|  |  |  |
| --- | --- | --- |
| vCity | Juni 10  2014 | |
| Entwicklungsdokumentation des SWP2-Projekts zur Berechnung des Volumens und Verschattung von Stadtmodellen mit Hilfe einer Grafikkarte. | | Softwareprojekt 2 |

Inhalt

[Aufgabenstellung 4](#_Toc390700922)

[Klassendiagramm 5](#_Toc390700923)

[Model 5](#_Toc390700924)

[OpenCL 6](#_Toc390700925)

[Klassenbeschreibungen 7](#_Toc390700926)

[Model 7](#_Toc390700927)

[City 7](#_Toc390700928)

[Building 7](#_Toc390700929)

[BoundarySurface 8](#_Toc390700930)

[Polygon 8](#_Toc390700931)

[Triangle 9](#_Toc390700932)

[ShadowTriangle 9](#_Toc390700933)

[Vertex 9](#_Toc390700934)

[OpenCL 10](#_Toc390700935)

[CalculatorInterface 10](#_Toc390700936)

[CalculatorImpl 10](#_Toc390700937)

[ShadowPrecision 10](#_Toc390700938)

[VolumeCalculatorInterface 11](#_Toc390700939)

[VolumeCalculatorJavaBackend 11](#_Toc390700940)

[VolumeCalculatorOpenClBackend 11](#_Toc390700941)

[ShadowCalculatorInterface 11](#_Toc390700942)

[ShadowCalculatorJavaBackend 11](#_Toc390700943)

[ShadowCalculatorOpenClBackend 11](#_Toc390700944)

[OpenClContext 12](#_Toc390700945)

[SunPositionCalculator 12](#_Toc390700946)

[Werkzeuge 12](#_Toc390700947)

[GUI Bedienungsanleitung 13](#_Toc390700948)

[Tastaturkürzel 13](#_Toc390700949)

[Maus 13](#_Toc390700950)

[GUI 13](#_Toc390700951)

[GUI Aufbau 14](#_Toc390700952)

[Klassendiagramm 14](#_Toc390700953)

[3D Viewer 15](#_Toc390700954)

[Menü 15](#_Toc390700955)

[Beschreibung des Parser-Moduls 16](#_Toc390700956)

[Benutzte Bibliotheken 16](#_Toc390700957)

[CityGML 16](#_Toc390700958)

[StAX 18](#_Toc390700959)

[JAXB 18](#_Toc390700960)

[OpenGL 18](#_Toc390700961)

[Prototyp 18](#_Toc390700962)

[UML Klassendiagramm 19](#_Toc390700963)

[Interfaces 20](#_Toc390700964)

[Import Funktionen 21](#_Toc390700965)

[Allgemeines zum CityGML Import 21](#_Toc390700966)

[Vorgehen: 21](#_Toc390700967)

[CityGML einlesen 21](#_Toc390700968)

[Polygon Triangulation 22](#_Toc390700969)

[Vorgehen 22](#_Toc390700970)

[Vertex Translation 24](#_Toc390700971)

[Verschiebung(Translation) 24](#_Toc390700972)

[Drehung (Rotation) 25](#_Toc390700973)

[Rechnung 25](#_Toc390700974)

[Datenexport 26](#_Toc390700975)

[CSV 26](#_Toc390700976)

[GML 26](#_Toc390700977)

[XML 26](#_Toc390700978)

[Testfälle 28](#_Toc390700979)

[Probleme und Lösungen 28](#_Toc390700980)

[Namenskonflikte mit Datenmodell 28](#_Toc390700981)

[Laufzeit 28](#_Toc390700982)

[Genauigkeit (ungelöst) 29](#_Toc390700983)

[CityGML-Dokumentation 29](#_Toc390700984)

[Quellen 30](#_Toc390700985)

[Fachreferate: 30](#_Toc390700986)

[Bildquellen: 31](#_Toc390700987)

# Aufgabenstellung

Die Aufgabenstellung war die Erstellung eines Programms in Java zur Volumenberechnung eines Stadtmodells auf der Grafikkarte. Dazu soll aus einer CityGML-Datei das Stadtmodell eingelesen werden. Dann sollen die Daten transformiert und in den Ursprung verschoben werden. Zur Anzeige des 3D-Stadmodells soll OpenGL verwendet werden, zur Berechnung auf der Grafikkarte haben wir uns für OpenCL entschieden.

Später kam noch die Verschattungsberechnung dazu. Die Verschattung soll mit dem Ray-Tracing-Verfahren durchgeführt werden.

# Klassendiagramm

## P:\SWP\Klassendiagramm_model.pngModel

## P:\SWP\Klassendiagramm_opencl.pngOpenCL

# Klassenbeschreibungen

## Model

City ist gedacht als übergeordnete Speicherklasse, die allgemeine Daten zur Stadt speichert, sowie die Gebäude beinhaltet. Ein Gebäude besteht aus mehreren **BoundarySurfaces** die wiederum aus Polygonen bestehen. Ein **Polygon** besteht aus mehreren Dreiecken.   
Zur besseren Genauigkeit werden diese Dreiecke bei der Schattenberechnung nochmal in kleinere Schattendreiecke unterteilt. Diese Schattendreiecke werden auch im **Polygon** gespeichert.

### City

City-Klasse bietet Zugriff auf die gesamte Stadt und alle ihre Unterelemente.

* **getInstance**() : City  
  City ist ein Singleton, hierdurch erfolgt der Zugriff auf das City-Objekt.
* **addBuilding**(b : Building) : void  
  fügt ein Gebäude der Stadt hinzu.
* **getBuildings**() : ArrayList<Building>  
  hierdurch erfolgt der Zugriff auf alle Gebäude der Stadt.
* **getCenter**() : Vertex  
  gibt das Zentrum der Stadt zurück.
* **setTotalVolume**(totalVolume : double) : void  
  setzt das Gesamtvolumen der Stadt.
* **getTotalVolume**() : double  
  gibt das Gesamtvolumen der Stadt zurück.
* **setTotalShadowTrianglesCount**(triangleCount : int) : void  
  setzt die Gesamtanzahl von Schattendreiecken der ganzen Stadt.
* **getTotalShadowTrianglesCount**() : int  
  gibt die Gesamtanzahl von Schattendreiecken zurück.

### Building

Repräsentiert ein gesamtes Gebäude.

* **setVolume**(volume : double) : void  
  setzt das Volumen des Gebäudes.
* **getVolume**() : double  
  gibt das Volumen des Gebäudes zurück.
* **addBoundarySurface**(bs : BoundarySurface) : void  
  fügt eine Gebäudeoberfläche dem Gebäude hinzu.
* **addBoundarySurface**(bs : ArrayList<BoundarySurface>) : void  
  fügt dem Gebäude mehrere Gebäudeoberflächen hinzu.
* **getBoundarySurfaces**() : ArrayList<BoundarySurface>  
  gibt alle Gebäudeoberflächen eines Gebäudes zurück.
* **getId**() : String  
  gibt die ID des Gebäudes zurück.
* **translate**(x: float, y: float, z: float) : void  
  verschiebt das Gebäude um die gegebenen Koordinaten.
* **scale**(x: float, y: float, z: float) : void  
  Verändert die Größe des Gebäudes um die gegebenen Parameter.
* **getCenter**() : Vertex  
  gibt den Mittelpunkt des Gebäudes zurück.
* **setCenter**(Vertex) : void  
  setzt den Mittelpunkt des Gebäudes.
* **getStreetName**() : String  
  gibt den Namen des Straße, in dem sich das Gebäude befindet, zurück.
* **setStreetName**(name : String) : void  
  setzt den Namen der Straße zu der das Gebäude gehört.

### BoundarySurface

Beschreibt die verschiedenen Oberflächen eines Gebäudes.

* **BoundarySurface**(id : String)  
  erzeugt eine neue Oberfläche mit gegebener ID.
* **getId**() : String  
  gibt die ID einer Oberflöche zurück.
* **setType**(type : SurfaceType) : void  
  setzt den Type einer Oberfläche (WALL, ROOF, GROUND, OTHER)
* **getType**() : SurfaceType  
  gibt den Typ der Oberfläche zurück.
* **getPolygons**() : ArrayList<Polygons>  
  gibt alle Polygone zurück, die die Oberfläche darstellen.
* **addPolygon**(p : Polygon) : void  
  fügt der Oberfläche ein Polygon hinzu.
* **addPolygons**(polygons : ArrayList<Polygon>) : void  
  fügt der Oberfläche mehrere Polygone hinzu.

### Polygon

* **setPercentageShadow**(percentageShadow : double[]) : void  
  setzt den prozentualen Anteil des Dreiecks der sich im Schatten befindet.
* **getPercentageShadow**() : double[ ]  
  gibt den prozentualen Anteil zurück zu dem sich das Dreieck im Schatten befindet.
* **Polygon**(String)  
  Konstruktor mit ID als Übergabeparameter.
* **getId**() : String  
  gibt die ID des Polygons zurück.
* **getArea**() : double  
  gibt den Flächeninhalt des Polygons zurück.
* **setArea**(a : double) : void  
  setzt den Flächeninhalt des Polygons.
* **getTriangles**() : ArrayList<Triangle>  
  gibt alle zum Polygon zugehörigen Dreiecke zurück.
* **addTriangle**(t : Triangle) : void  
  fügt dem Polygon ein Dreieck hinzu.
* **addTriangles**(ts : ArrayList<Triangle>) : void  
  fügt dem Polygon mehrere Dreiecke hinzu.
* **addShadowTriangle**(st : ShadowTriangle) : void  
  fügt dem Polygon ein Schattendreieck hinzu.
* **getShadowTriangles**() : ArrayList<ShadowTriangle>  
  gibt alle zum Polygon gehörenden Schattendreiecke zurück.

### Triangle

* **getNormalVector**() : Vertex  
  gibt den Normalenvektor des Dreiecks zurück.
* **setNormalVector**(normalVektor : Vertex) : void  
  setzt den Normalenvektor des Dreiecks.
* **Triangle**(vertices : Vertex[])   
  Konstruktor mit drei Punkten als Array übergeben zum Konstruieren des Dreiecks.
* **Triangle**(v0 : Vertex, v1 : Vertex, v2 : Vertex)   
  Konstruktor mit drei Punkten die direkt übergeben werden.
* **getVertices**() : Vertex[ ]  
  gibt alle Punkte des Dreiecks zurück.

### ShadowTriangle

* **ShadowTriangle**(v : Vertex[], b : Building)  
  Konstruktor mit drei Punkten als Array die das Schattendreieck definieren und zusätzlich wird das Gebäude übergeben zu dem das Schattendreieck gehört.
* **ShadowTriangle**(v0 : Vertex, v1 : Vertex, v2 : Vertex, b : Building)  
  Konstruktor mit drei Punkten die direkt als Punkte übergeben werden.
* **getShadowSet**() : BitSet  
  gibt das zu diesem Schattendreieck gehörige SchattenBitSet zurück.
* **getBuilding**() : Building  
  gibt das Gebäude zurück zu dem das Schattendreieck gehört.
* **setShadowSet**(s : BitSet) : void  
  setzt das SchattenBitSet für dieses Schattendreieck.
* **getCenter**() : Vertex  
  gibt den Mittelpunkt des Schattendreiecks zurück.

### Vertex

* **Vertex**(coordinates : float[])  
  Konstruktor in dem die Koordinaten für den Ortsvektor als Array übergeben werden.
* **Vertex**(x : float, y : float, z : float)  
  Konstruktor in dem die Koordinaten für den Ortsvektor direkt als float übergeben werden.
* **getCoordinates**() : float[]  
  gibt die Koordinaten des Ortsvektors zurück.
* **getX**() : float  
  gibt die x – Koordinate des Ortsvektors zurück.
* **getY**() : float  
  gibt die y – Koordinate des Ortsvektors zurück.
* **getZ**() : float  
  gibt die z – Koordinate des Ortsvektors zurück.
* **translate**(x : float, y : float, z : float) : void  
  verschiebt den Ortsvektor um die gegebenen Parameter.
* **scale**(x: float, y : float, z : float) : void  
  skaliert den Ortsvektor um die gegebenen Parameter.

## OpenCL

Das **CalculatorInterface** und die Implementierung **CalculatorImpl** dienen als Schnittstelle um die Volumen- sowie die Schattenberechnung aufzurufen.  
In der Aufzählung **Shadowprecision** wird eine Unterteilung der Präzision der Schattenberechnung durch Festlegung der maximal erlaubten Größe der Dreiecksflächen vorgenommen.  
Falls die Berechnung mit Hilfe von OpenCL fehlschlagen sollte, so wird die Berechnung mit Java durchgeführt.   
Die Klasse **SunPositionCalculator** dient dazu die Position der Sonne einem Himmelsabschnitt zuzuordnen.

### CalculatorInterface

* **calculateVolume**() : void  
  berechnet das Volumen aller Gebäude der Stadt.
* **calculateShadow**(precision : ShadowPrecision, splitAzimuth : int, splitHeight : int) : void  
  berechnet die Verschattung der Stadt mit der gegebenen Präzision und der gegebenen Aufteilung des Himmelsmodells.
* **calculateArea**() : void  
  berechnet die Fläche aller Polygone der Stadt.

### CalculatorImpl

Implementiert das **CalculatorInterface**

### ShadowPrecision

* **VERY\_LOW**  
  Schattendreiecke mit maximaler Fläche von 5m²
* **LOW**  
  Schattendreiecke mit maximaler Fläche von 2,5m²
* **MID**  
  Schattendreiecke mit maximaler Fläche von 1,25m²
* **HIGH**  
  Schattendreiecke mit maximaler Fläche von 0,75m²
* **ULTRA**  
  Schattendreiecke mit maximaler Fläche von 0,375m²
* **HYPER**  
  Schattendreiecke mit maximaler Fläche von 0,10m²
* **AWESOME**  
  Schattendreiecke mit maximaler Fläche von 0,01m²
* **getArea**() : float  
  gibt die maximal erlaubte Fläche der Präzisionsstufe zurück.
* **getName**() : String  
  gibt den Namen der Präzisionsstufe zurück.

### VolumeCalculatorInterface

* **calculateVolume**() : void  
  berechnet das Volumen der Gebäude die zur Stadt gehören.

### VolumeCalculatorJavaBackend

Implementiert das Interface **VolumeCalculatorInterface**.

### VolumeCalculatorOpenClBackend

Implementiert das Interface **VolumeCalculatorInterface**.

### ShadowCalculatorInterface

* **calculateShadow**(precision : ShadowPrecision, splitAzimuth : int, splitHeight : int) : void  
  berechnet die Verschattung der Stadt mit der gegebenen Präzision und der gegebenen Aufteilung des Himmelsmodells.
* **vertexDiff**(v1 : Vertex, v0 : Vertex) : Vertex

### ShadowCalculatorJavaBackend

Erweitert die abstrakte Klasse **ShadowCalculatorInterface**.

### ShadowCalculatorOpenClBackend

Erweitert die abstrakte Klasse **ShadowCalculatorInterface**.

### OpenClContext

* **getInstance**() : OpenClContext  
  Gibt das Objekt von OpenClContext zurück.
* **reinit**() : void  
  erstellt einen neuen OpenCL-Kontext.
* **createKernelFromFile**(filename : String) : cl\_kernel  
  erstellt einen OpenCL – Kernel aus der angebenen Datei.
* **finalizeKernel**(cl\_kernel : kernel) : void  
  gibt die Ressourcen des Kernels wieder frei.
* **getDevice**() : cl\_device\_id  
  Gibt die ID der zu verwendenden GPU zurück.
* **getClContext**() : cl\_context  
  gibt den OpenCL - Kontext zurück.
* **getClCommandQueue**() : cl\_command\_queue  
  gibt die OpenCL – Commandqueue zurück.
* **profile**(kernelEvent : cl\_event) : void  
  schreibt Timing-Daten auf die Console.

### SunPositionCalculator

* **SunPositionCalculator**(d : Date, parser : ParserInterface)  
  Konstruktor mit übergebenem Datum und ParserInterface. Berechnet die Azimuth- und Höhenwinkel der Sonne zum gegebenen Datum und Ort.
* **getAzimutAngle**() : double  
  gibt den Azimuthwinkel zurück.
* **getAltitude**() : double  
  gibt den Höhenwinkel zurück.
* **getX**() : double  
  gibt die x – Koordinate des Vektors in Richtung der Sonne zurück.
* **getY**() : double  
  gibt die y – Koordinate des Vektors in Richtung der Sonne zurück.
* **getZ**() : double  
  gibt die z – Koordinate des Vektors in Richtung der Sonne zurück.
* **getSunPosition**(splitAzimuth : int, splitHeight : int) : int  
  gibt den Index des Himmelsabschnitts in dem sich die Sonne befindet zurück.

# Werkzeuge

Eclipse

Git

Notepad++

UMLet/ Enterprise Architect

# GUI Bedienungsanleitung

### Tastaturkürzel

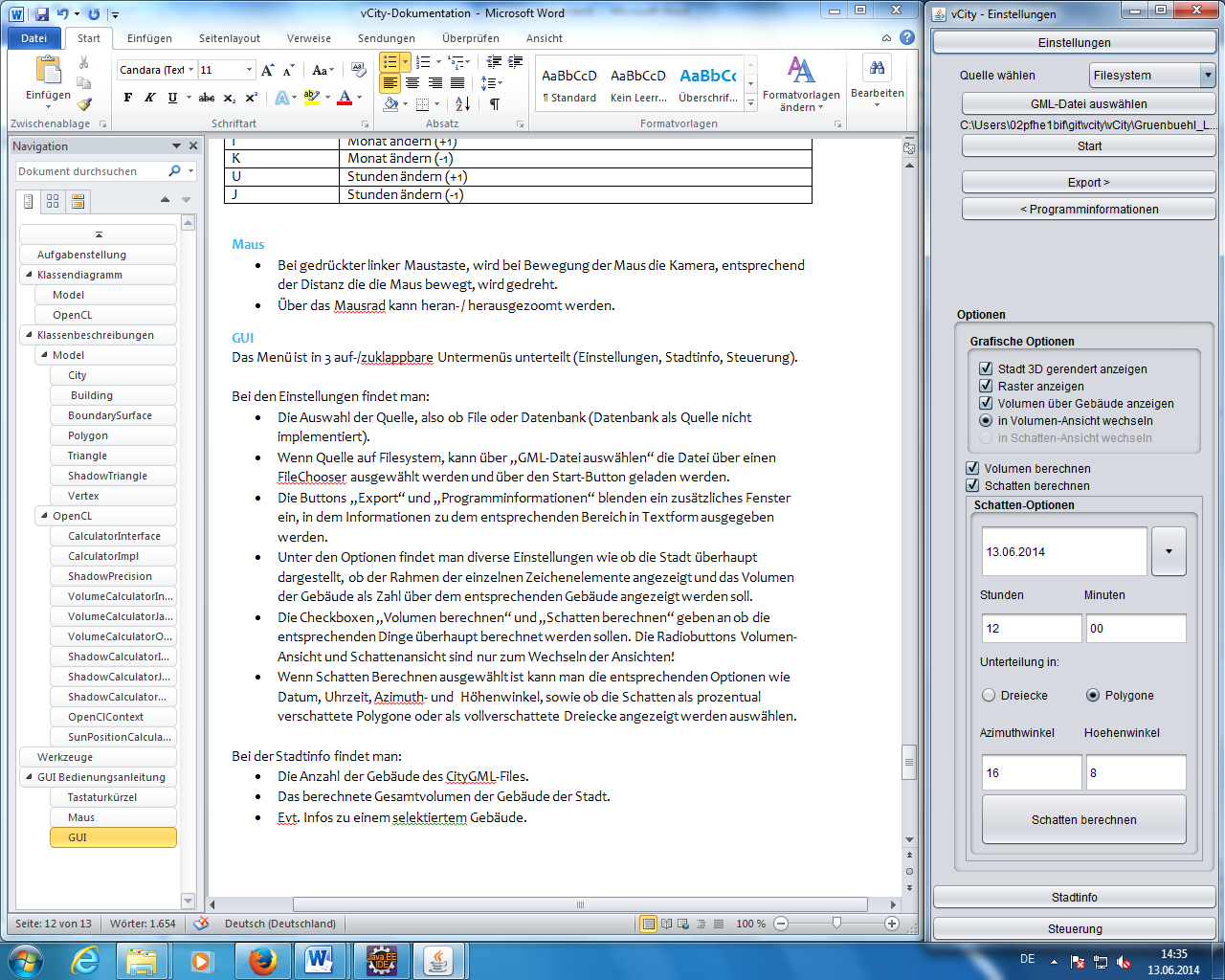
|  |  |
| --- | --- |
| W / Arrow Up | Kamera nach vorne bewegen |
| A / Arrow Left | Kamera nach links bewegen |
| S / Arrow Down | Kamera nach hinten bewegen |
| D / Arrow Right | Kamera nach rechts bewegen |
| O | SkayPatch nach links ändern |
| L | SkayPatch nach rechts ändern |
| E | Sonnenposition ein-/ausblenden |
| I | Monat ändern (+1) |
| K | Monat ändern (-1) |
| U | Stunden ändern (+1) |
| J | Stunden ändern (-1) |

### Maus

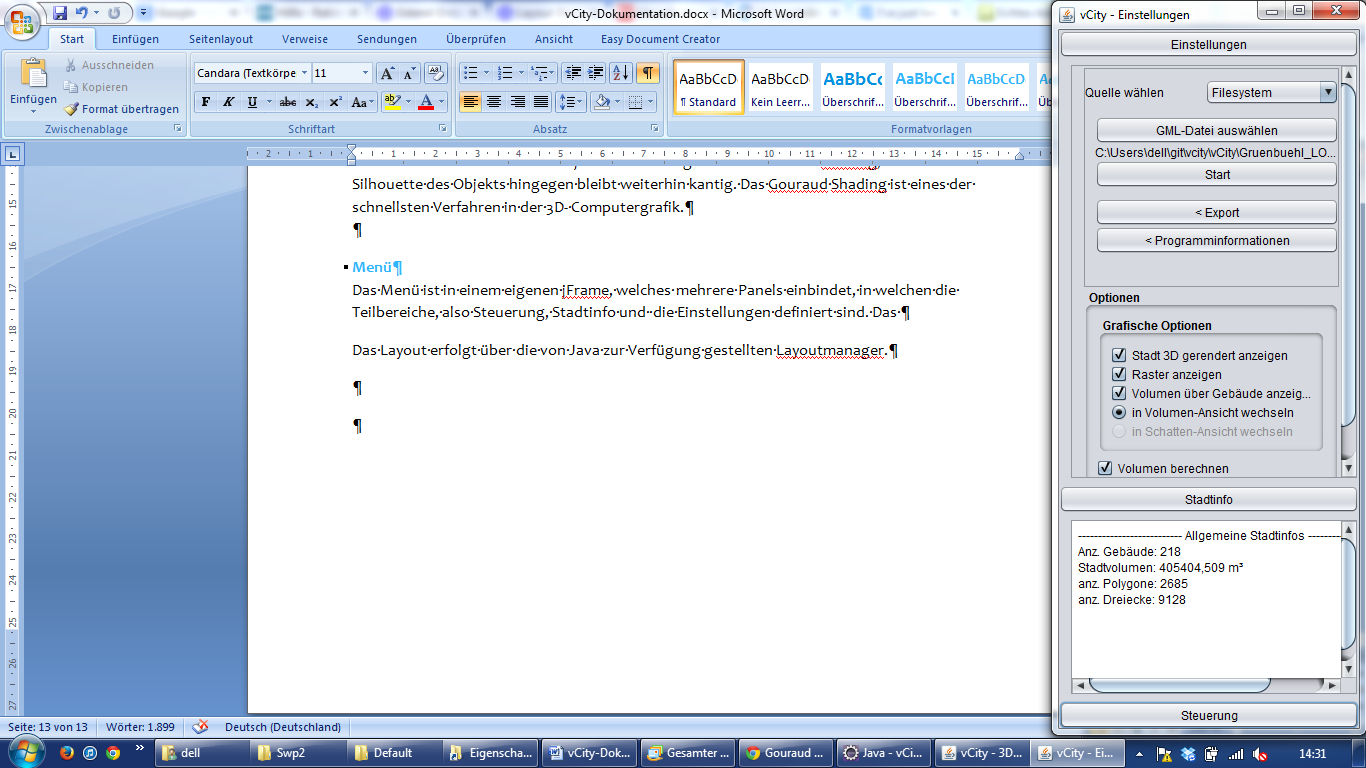
* Bei gedrückter linker Maustaste, wird bei Bewegung der Maus die Kamera, entsprechend der Distanz die die Maus bewegt, wird gedreht.
* Über das Mausrad kann heran- / herausgezoomt werden.

### GUI

Das Menü ist in 3 auf-/zuklappbare Untermenüs unterteilt (Einstellungen, Stadtinfo, Steuerung).

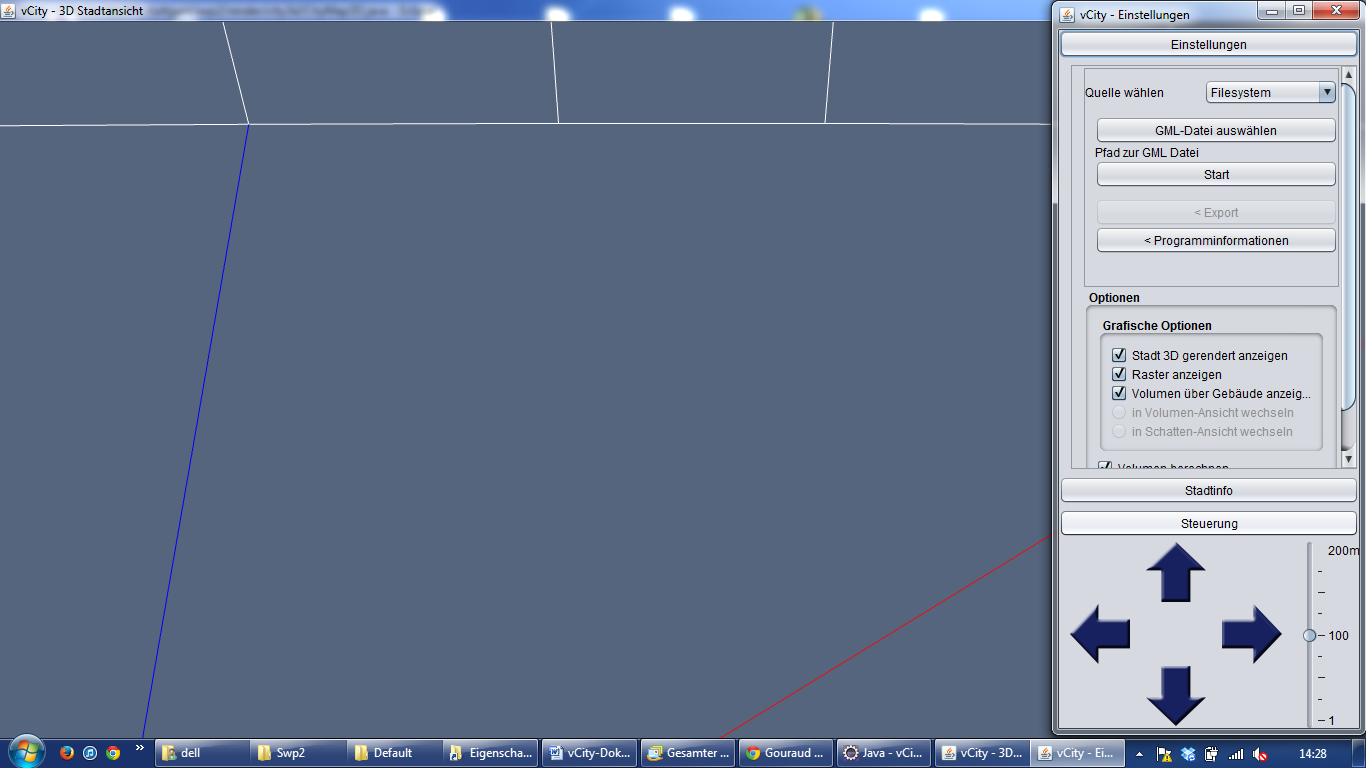
Bei den Einstellungen findet man:

* Die Auswahl der Quelle, also ob File oder Datenbank (Datenbank als Quelle nicht implementiert).
* Wenn Quelle auf Filesystem, kann über „GML-Datei auswählen“ die Datei über einen FileChooser ausgewählt werden und über den Start-Button geladen werden.
* Die Buttons „Export“ und „Programminformationen“ blenden ein zusätzliches Fenster ein, in dem Informationen zu dem entsprechenden Bereich in Textform ausgegeben werden.
* Unter den Optionen findet man diverse Einstellungen wie ob die Stadt überhaupt dargestellt, ob der Rahmen der einzelnen Zeichenelemente angezeigt und das Volumen der Gebäude als Zahl über dem entsprechenden Gebäude angezeigt werden soll.
* Die Checkboxen „Volumen berechnen“ und „Schatten berechnen“ geben an ob die entsprechenden Dinge überhaupt berechnet werden sollen. Die Radiobuttons Volumen-Ansicht und Schattenansicht sind nur zum Wechseln der Ansichten!
* Wenn Schatten Berechnen ausgewählt ist kann man die entsprechenden Optionen wie Datum, Uhrzeit, Azimuth- und Höhenwinkel, sowie ob die Schatten als prozentual verschattete Polygone oder als vollverschattete Dreiecke angezeigt werden auswählen.



Bei der Stadtinfo findet man:

* Die Anzahl der Gebäude des CityGML-Files.
* Das berechnete Gesamtvolumen der Gebäude der Stadt.
* Eventuell Infos zu einem selektierten Gebäude.

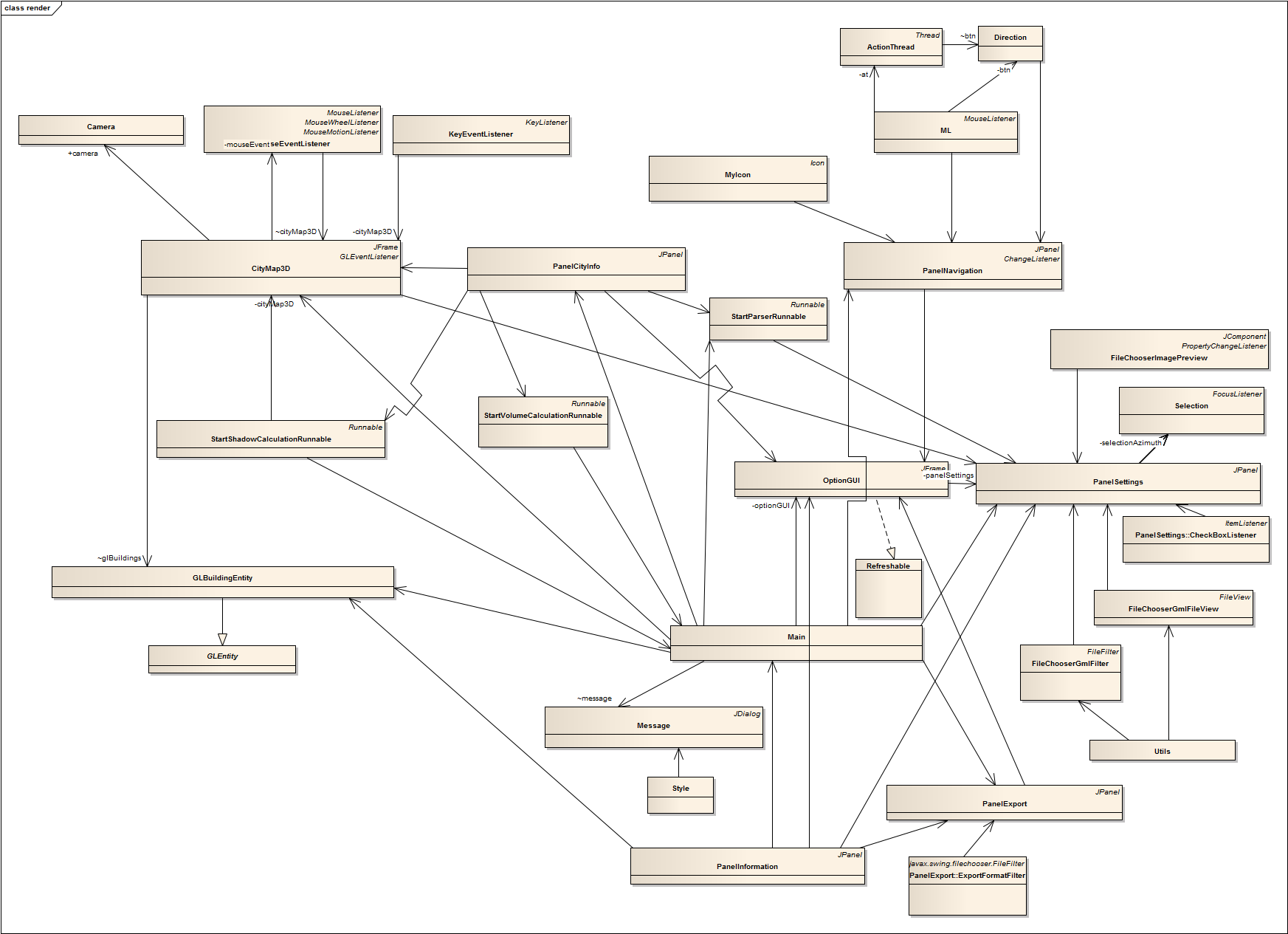
Steuerung:

Die Steuerung besteht aus einem Slider zum Zoomen, sowie einem Steuerkreuz mit welchem die Kamera bewegt werden kann. Die Steuerkreuze sind eigentlich Bilder die beim Presse und Release andere Bilder einfügen um eine Animation zu erstellen.

# GUI Aufbau

Grundsätzlich setzt sich die GUI aus 2 getrennten Fenstern, nämlich dem 3D Viewer und dem eigentlichen Menü, zusammen. Die Idee dahinter war die 2 Fenster auf 2 Bildschirmen anzeigen zu lassen um die Usability zu verbessern.

### Klassendiagramm



### 3D Viewer

Der 3D Viewer wurde in Java und OpenGL über die Schnittstelle/Bibliothek JOGL realisiert, welche dem Java-Code entsprechenden Code in C ausführt.

Beim Gouraud Shading werden die Normalen der Oberfläch interpoliert, dadurch erscheinen facettierte Oberflächen eines Objekts nicht kantig wie z.B. beim Flat Shading, sondern weich. Die Silhouette des Objekts hingegen bleibt weiterhin kantig. Das Gouraud Shading ist eines der schnellsten Verfahren in der 3D- Computergrafik.

### Menü

Das Menü ist in einem eigenen jFrame, welches mehrere Panels einbindet, in welchen die Teilbereiche, also Steuerung, Stadtinfo und die Einstellungen definiert sind. Das

Das Layout erfolgt über die von Java zur Verfügung gestellten Layoutmanager. Z.b. ist das OptionGUI ein GridLayout und das PanelSettings ein GridBackLayout.

Die Volumen und Schatten Berechnung wird in einem Thread realisiert wobei hier ein ExecutorService in der Main für die Verwaltung der Threads zuständig ist.

# Beschreibung des Parser-Moduls

Im nachfolgenden Kapitel der Dokumentation werden wir auf die Funktionen und die Benutzung des Parsers eingehen. Das Modul deckt die folgenden Bereiche der vCity-Software ab:

* Import und parsen der CityGML-Datei
* Koordinatentransformation zu einem definierten Referenzwert
* Polygon-Triangulation
* Drei verschiedene Exportvarianten

# Benutzte Bibliotheken

## CityGML

CityGML wurde 2002 in NRW in offener Arbeitsgemeinschaft unter Mitarbeit von über 70 Institutionen entwickelt. Es ist ein XML-basiertes Datenmodell für die Speicherung von 3D-Stadtmodellen. Es werden relevante Objekte des städtischen Raumes klassifiziert und ihre Eigenschaften beschrieben. Auch ohne Visualisierung lassen sich bereits viele Informationen aus den XML-Daten extrahieren, da verschiedene thematische Attribute wie zum Beispiel die Adresse oder das Baujahr vieler Gebäude mit angegeben sind. Seit 2004 wird CityGML als internationale Standardisierung vorangetrieben und auch zunehmend in der Forschung verwendet. Dies ist an der großen Bandbreite an Einrichtungen zu erkennen, die eigene Parser oder Visualisierungen für CityGML anbieten. Das KIT (Karlsruhe), die Universität Bonn, die TU Berlin und viele weitere vermessen Gebäude vieler vorwiegend größerer Städte (zum Beispiel Berlin, Hamburg, Frankfurt, Stuttgart, ...). Grundsäztlich finden einige Basisklassen in CityGML verwendung: Gebäude, Gelände, Vegetation, Wasserkörper, Straßenmöbel und einige weitere. Für diese Arbeit sind die Gebäude wichtig, andere geographische Merkmale finden bisher keine Verwendung. Jede der genannten Basisklassen besitzt wiederrum ein “Level Of Detail” (LOD), also eine Angabe wie detailliert die XML-Daten angegeben wurden. Die Stufen sind am Beispiel von Gebäuden wie folgt zu verstehen:



Abbildung 1 citygml2: CityGML Encoding Standard, Kolbe et al. 2012

Skalenbereiche, Levels of Detail, LOD:  
  
Die LOD-Stufen nach [GRÖGERet al.] am Beispiel von Gebäuden:

• LOD 0 bietet 3D-Landmarken, die gegebenenfalls mit Luftbildern versehen werden kann. Man nennt es “Regionalmodell”.

• LOD 1 beinhaltet Häuser quaderförmiger Struktur. Materialeigenschaften können angegeben werden.

• LOD 2 besitzen einzelne Oberflächen eine eigene Geometrie, Dächer und Wände werden vom restlichen Gebäude unterschieden. Einzelne Flächen können hier mit Eigenschaften belegt werden.

• LOD 3 ermöglicht detaillierte Hausfassaden mit “Löchern” in der Textur (zum Beispiel Fenster, Türen, ...).

• LOD 4 beinhaltet auch die Innenarchitektur der Gebäude samt Einrichtungsgegenständen

Für die Volumenberechnung in diesem Projekt wurde die Detailstufe “LOD 2” gewählt, da sie die Ergebnisse bei hinnehmbarem Aufwand schon genau genug abbilden kann und zudem viele Testdaten in dieser Stufe vorhanden sind.

## StAX

Die “Streaming API for XML” (StAX) wurde 2006 vom “Java Community Process” ist dafür gedacht, XML-Dateien mit Java zu verarbeiten. Es stellt einen Mittelweg zwischen den zuvor beliebten DOM- und SAX-Parsern dar. Statt einer Baumstruktur, die komplett in den Arbeitsspeicher geladen wird, wie es bei DOM-Parsern der Fall ist, verwendet es einen Mechanismus der die Daten genau dann abruft, wenn sie gebraucht werden. Statt wie beim SAX-Parser mit mehreren Durchläufen zu arbeiten, ist es mit StAX möglich, den Baum dynamisch zu verändern.

## JAXB

Neben StAX wird in diesem Projekt noch die API “Java Architecture for XML Binding” (JAXB) als Alternative zu SAX und DOM verwendet, da sich herausstellte, dass die Möglichkeiten zum Bearbeiten der XML-Dokumente aus der Java-Anwendung wesentlich komfortabler möglich ist.

## OpenGL

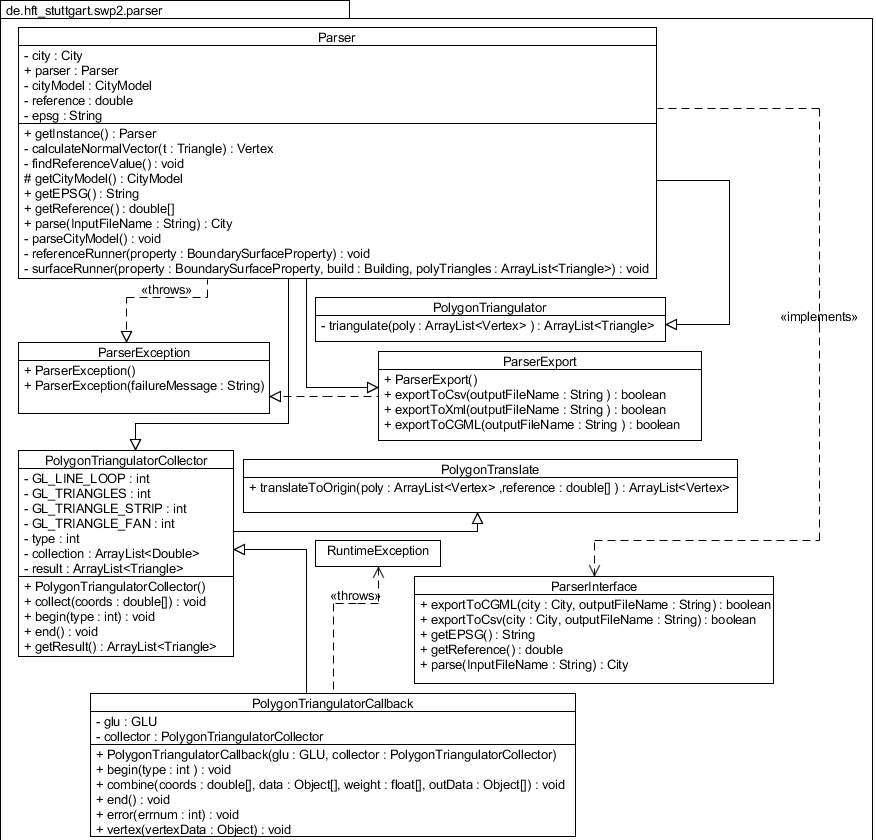
OpenGL wurde 1992 von der Khronos Group, einem Industriekonsortium mit namhaften Mitgliedern wie zum Beispiel AMD, Google und Oracle entwickelt und stetig weiterentwickelt. In diesem Projekt wird OpenGL zum Triangulieren genutzt.

# Prototyp

Zum Einarbeiten in CityGML und um dessen Funktionsweise zu verstehen, wurde ein Prototyp geschaffen. Er entstand in der ersten Iteration und wurde stetig weiterentwickelt, sobald Fortschritte erfolgt waren. In einem späteren Kapitel werden die Probleme Erläutert, die dabei enststanden.

# Klassendiagramm

Dieses Klassendiagramm wurde mit [UMLet](http://www.umlet.com/umlet_12_2/umlet_12.2.zip)  Version 12.2 erstellt



Das Package besteht aus 8 Klassen und 2 Interface-Klassen. Die Hauptklasse ist die Parser-Klasse. Sie enthält 3 public Methoden und 5 private Methoden.

Die Methode “parse“ liest die CityGML- Datei, bzw. die Gebäude („Buildings“), die darin enthalten sind, transformiert deren Koordinaten zum Ursprung, wandelt dann die gewonnenen Polygone zu Vertices um und erstellt anschließend ein Building Objekt. Dieses Objekt enthält Building-ID, Boundary-Surface, Polygon und Vertices.

Es werden zusätzlich zu den Klassen im Parser-Klassendiagramm noch die Objekte aus dem

Package-Model benötigt:

* City
* Building
* Vertex
* BoundarySurface
* Polygon
* Triangle

Die Methode „parseCityModel“ sucht nach Buildings und deren Flächen beziehungsweise Koordinaten und anhand gefundener Polygone, speichert die x, y, z-Koordinaten aus jeder Oberflächen-Ecke, die sie findet in einem Array vom Typ Double.

Die Methode „surfaceRunner“ verwendet „PolygonTriangulator“ um an die Polygone der Gebäude ran zukommen und verwendet danach „PolygonTranslate“ um die Gebäude in die im Koordinatensystem auf die richtige Position zu verschieben. Die Koordinaten von triangulierten Polygonen speichert sie in einen ArrayList<Triangle>.

Mit der statischen Methode „translateToOrigin“ werden die Koordinatenachsen transformiert. Das heißt sie transformiert die komplexen Koordinaten so, dass die näher am Ursprung des Koordinatensystems liegen. Das dient dazu um später die Koordinatendaten einfacher zu verarbeiten zu können.

Die public Methode „triangulate“ aus der Klasse PolygonTriangulator nutzt die JOGL-Bibliothek um aus den Polygonen aus dem Package Model Dreiecke (Triangles) zu erzeugen und speichert sie in einem ArryList<Triangle>. Die detaillierte Beschreibung des Algorithmus werden wir im nächsten Kapitel erläutern.

# Interfaces

Das Parser-Modul enthält ein ParserInterface mit den Methoden, „parse“, „getEPSG“, „getReference“, wie man aus dem UML-Diagramm oben entnehmen kann.

Die Methode „getEPSG“ gibt den EPSG als String aus der GML-Datei und die Methode „getReference“ gibt die Referenzwerte aus, die für die Koordinaten-Transformationen später benötigt werden.

Das Parser-Modul enthält ein zweites Interface für den Export, mit den Methoden, „exportToCGML“, „exportToCSV“ und „exportToXML“ .Die Methoden dienen zur Exportierung verschiedener Daten. Durch die Parameterangabe outputFileName kann man bestimmen, wie die exportierte Datei später heißen soll. Die Methode gibt true, falls Exportvorgang erfolgreich verläuft, sonst false und/oder wirft eine ParserException.Die drei Export-Varianten werden wir im weitern Verlauf noch näher erläutern.

# Import Funktionen

## Allgemeines zum CityGML Import

In der Parser.java Klasse werden die Daten nach den gegebenen Anforderungen geparst und modifiziert. Dazu zählen das Einlesen einer GML-Datei, die Koordinatentransformation und das Aufbauen und füllen des Datenmodells.

### Vorgehen:

Durch das Singleton-Pattern wird sichergestellt, dass nur eine Instanz des Parsers erzeugt wird. Nachdem durch das Parserinterface eine Instanz des Parsers erzeugt wurde, ist es nur notwendig die Methode parse(String InputFileName) aufzurufen. Die restlichen Schritte werden intern durch den Parser übernommen.

Funktionen des Parsers:

* CityGML-Datei einlesen
* Koordinaten transformieren
* Triangulieren
* Normalen Vektor eines Dreiecks ausrechnen

### CityGML einlesen

Analog zu der GML Datei muss der Parser durch die verschiedenen Tags durchgehen. Hierbei geht der Parser jeweils durch die cityObjectMember denn diese enthalten die Gebäude. Danach wird aus der geparsten Gebäude ID ein Gebäude aus dem internen Datenmodell erzeugt. Als Nächstes werden dem Gebäude der Stadtname, in dem es sich befindet und der Straßenname hinzugefügt. Um die Koordinaten der Polygone herauszufinden, wird hierbei die Methode surfaceRunner aufgerufen.

Das Durchlaufen der GML Datei geschieht hierbei durch die beiden Methoden:

referenceRunner**(**BoundarySurfaceProperty property**) und**

surfaceRunner**(**BoundarySurfaceProperty property**,** Building build**,** ArrayList**<**Triangle**>** polyTriangles**)**

Der referenceRunner durchsucht die GML nach dem Referenzwert, dieser Wert wird benötigt um die Koordinaten zum Ursprung zu transformieren. Der Referenzwert besteht aus dem Vertex mit der kleinsten x-Koordinate und dem dazugehörigen y-und z-Koordinaten.

Der SurfaceRunner durchsucht die Oberflächen nach Koordinaten. Dazu geht der Parser über alle Oberflächen und speichert ihren Typ. Danach gelangt der Parser über den LinearRing zu den gewünschten Gebäudekoordinaten. Diese Koordinaten werden in einer Liste gespeichert. Um die Koordinaten anschaulich zu machen, werden diese noch zum Koordinatenursprung transformiert und trianguliert. Nach der Triangulation wird hierbei gleich der Normalvektor der Dreiecksfläche ausgerechnet und dem Modell hinzugefügt.

Als letzten Schritt werden diese Dreiecke dem Datenmodell Polygon, die Polygone der BoundarySurface und der BoundarySurface dem Gebäude hinzugefügt. Zu guter Letzt werden die Gebäude der City hinzugefügt und zurückgegeben.

## Polygon Triangulation

Der für die Renderer benötigte Bibliothek kann nur konvexe Polygone zeichnen. Ein Polygon ist konvex, wenn der Innenwinkel an jedem Eckpunkt höchstens 180° beträgt. Falls auch nur ein Innenwinkel größer als 180° ist, ist das Polygon konkav. Um konkave Polygone zeichnen zu können, müssen diese Polygone erst in Dreiecke unterteilt werden denn Dreiecke sind immer konvex. Diese Unterteilung der Polygone nennt man Triangulation (engl.Tesselation).

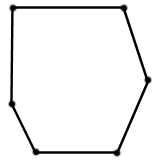
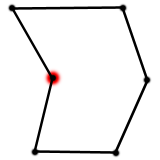
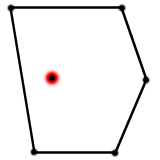
  

Abbildung 2 konvexes Polygon Abbildung 3 konkaves Polygon Abbildung 4 konkaves Polygon mit Fehler

Um diese Triangulation durchzuführen bietet das Framework JOGL verschieden Routinen an. Diese Routinen erhalten als Input das Polygon und als Ergebnis werden die Dreiecke abhängig von Typ zurückgegeben.

### Vorgehen

In der Klasse PolygonTriangulator wird zuerst ein Tesselator Objekt erzeugt dies geschieht indem GLU.*gluNewTess*() aufgerufen wird. Danach werden durch verschiedene Callback Routinen Funktionen für das Objekt festgelegt. Diese Funktionen werden durch *gluTessCallback* definiert*.*

*gluTessCallback* ist folgendermaßen aufgebaut*:*

GLU.*gluTessCallback*(tessobj, type, funktion);

Hier werden dem Objekt tessobj die Funktion funktion hinugefügt. Der Typ des Callbacks wird durch type angeben.

In unserem Programm werden die Typen *GLU\_TESS\_VERTEX*, *GLU\_TESS\_BEGIN*, *GLU\_TESS\_END*,

*GLU\_TESS\_ERROR* und *GLU\_TESS\_COMBINE* verwendet.

*GLU\_TESS\_VERTEX* speichert die Eckpunkte eines Polygons

*GLU\_TESS\_BEGIN* startet ein neues Polygon

*GLU\_TESS\_END* beendet das aktuelle Polygo*n*

*GLU\_TESS\_ERROR* gibt eine Fehlermeldung mit dem Fehler aus

*GLU\_TESS\_COMBINE* erzeugt ein neues Vertex falls sich 2 Seiten schneiden.

Diese Callback Routinen werden im Laufe des Triangulationsvorgangs von der GLU-Bibliothek aufgerufen. Die Routinen werden in der Klasse PolygonTriangulatorCallback definiert und bekommt seine Daten wiederum durch die Klasse PolygonTriangulatorCollector.

In der Klasse PolygonTriangulatorCollector werden die Vektoren gesammelt und anhand ihres Typs ausgewertet. Diese Typüberprüfung findet am Ende eines Polygons statt.

Diese Typen sind:

* GL\_TRIANGLE
* GL\_Triangle\_STRIP
* GL\_TRIANGLE\_FAN

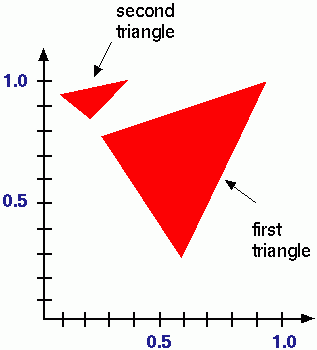


Abbildung 5 GL\_TRIANGE

Jeweils 3 Vektoren bilden hierbei ein unabhängiges Dreieck.

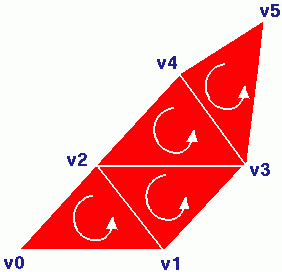


Abbildung 6 GL TRIANGLE\_STRIP

Die Dreiecke sind durch eine Seite (2 Eckpunkte) miteinander verbunden. Hier bei ist zu beachten, dass alle Dreiecke die gleiche Drehrichtung haben.

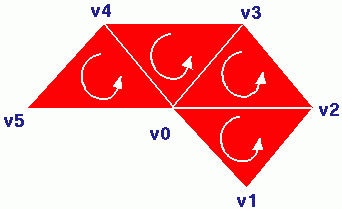


Abbildung 8 GL\_TRIANGLE\_FAN

Der Triangle Fan ist ähnlich zu dem Triangle Strip außer, dass der Anfangspunkt bei alles Dreiecken gleich ist.

## Vertex Translation

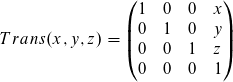
Damit die Daten beim Renderer passend ankommen müssen sie etwas verändert werden, dazu müssen die Koordinaten einmal zum Koordinatenursprung verschoben und gedreht werden. Diese beiden Veränderungen geschehen mit der Methode:

translateToOrigin**(** ArrayList**<**VertexDouble**>** poly**,** double**[]** reference**)**

Diese Methode bekommt als Parameter eine Liste vom Typ VertexDouble und ein Array mit den Referenzwerten. Die Liste enthält die geparste Posliste aus der GML Datei. Das Referenzarray wird vom Parser ermittelt.

### Verschiebung(Translation)

Die x,y und z Koordinaten einer Vertex werden durch eine Matrix zum Koordinatenursprung verschoben. Die Matrix lautet:



Dabei entsprechen die Werte x,y und z den aus dem Parser ermitteltem Referenzwert.

### Drehung (Rotation)

Die Koordinatenachsen müssen an der X-Achse gegen den Uhrzeigersinn um 90° gedreht werden, weil die Achsen aus der GML Datei nicht mit denen des Renderers übereinstimmen.

x’ ´

z ’

y ’

x

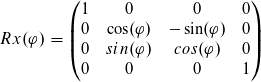
y

z

Koordinatenachsen aus der Koordinatenachsen des Renderers

GML Datei

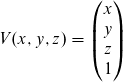
Die Matrix für die Rotation lautet:



φ entspricht hierbei -90°

### Rechnung

Die zu verändernden Koordinaten werden aus der geparsten Liste entnommen und in diese Form eingefügt:



Im nächsten Schritt werden die Verschiebe- und Rotationsmatrizen miteinander multipliziert. Schließlich werden Alle Vektoren V mit der zuvor berechneten Transformationsmatrix multipliziert. Als Ergebnis kommen die Transformierten Vektroren heraus.

# Datenexport

## CSV

Diese Option schreibt die Gebäude-IDs und deren berechneten Volumen in eine einfache CSV (**C**omma **S**eparated **V**alues) Datei.

Die Exportmethode verwendet dabei einen Java **FileWriter**, weil wir es unnötig fanden, hierfür ein extra Framework heran zu ziehen.

Building,Volume

DEBW\_LOD2\_1007722,112.82454681396484

## GML

Die Software kann die eingelesene GML (**G**eography **M**arkup **L**anguage)-Datei wieder exportieren. Dabei wird dem geparsten Datenmodell das errechnete Volumen der Gebäude als double-Wert hinzugefügt und danach als GML-Datei ausgeschrieben.

Der Export verwendet das **CityGML4J** Framework, welches auch für den Import verwendet wird. Das Framework steht unter der GNU LGPL-Lizenz.

<gen:doubleAttribute name="Volume">

<gen:value>112.82454681396484</gen:value>

</gen:doubleAttribute>

## XML

Die dritte Exportmöglichkeit ist der Export als XML (*e****X****tensible* ***M****arkup* ***L****anguage*)-Datei. Dieser Export ist für eine Weiterverwendung mit dem externen Programm “**INSEL**” gedacht.

Das XML-Datenformat ist dabei nach den Anforderungen eines Kundenreviews entworfen worden. Es ist zu beachten, dass Koordinaten und ähnliches nicht in diesem Format exportiert werden.

Die DTD (**D**ocument **T**ype **D**efinition) des XML-Dokuments ist wie folgt definiert:

<!ELEMENT City (SkyModel, Building+)>

<!ELEMENT SkyModel (SplitAzimuth, SplitHeight)>

<!ELEMENT SplitAzimuth (#PCDATA)>

<!ELEMENT SplitHeight (#PCDATA)>

<!ELEMENT Building (BoundarySurface)+>

<!ATTLIST Building

id CDATA #REQUIRED

>

<!ELEMENT Volume (#PCDATA)>

<!ATTLIST Volume

uom CDATA #REQUIRED

>

<!ELEMENT BoundarySurface (Polygon)+>

<!ATTLIST BoundarySurface

id CDATA #REQUIRED

type (GROUND|ROOF|WALL|OTHER) #REQUIRED

>

<!ELEMENT Polygon (Area, Shadow)>

<!ATTLIST Polygon

id CDATA #REQUIRED

>

<!ELEMENT Area (#PCDATA)>

<!ATTLIST Area

uom CDATA #REQUIRED

>

<!ELEMENT Shadow (#PCDATA)>

Zum erstellen des XML-Datenobjekts verwenden wir das **StAX** Framework, das unter der Apache v2.0 Lizenz vorliegt. Wir haben uns für dieses Framework entschieden, weil es Performant und leicht zu verwenden ist.

Wir erstellen XML-Objekte mit der Methode Element e = doc.createElement("Element"); . Attributs werte werden mit der Methode Attr a = doc.createAttribute("Attribute"); erstellt. Der Vorteil hierbei ist, dass wir zum Füllen des Datenobjekts einfach über jedes Gebäude gehen und die benötigten Daten auslesen können.

for (each Building)

add ID;

add Volume;

for (each Surface)

...

# Testfälle

Um die Funktionalität des Imports und des Exports zu testen, haben wir eine JUnit-Testklasse geschrieben. Die Klasse enthält diverse Testfälle für den Import, das Parsen, die Triangulation, die Transformation und die Exportmöglichkeiten. Die größte Methode hier ist testReadAndParseAndValidateEinHaus(), welche einen kompletten Import der Gruenbuehl GML-Datei enthält, dabei wird jeder Schritt geprüft und validiert. Die drei Exportvarianten werden ebenfalls ausgeführt, jedoch müssen die geschriebenen Dateien bislang noch manuell überprüft werden.

# **Probleme und Lösungen**

## **Namenskonflikte mit Datenmodell**

Nachdem das Datenmodell entworfen war, entstanden in der Parser-Gruppe Namenskonflikte, die wir nur durch direktes “Ansprechen” der gewünschten Klasse beheben konnten (zum Beispiel org.citygml4j.model.citygml.building.Building Building, damit nicht das Building unseres eigenen Datenmodells verwendet wird).

## Laufzeit

Das Parsen der XML-Daten benötigt nach wie vor relativ lange Zeit. So hat die Beispieldatei eines kleinen Vorortes von Ludwigsburg weniger als 100MB Inhalt. Wählt man das Stadtmodell größer, werden definitiv nicht hinnehmbare Zeiten entstehen. Das Stadtmodell von Berlin ist zum Beispiel 21,1GB groß und lässt erahnen, dass die Laufzeit für unser Projekt in Ordnung ist, für größere Projekte an der Performanz aber noch gearbeitet werden muss.

Genauigkeit (ungelöst)   
Durch das “casten” von double-Werten und dem damit verbundenen Abschneiden von Nachkommastellen, verlieren die Daten an Genauigkeit. Da die Weiterverarbeitung jedoch mit double-Werten nicht funktioniert, muss dieser Schritt beim Parsen unternommen werden. Denkbar wäre eine andere Datenstruktur, die die Daten in zwei Abschnitten bearbeitet und das Ergebnis wieder zusammenführt.

## CityGML-Dokumentation

Zum Zeitpunkt der Erstellung des Prototypens gab es nicht viele Tutorials oder Beispiele im Internet zu finden, mit denen man hätte arbeiten können. So musste an manchen Stellen “gecastet” werden, die wir ohne die Hilfe von Prof. Dr. Coors nur mit wesentlich mehr Zeitaufwand eventuell gefunden hätten.

Um auf den Namen der Stadt zuzugreifen in dem sich ein Gebäude befindet, ist zum Beispiel folgende Verkettung von Methodenaufrufen notwendig, um den XML-Baum bis an die gewünschte Stelle hinab zu gehen:

String theCity = building.getAddress().get(0).getAddress().getXalAddress().getAddressDetails().getCountry().getLocality().getLocalityName().get(0).getContent();  
  
Dabei sind theoretisch auf jeder Ebene NullPointerExceptions möglich, die abgefangen werden sollten.

# Quellen

## Fachreferate:

T.H. Kolbe, 2008, CityGML - Ein Standard für virtuelle 3D-Stadtmodelle, Skript  
T.H. Kolbe, 2005, CityGML, OGC TC Meeting Bonn (Open Geospatial Consortium Technical committee meeting)

## Veröffentlichungen:

C.A.L.Sánchez, 2013, Estimation of Electric Energy Demand using 3D City Models, Master Thesis  
Löwner et al., 2013, CityGML 2.0 – Ein internationaler Standard für 3D-Stadtmodelle, Zeitschrift für Geodäsie, Geoinformation und Landmanagement (138 Jg., Heft 2)

G.Gröger, Lutz Plümer, 2012, CityGML - Interoperable semantic 3D city models, ISPRS Journal of Photogrammetry and Remote Sensing  
Felix Kunde, 2012, CityGML in PostGIS, Master Thesis, Universität Potsdam

Open GL Programming,

<http://profs.sci.univr.it/~colombar/html_openGL_tutorial/en/02rendering_022.html>, 15.06.2014

DGL Wiki, <http://wiki.delphigl.com/index.php/Konvex>, 15.06.2014

Kiet Le Polygon Tessellation in JOGL,

<http://www.java-tips.org/other-api-tips/jogl/polygon-tessellation-in-jogl.html>, 15.06.2014

GLU Polygon Tessellation, <http://www.felixgers.de/teaching/jogl/gluTessellation.html>, 15.06.2014

Estevao, Martin, Polygon Tessellation In OpenGL,

<http://www.flipcode.com/archives/Polygon_Tessellation_In_OpenGL.shtml>, 15.06.2014

## Bildquellen:

Abbildung 1 <http://lh5.ggpht.com/-TQoisMDRkRM/T_QLRsg2UeI/AAAAAAAAAPA/ZmbB3KCsH9g/image_thumb%2525255B13%2525255D.png%3Fimgmax%3D800> 15.06.2014

Abbildung 2: <http://wiki.delphigl.com/images/2/28/Konvex_normal.png>,15.06.2014

Abbildung 3: <http://wiki.delphigl.com/images/4/4c/Konvex_konvex.png>,15.06.2014

Abbildung 4: <http://wiki.delphigl.com/images/2/29/Konvex_konvex2.png>,15.06.2014

Abbildung 5: <http://profs.sci.univr.it/~colombar/html_openGL_tutorial/images/triangles.gif>,15.06.2014

Abbildung 6:

<http://profs.sci.univr.it/~colombar/html_openGL_tutorial/images/triangle_strips.gif>, 15.06.2014

Abbildung 7: <http://profs.sci.univr.it/~colombar/html_openGL_tutorial/images/triangle_fans.gif>,15.06.2014