfde programma. Omdat layer 2 bepaalde hoe de visualisatie gedaan wordt, is het vrij eenvoudig om dit mogelijk te maken in hetzelfde programma. Hierdoor kan er nu snel geschakeld worden tussen verschillende vormen van visualisatie zonder dat de applicatie opnieuw opgestart hoeft te worden. Hetzelfde geldt voor layer 1, waar de keuze gemaakt kan worden welke OWL bestand geopend wordt.
Om dit visueel mogelijk te maken is bovenaan de applicatie een balk toegevoegd waarin een aantal keuzes gemaakt kunnen worden. Zo kan gekozen worden tussen de diepte waarin deze door de reasoner wordt heen gehaald, de visualisatie die toegepast wordt, alsmede welk OWL bestand geopend moet worden. In figuur 5 zijn deze functionaliteiten van links naar rechts te zien zoals ze bovenin de applicatie staan.
Qua interacties binnen de visualisaties is geprobeerd zoveel mogelijk vast te houden aan de interacties zoals die in het proefschrift beschreven en geïmplementeerd waren, maar wel gelijk getrokken tussen alle visualisaties. Op een paar kleine uitzonderingen na, doet een muisbeweging hetzelfde voor alle visualisaties.
Voor alle visualisaties kan met de linker muisknop worden geroteerd, met de rechter muisknop worden rond bewogen en met de middelste muisknop worden ingezoomd. In de containment visualisatie wordt er bij inzoomen meer en meer details zichtbaar. Door in de graph visualisaties te dubbel klikken op nodes kan daarop gecentreerd worden. Door middel van de "Back"knop die dan actief wordt kan er terug worden gegaan naar de vorige node die geselecteerd was. Hierdoor is het eenvoudig mogelijk te navigeren door de boom aan informatie.
De nadruk op de veranderingen van de interacties lag op het makkelijker maken voor de gebruiker om te schakelen tussen visualisaties, zonder af te doen aan de interacties per visualisatie zoals die al bestonden.



Figuur 6: De Graph.Disk visualisatie



Figuur 7: De Graph.Poincare visualisatie

3.4 Visualisaties

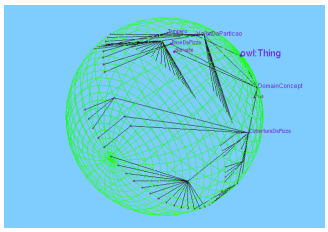
Zoals eerder beschreven worden drie categorieën aan visualisatie geïmplementeerd: graph, sphere en containment. Alle drie hebben ze andere visualisaties. Het onderzoek en uitwerkingen van de specifieke details van elke visualisatie worden uitgelegd in het proefschrift van Julia B. Dmitrieva; de termen die hier gebruikt worden zijn daar direct uit overgenomen.

3.4.1 Graph.Disk

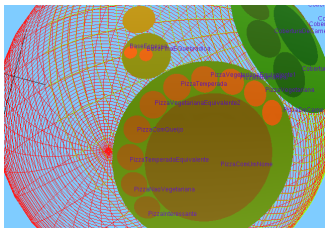
In figuur 6 is een voorbeeld van de eerste en simpelste visualisatie die in de applicatie zit. De hoofd node zit in het midden, zijn directe kinderen in een cirkel om hem heen, de kinderen daarvan in een grotere cirkel om de hoofd node heen, etc.

3.4.2 Graph.Poincare / Graph.poinlike

In figuur 7 is een voorbeeld van een ingewikkeldere visualisatie, waar geen vaste afstand wordt gebruikt van de ene cirkel naar de andere, maar een hyperbool. Door middel van de rechter muisknop kan over de hyperbool bewogen worden. Deze interactie is iets anders dan in de andere visualisaties, waar enkel alleen de visualisatie heen en weer wordt bewogen. In deze (en in poinlike) visualisatie worden de punten getransponeerd over de hyperbool.
De poinlike verschilt met de poincare voornamelijk in de spreiding van de punten over de gehele cirkel, maar is verder gelijkend opgezet.



Figuur 8: De Sphere.Steroegraphic visualisatie



Figuur 9: De Containement.Scaled visualisatie

3.4.3 Sphere.Stereographic / Sphere.Hyperbolic

In figuur 8 is een voorbeeld een sphere visualisatie. Hier zijn de punten niet op het platte vlak gelegd, maar op de oppervlakte van een bol. De stereographic doet dit op een enkele bol. Voor de hyperbolic ligt de hoofd node in het midden van de bol, en liggen de punten van zijn directe kinderen op een kleine bol daaromheen. De kinderen daarvan liggen weer op kleine bollen daaromheen, etc.

3.4.4 Containement.Plain / Containement.Scaled

In figuur 9 is een voorbeeld een containment visualisatie. In deze visualisatie liggen er schillen om een bol heen, die ieder een node aangeeft. Als er wordt ingezoomd worden de kinderen van een node als kleinere schillen neergelegd binnen de schil van de node zelf. Deze schillen zijn vanaf bovenaf gezien cirkels op de bol (een hemisphere als het ware). Dit is verre weg de meest complexe vorm van visualisatie die in deze applicatie zit.
Het verschil tussen plain en scaled is dat de schil per node in plain altijd even groot is ongeacht het aantal kinderen van een node, maar in scaled de grootte van een schil aangepast wordt aan de hoeveelheid die daar weer in zit.

3.5 Dependencies

De software maakt voor alle drie de layers gebruik van externe libraries:
Layer 1: voor het inlezen van het OWL formaat en om reasoning te doen, wordt er gebruik gemaakt van semanticweb en HermiT. Semanticweb leest het OWL bestand in, en HermiT doet de reasoning hierover. Bijgeleverd zijn de versies van beide libraries die werken. Upgraden zal geen probleem mogen zijn, maar tijd heeft geleerd dat hun API nog wel eens wil veranderen, en zal dus met aandacht gebeuren moeten.
Layer 2: om zelf geen vector math te hoeven implementeren, wordt er gebruik gemaakt van vecmath. Dit is een kleine library waar allerlei vector-related operaties in zitten, in een erg efficiënte manier. Dit scheelt veel code en fouten, en is een veel gebruikte library in de Java wereld.
Layer 3: voor rendering maken we gebruik van JoGL. Er zijn andere keuzes, zoals Java3D, maar omdat JoGL directe binds zijn om OpenGL leek dat de beste keus. Omdat de software layered en modulair is, is het vrij triviaal een andere library toe te voegen, mocht dit nodig zijn en/of worden. De versie van JoGL is vastgezet, omdat inmiddels al een nieuwere versie bestaat waarvan de API niet compatibel is met de huidige code. Wel wordt de library van externe servers binnen geladen, waardoor in de toekomst het mogelijk is dat deze niet meer bereikbaar zijn.
Verder is er geen gebruik gemaakt van externe libraries. Hoe minder externe libraries, hoe makkelijker het is software te onderhouden, en er is daarom ook bewust gekozen geen extra libraries toe te voegen indien het niet echt nodig zou zijn. Deze minimale set van libraries lijkt afdoende.

3.6 Java Applet

Voor de implementatie is gebruik gemaakt van een Java Applet. Deze heeft als voordeel dat hij via een webpagina uitgevoerd kan worden. Hierdoor is het mogelijk om via een browser het resultaat te bekijken. Dit is vrijwel altijd makkelijker dan via het downloaden van een Java applicatie.
Omdat deze Applet afhangt van een aantal externe libraries, wordt er gebruik gemaakt van jnlp. Dit is een methode die het mogelijk maakt voor een Java Applet om externe dependencies te downloaden. Hiermee hoeft dus niet zelf libraries als JoGL mee verscheept te worden, maar kunnen deze van een extern domein worden gedownload. Dit vormt alleen een probleem als het externe domein deze libraries verwijderd of vervangt.
Helaas, vooral door veel kritiek op Sun/Oracle voor het jaren negeren van veiligheid, zijn er de laatste paar versie erg veel restricties opgesteld voor wat een Java Applet wel en niet mag doen. Zo was het in Java 1.6 nog makkelijk mogelijk een bestand te downloaden van een ander domein, zo was dat in Java 1.7 nog mogelijk onder restricties te doen, zo is dat in Java 1.8 nagenoeg onmogelijk zonder een aantal security settings uit te zetten.
Alhoewel het volledig te begrijpen is dat deze restricties aan zijn gezet, het maakt het niet makkelijker voor de ontology viewer, waarvan je het liefst hebt

dat een gebruiker een web adres van zijn eigen OWL bestand kan invullen en hij daar een visualisatie van krijgt. Helaas zijn we dus genoodzaakt een vooropgesteld lijstje in de Applet te zetten met bestanden die op hetzelfde lokale domein staan als de Applet zelf.
Ook is het sinds Java 1.8 niet meer mogelijk om externe libraries te hebben die door een ander zijn gesigned dan je eigen library op hetzelfde domein. Ook is het nu verplicht je eigen library te signen (hetzij via een self-signed, maar het is verplicht). Dit maakt de publicatie van een applet erg omslachtig. Het is nu dus nodig om alle externe libraries te ontdoen van hun eigen signing (als aanwezig), en die opnieuw te signen met je eigen key. Maar om te zorgen dat de Applet nog werkt op Java 1.8 is hier wel voor gekozen. Bij nieuwere versies van Java zijn hier weer problemen mee te verwachten.
Meer en meer browsers weren Java Applets, door het erg lastig tot zelfs onmogelijk te maken deze uit te voeren. Het ligt ook in de lijn der verwachtingen dat over een paar jaar Java Applets helemaal weg zijn. Een goed ander alternatief voor dit project zou bijvoorbeeld WebGL zijn, waarmee in Javascript direct 3D renderingen te maken zijn. Dit echter staat nog in de kinderschoenen en heeft nog een lange weg te gaan om brede ondersteuning van alle browsers te krijgen. Voornamelijk is het probleem dat nog niet op alle Operating Systems gebruik gemaakt kan worden van 3D-acceleratie, waardoor WebGL erg langzaam kan zijn.

4 Discusie

4.1 Werk in de toekomst

In de huidige implementatie is het alleen mogelijk om OWL bestanden te lezen, met een standaard reasoner, en CSV bestanden. Het zal niet lastig zijn dit uit te breiden met bijvoorbeeld MySQL ondersteuning (of een andere database), of om andere ontology formaten te ondersteunen. Door de simpliciteit van deze layer en de modulariteit van de software kan deze toegevoegd worden en gebruikt worden daar waar nodig. Daarbij moet wel goed gekeken worden naar de beveiliging van Java 1.8, die verbindingen naar bijvoorbeeld MySQL niet zonder meer toestaat.
Het gebruiken van JoGL geeft de software de komende paar jaar nog wel up-to-date ondersteuning voor 3D rendering. Maar JoGL aan het begin van dit jaar een nieuwe versie uitgebracht waarvan de API niet compatibel is met de gebruikte versie. Het zal dus nodig zijn om naast de huidige implementatie ook een implementatie te maken die compatibel is met de nieuwe versie. Ook kan het voor performance redenen nuttig zijn om andere 3D libraries te proberen. Verder zijn de meeste 3D operaties niet geoptimaliseerd voor snelheid; zo wordt er geen gebruik gemaakt van pre-rendering en andere acceleratie mogelijkheden.
Bij het implementeren van de containment bleek er zich een probleem voor te doen. Layer 1 geeft een graaf terug met een maximale diepte (omdat de reasoner strikt gezien een oneindig diepe graaf kan terug geven). Voor graph en sphere visualisaties is dit precies wat je wil. Voor semantische zoom wil je als je verder inzoomt dat er meer informatie tevoorschijn komt. Dit betekent dat layer 2 dus andere informatie moet opvragen aan layer 1 afhankelijk van het zoom niveau. In de software gemaakt voor dit paper is dat niet meegenomen, maar voor verder werk wordt zeker geadviseerd hier naar te kijken. Zeker voor de scaled containment kan dit een aardige uitdaging zijn om goed op te lossen. Nu wordt de voorop ingestelde diepte meegenomen als maximale zoom van informatie.
Een van de voornaamste problemen met containment is resolutie van de bol. Het renderen van een 3D bol is eigenlijk niets wat het lijkt: een heleboel kleine driehoekjes worden aan elkaar geplakt om het gevoel van een rond object na te bootsen. Hoe hoger de resolutie van je bol, hoe kleiner de driehoekjes (en hoe meer er nodig zijn). Bovenop de grote bol wordt een kleine halve bol gelegd om de nodes aan te geven. Dit op dezelfde manier als hierboven, met kleine driehoekjes. Als de resolutie van beide bollen niet hoog genoeg is, snijden de bollen elkaar in plaats van dat ze boven op elkaar liggen. Maar als je de resolutie te hoog maakt, wordt de applicatie te zwaar voor de gemiddelde videokaart. Een balans tussen twee moest daarom gevonden worden, die niet altijd even goed werkt. In toekomst werk zal het gebruik van de glux routines om bollen te maken vervangen kunnen worden door een eigen algoritme die hier rekening mee kan houden, door bijvoorbeeld de resolutie van een halve bol hoger te maken dan de grote bol.

4.2 Portability

Omdat de applicatie geschreven is in Java met de gedachte beschikbaar te moeten zijn voor het web, is er gebruik gemaakt van libraries die werken op alle gangbare Operating Systems. Zo is JoGL er voor Windows, Linux en Mac. Hiermee werkt deze Java Applet op alle gangbare Operating Systems zonder problemen, getest met Java 1.6 tot en met Java 1.8. Deze keuze maakte de implementatie niet makkelijk. Zeker het ondersteunen van nieuwe versies die veelal tot nieuwe security settings leidde, heeft zo zijn nodige uren gekost.

4.3 Bekende problemen

Op dit moment is de Java Applet gesigned met een self-signed certificaat. Dit zorgt ervoor dat voor het openen van de Applet de URL waar deze zich onder bevindt toegevoegd moet worden aan de lijst met uitzonderingen. Zonder dat, is het niet mogelijk om de Applet op te starten. Dit kan vrij simpel opgelost worden door de Applet te signen met een geldig certificaat. Tot die tijd zal het openen van de Applet omslachtig blijven. Door hoe certificaten zijn opgebouwd zal het wel nodig zijn dat ieder jaar de Applet opnieuw gesigned wordt met een nieuw certificaat, omdat meestal elk jaar certificaten verlopen.
Containment visualisatie maakt gebruik van glut voor het opbouwen van de bollen. Voor iedere verandering aan de 3D omgeving wordt deze functie opnieuw aangeroepen; dit is een erg trage manier, waardoor de animatie ook erg traag kan aanvoelen. Met ondersteuning van goede 3D-acceleratie wordt dit probleem grotendeels weggenomen, maar meestal werkt de 3D-acceleratie slecht (of niet) via de Java Applet. Binnen OpenGL zijn hier oplossingen voor, maar daar is geen gebruik van gemaakt.
Door het toevoegen van lichtbronnen is de applicatie trager. Het uitzetten van lichtbronnen maakt 3D animatie aanzienlijk sneller, maar je verliest daarbij ook direct het gevoel van perspectief. Daarom is gekozen lichtbronnen aan te zetten, ten koste van de animatie snelheid.
Teksten printen in OpenGL is vrij omslachtig. Deze worden achteraf als 2D tekst op de rendering gezet. Hierdoor zitten de labels nog wel eens in de weg. Een oplossing hiervoor zou zijn gebruik te maken van 3D fonts.
Containment methoden gebruiken semantische zoom, maar alleen tot het niveau aangegeven met 'depth'. Hogere depths echter maken het voorbereiden van de visualisatie aanzienlijk trager. Hier moet de gebruiker zelf dus een afweging in maken.

5 Voor de ontwikkelaar

In onderstaande hoofdstukken bespreken we een aantal belangrijke zaken voor het ontwikkelen met deze software. De meeste punten staan ook beknopt benoemd in de README en we zullen daar hier wat dieper op in gaan.

5.1 Ontwikkelen in Eclipse

Als je wilt ontwikkelen aan dit project via Eclipse, hoef je alleen maar het Eclipse project te openen. Om de Applet te starten via Eclipse moet je in de code een bestandsnaam opgeven van een OWL bestand om die te zien. Dit is omdat via de normale manier deze worden meegegeven als "paramïn de jnlp opstart bestand. Via Eclipse wordt deze niet gebruikt, en moet je zelf even opgeven waar hij een bestand kan vinden. Dit kun je doen in Applet.java, rond regel 274.
Ook is het belangrijk voor je begint te zorgen dat de JoGL lib/native bestanden in de map staan. Per Operating System zijn dit andere bestanden, omdat het native bestanden zijn (DLL, so, dylib, ..). Onder lib/native staan per Operating System in een map de correcte libraries. Kopieer degene die nodig zijn naar lib/native; JoGL kan deze daar vinden. Dit hoef je niet te doen als je via een browser de Applet opstart, omdat JoGL dit dan voor je doet via jnlp; dit werkt echter niet via Eclipse.

5.2 OWL bestanden opgeven

Als je wilt opgeven welke OWL bestanden geopend kunnen worden via de Java Applet, moet je in de jnlp file opgeven waar deze gevonden kunnen worden. Dit gebeurt door middel van twee "param"regels.

```
<param name="url1" value="http://<url>/pizza.owl" />
<param name="name1" value="Pizza" />
```

Deze moet in de äpplet-desc"tag, en het kunnen er zo veel zijn als je wilt. Wel moeten de nummers zichzelf opvolgen (beginnend bij 1), en ürlën "naam"samen altijd een paar vormen. Na deze wijziging zul je in de Java Applet de nieuwe bestanden zien bij het herladen van de Applet.

5.3 Updaten van JoGL

Van tijd tot tijd brengt JoGL nieuwe versies uit. Het updaten hiervan kan lastig zijn, zeker als ze API wijzigingen hebben doorgevoerd. In de README staat beschreven hoe dit update proces werkt.

5.4 Publiseren voor het web

Voordat de Applet werkt, moet deze eerst gesigned worden. Let hierbij ook op dat alle lokale libraries gesigned moeten worden met zelfde key. Dat is dus ook

de vecmath.jar etc. Vaak komen zulke libraries zelf ook al met een signing. Het is daarbij belangrijk eerst de signing weg te halen voor deze opnieuw gesigned moet worden. Dit kan door handmatig wijzigingen van de MANIFEST. Hierbij moet wel opgemerkt worden dat dit eigenlijk niet de goede manier is, omdat je een bestaande (meestal geldige) signing verwijderd, en dus de echtheid van de library daarmee niet meer te herleiden is.
Het signen gaat via jarsigner. In de README staat precies beschreven hoe dit process werkt. Het beste kan het gedaan worden met een geldig certificaat, maar een self-signed werkt ook. Dan moet wel de Applet in de lijst met uitzonderingen worden toegevoegd op de computer waar de Applet uitgevoegd gaat worden.

5.5 Openen van de Applet

De Applet kan geopend worden via Eclipse (zie eerder hoofdstuk), via Java webstart (applet-ontology.jnlp), of via een browser (applet.html). Deze laatste start de Java webstart om de voorgaande jnlp uit te voeren.

5.6 Opbouw van de code

Per layer is er een subdirectory. In iedere subdirectory is een API map, waarin de API gedefinieerd is. Deze API is volledig gedocumenteerd met met Javadoc en is zelf beschrijvend.
Andere mappen staan vol met implementaties voor de layer waar hij onder staat. Visualisation is opgedeeld in graph, sphere en containement om onderscheid te maken tussen de verschillende visualisatie methoden.
Als laatste is er nog de Applet.java, waarin de Java Applet zelf staat. Hierin staan de visualisatie methoden hardcoded beschreven (bij het verwijderen en/of toevoegen van een nieuwe zal deze dus aan deze lijst moeten worden toegevoegd), en ook staat de “depth” hier hardcoded in beschreven.

6 Conclusie

De software gemaakt voor dit paper introduceert een modulaire, layer opgebouwde versie van de software geschreven in het proefschrift van Jullie B. Dmitrieva, Aspects of Ontology Visualization and Integration. De nadruk heeft gelegen op maintainability en readability, waar veel aandacht is gegeven aan te zorgen dat anderen dit werk kunnen oppakken en verder kunnen brengen.
Bewust is gekozen om alle code zoveel mogelijk onafhankelijk te laten zijn. Door in ieder layer de keuze vrij te laten voor de gebruiker wat er gebruikt moet worden, is er software neergezet waarvan delen makkelijk hergebruikt kunnen worden voor andere doeleinden, en kan een selectie van de software gebruikt worden voor een minimale versie. De software zelf is verre van af. Er zijn, zoals in het hoofdstuk Discussie langs is gekomen, nog veel dingen te verbeteren en toe te voegen. Dit paper beschrijft vooral het grondwerk voor verder werk.
De software bij het originele werk was zonder meer een uitdagend. Veel code was bedoeld om alleen maar tijdelijk gebruikt te worden. Gelukkig was het rode draad duidelijk, en is het uiteindelijk prima gelukt om de belangrijke onderdelen te vinden en om te zetten.
De huidige software staat vol met commentaar en Javadoc commentaar, functies zijn gekozen om aan te geven ze doen, variabelen zijn voluit geschreven; dit alles om de leesbaarheid zo hoog mogelijk te houden.
De interacties zijn ook verbeterd. In de applicatie zelf is het mogelijk om te schakelen tussen verschillende visualisaties en ontologiën, zonder dat de applicatie opnieuw opgestart moet worden om bijvoorbeeld van categorie te veranderen. Ook is geprobeerd de interacties binnen de visualisaties te verbeteren, waardoor het geheel meer als een compleet product aanvoelt.
Als resultaat staat er een Java Applet die werkt, waarvan 7 visualisaties functioneel werkend zijn, er verschillende bronbestanden gelezen kunnen worden, en er een duidelijk interactie model is gemaakt om 3D manipulaties uit te voeren. Door middel van deze applet kan een ontologie worden ingelezen en op verschillende manieren visueel benaderd worden.

A Javadoc

Bijgevoegd is een zip bestand van de Javadoc in HTML formaat.

B Broncode

Bijgevoegd is een zip bestand met de broncode (inclusief Eclipse project) van het gehele software project.

C Java Applet

Bijgevoegd is een zip bestand met de Java Applet (alleen de web-bestanden).

D Readme

Before you start

For the javaws and webstart version, it automatically detects which OS you are on, and selects the right libraries. For the standalone version, you have to make sure the right libraries are under lib/native. This is because the standalone version can't autodetect this.

For example, if you use Mac OS X, you go to your terminal, go to lib/native and execute:

```
cp macosx-univeral/* .
```

This will extract the right files. Now you can start the Ontology Viewer.

Defining the source files

In the jnlp file you can define what files can be opened from the viewer. Remember that because of security restrictions this has to be a file on the same domain as the jnlp is. For example:

```
<applet-desc
    name="Ontology-Applet"
    main-class="edu.liacs.ontology.Applet"
    width="640"
    height="480">
    <param name="url1" value="http://<url>/pizza.owl" />
    <param name="name1" value="Pizza" />
    <param name="url2" value="http://<url>/amino-acid.owl" />
    <param name="name1" value="Amino Acid" />
</applet-desc>
```

You can keep numbering up url1, ulr2, ... ulrN to define more urls. url and name should always be defined together.

Updating JOGL

JOGL, the 3D engine used, updates frequently. You can find the latest files at:

`http://jogamp.org/deployment/jogamp-current/archive/`

You only need to update if you want to use the standalone version. Any development you do will be done on the standalone version, so update if you want to develop.

```
Download the file jogamp-all-platforms.7z.
Extract the lib/ directory to lib/native/.
Extract the following files from jar/ to lib/:

 lib/gluegen-rt.jar
 lib/jogl.jar
```

Publishing to the web

```
In Eclipse, go to File -> Export.
Select Java -> JAR File
Select "src" (and nothing else)
JAR File: OntologyVisualization/web/edu.liacs.ontologyVis.jar
Export
Sign the JAR file (use keytool to make a signature):
  jarsigner -keystore compstore -signedJar \
    OntologyVisualization\web\edu.liacs.ontologyVis-signed.jar \
    OntologyVisualization\web\edu.liacs.ontologyVis.jar signFiles
Upload the file to the web.
```

Run and debug the viewer

Open the supplied project with Eclipse. Run Applet, and it boots up normally.

View in javaws

Execute: javaws http://url/applet-ontology.jnlp

View in browser

Open applet.html via http://url/applet.html