

Rheinische Friedrich-Wilhelms-Universität Bonn

Institut für Kartographie und Geoinformation

**Digitale Geländemodelle und Texturen
in einer objekt-relationalen
3D-Geodatenbank**

Diplomarbeit

Jörg Schmittwilken

29.März 2004

Betreuer:

Prof. Dr. Lutz Plümer

Dr. Thomas H. Kolbe

Aufgabenstellung

Problemstellung

Am Institut für Kartographie und Geoinformation wird ein Kernel eines 3D-Geoinformationssystems auf Grundlage des objekt-relationalen Datenbankmanagementsystems (DBMS) Oracle Spatial entwickelt. Es wurde in Kooperation mit dem Institut für Photogrammetrie und der Stadt Wuppertal ein geometrisch-topologisches Gebäudemodell erstellt. Die Realisierung des Modells basiert auf dem Nutzen offener Normen und Standards. Eine Systemkomponente zur Verwaltung eines 3D-Stadtmodells ist bereits implementiert. Doch erst die Verschneidung der Gebäude mit einem digitalen Geländemodell als Bezugsgrundlage macht eine grafische Ausgabe für den Benutzer sinnvoll. Hierzu bietet das Landesvermessungsamt Nordrhein-Westfalen für einen großen Teil der Landesfläche Daten digitaler Geländemodelle mit einer Rasterweite von drei bis fünf Metern an. Ihre Verwendung setzt wegen des großen Datenvolumens insbesondere eine effiziente Verwaltung und einen schnellen Datenzugriff voraus. Hierzu sind DBMS in besonderer Weise geeignet. Das o.g. Modell sieht bislang kein Konzept zur Oberflächentexturierung vor, obwohl gerade diese Art der Visualisierung eine sehr gute Möglichkeit bietet, digitale 3D-Daten photorealistisch darzustellen.

Aufgabenstellung

Die Diplomarbeit soll das vorhandene Objektmodell um das digitale Geländemodell und Texturen erweitern und die 3D-Geodatenbank entsprechend ergänzen. Es geht dabei konkret um Ergänzungen des UML-Modells, die Herleitung eines entsprechenden Datenbankschemas und die Implementierung in Oracle Spatial. Ein weiterer Schwerpunkt soll auf der Visualisierung texturierter digitaler Geländemodelle liegen. Es soll eine Schnittstelle für den Export aus der Datenbank in die Virtual Reality Modeling Language geschaffen werden. Die Randbedingungen und Anforderungen von VRML an texturierte Darstellungen sollen bereits bei dem Datenbankentwurf berücksichtigt werden. Für den Import des DGM aus ArcInfo soll ebenfalls eine Schnittstelle bereitgestellt werden.

Ausgabedatum: 11. 12. 2003

Bearbeitungszeit: vier Monate

Erklärung

Hiermit erkläre ich, dass ich diese Diplomarbeit selbständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Bonn, 28. März 2004

(Jörg Schmittwilken)

Danksagung

Ich danke Carola und Markus S., Markus R., Michael und Philipp für das Korrekturlesen dieser Arbeit.

Für die fachliche Unterstützung und die konstruktiven Diskussionen bedanke ich mich bei Gerd, Thomas, Viktor, Peter und Markus R.

Für die Ermöglichung dieser Arbeit und die Förderung und Motivation meines wissenschaftlichen Arbeitens danke ich Lutz.

Bei meinen Eltern bedanke ich mich für die Förderung und Unterstützung meines Studiums. Sie haben meine Pläne und Ziele stets befürwortet und mich in meinen Vorhaben unterstützt.

Ganz besonderer Dank gilt meiner Frau Britta und meiner Tochter Pauline. Sie haben während der Forschung und insbesondere während der Dokumentation dieser Arbeit weitestgehend auf gemeinsame Zeit mit mir verzichten müssen.

Inhaltsverzeichnis

1	Einleitung	11
2	3D-Geodatenbanken – Der Status quo	13
2.1	Modelle räumlicher Daten	14
2.1.1	Digitale Geländemodelle	14
2.1.2	3D-Stadtmodelle	16
2.2	3D-Geoinformationssysteme	16
2.3	3D-Geodatenbanken – eine Alternative?	17
2.3.1	Abgrenzung dieser Arbeit	18
3	Abstrakte Modellierung von DGM	21
3.1	Modell der Geländeoberfläche	21
3.1.1	Modellierung der Geometrie	21
3.1.2	Modellierung der Beschaffenheit	22
3.1.3	Zusammenführen der Modelle	24
3.2	Integration in das Gesamtmodell	24
4	Texturierte Darstellung von Geometrien	27
4.1	Grundlagen der Texturierung	27
4.2	VRML97	29
4.2.1	Grundlagen	30
4.2.2	Rasterbild-Texturen	31
4.3	Orthophotos	35
4.3.1	Georeferenzierung	35
4.3.2	Optimierung des Speicherbedarfs	36
5	Abbildung des Modells auf eine 3D-Geodatenbank	41

5.1	Datenbankmanagementsysteme	41
5.1.1	Relationale Datenbanksysteme	42
5.1.2	Objektorientierte Datenbanksysteme	44
5.1.3	Objekt-relationale Datenbanksysteme	44
5.2	Datenbankkommunikation	46
5.2.1	SQL	46
5.2.2	PL/SQL	48
5.2.3	JDBC	49
5.3	Spezielle Objekttypen	51
5.3.1	ORDImage	51
5.3.2	Oracle Spatial	53
5.4	Datenbankschema	59
5.4.1	Erweiterung des vorhandenen DB-Schemas	59
5.4.2	Mögliche Änderungen des DB-Schemas	66
5.5	Schnittstellen	67
5.5.1	Import in die Datenbank	69
5.5.2	Export nach VRML	70
6	Zusammenfassung und Ausblick	75
A	VRML-Beispiele	79
A.1	Texturierung ohne Angabe von Parametern	79
A.2	Texturkoordinaten	80
A.3	Transformation der Rasterbildtextur	81
B	Datenbankschema	83
B.1	SQL-Befehle zum Generieren des DB-Schemas	83
B.2	SQL-Befehle zum Löschen des DB-Schemas	95
B.3	Grafische Übersicht des DB-Schemas	96
B.4	Klassendiagramm der Schnittstellen	97
C	PL/SQL	99
D	Inhaltsverzeichnis der CD-ROM	103

Kapitel 1

Einleitung

Digitale Geländemodelle (DGM) beschreiben das Relief der Erdoberfläche in einer für die elektronische Datenverarbeitung verwendbaren Form. Es werden diskrete Punkte des Geländes durch Lage- und Höhenkoordinaten beschrieben. Beispielsweise werden vom Landesvermessungsamt Nordrhein-Westfalen mittlerweile flächendeckend DGM in Form eines Rasters mit Kantenlängen von bis zu einem Meter angeboten, die diese diskretisierte Beschreibung des Reliefs ermöglichen.

DGM können genutzt werden, um durch computergestützte Verfahren Berechnungen durchzuführen. Das können beispielsweise Erdmassen für Abgrabungen, die Richtung des Oberflächenabflusses eines Niederschlagsereignisses oder Quer- und Längsprofile für den Straßenbau sein. Für diese Anwendungen ist eine grafische Darstellung des DGMs nicht zwingend notwendig. Anders verhält es sich beispielweise bei der Nutzung von DGM für virtuelle Stadtrundgänge oder der 3D-Darstellung eines geplanten Bauvorhabens unter Berücksichtigung der vorhandenen Bebauung. Je nach manuellem Aufwand der Herstellung wird dem Benutzer ein detailliertes, photorealistisches 3D-Stadtmodell präsentiert, in dem er möglicherweise interaktiv navigieren kann. Doch was wäre dieser Spaziergang im virtuellen Raum ohne die Darstellung der Geländeoberfläche? Bei fehlender Visualisierung des DGMs würden die Häuser „in der Luft hängen“, und der Beobachter hätte das Gefühl zu „fliegen“. Erst in Verbindung mit der grafischen Darstellung des Reliefs erzeugt die Visualisierung eines 3D-Stadtmodells beim Benutzer einen realistischen Eindruck. DGM können durch Dreiecksnetze beschrieben werden, die unabhängig von Luftbildaufnahmen des gleichen Ausschnitts der Erdoberfläche erzeugt werden. Bei der Kombination dieser beiden Oberflächenrepräsentationen zur grafischen Darstellung können am Rande des Dreiecksnetzes unerwünschte Effekte auftreten. Stimmen Luftbildkanten und Dreieckskanten nicht überein, so wird z.B. ein Dreieck von bis zu vier Luftbildern beschrieben, was die Verwendung von standardisierten Texturierungsverfahren unmöglich macht. Diese Arbeit zeigt sowohl eine theoretische Lösung dieses Problems als auch die praktische Umsetzung, die auf der Nutzung einer 3D-Geodatenbank beruht.

Neben den eingangs angeführten Nutzungsmöglichkeiten von DGM sind z.B. für Raumplaner oder Katastrophenmanager weitere Fragestellungen von Interesse. Diese könnten beispielsweise lauten: „Welche Gebäude mit mehr als 10m Höhe stehen an ei-

nem Hang mit mehr als 5% Gefälle?“ oder „Welche Gebäude sind bei einem Stand des Bonner Rhein-Pegles von 8,30m nicht mehr trockenen Fußes zu erreichen?“. Es handelt sich um raumbezogene Analysen, die 3D-Daten und Nachbarschaftsbeziehungen als Berechnungsgrundlage nutzen und von Geoinformationssystemen (GIS) durchgeführt werden. GIS sind für die Verwaltung, Visualisierung und besonders für die Analyse raumbezogener Daten konzipiert. Da bis vor wenigen Jahren sowohl Software als auch Daten auf zwei Dimensionen beschränkt waren, ist die Forderung nach 3D-GIS-Funktionalität für 3D-Stadt- oder Geländemodelle relativ neu. Ein weiterer Aspekt dieser Entwicklung ist, dass das Datenvolumen aufgrund der Dreidimensionalität um ein Vielfaches steigt und die Analyse dreidimensionaler Daten im Vergleich zum Zweidimensionalen an Komplexität zunimmt. Aus diesen Gründen bieten kommerzielle GIS-Produkte nur unzureichende Unterstützung und Funktionalität für 3D-Daten, so dass sie für o.g. Problemstellungen nicht verwendet werden können.

Datenbankmanagementsysteme (DBMS) wurden ursprünglich zur effizienten Verwaltung großer Datenmengen (z.B. Adressen, Auftragsverwaltung) entwickelt. Die Hersteller integrieren in neuere Versionen ihrer DBMS Datentypen und Funktionen für raumbezogene Daten, so dass diese auch als *Geodatenbanken* bezeichnet werden, die auf den Umfang und die Leistung ihre GIS-Funktionalität hin zu untersuchen sind. Diese Arbeit bietet einen Überblick über einige in der Fachliteratur erläuterte Arbeiten mit Geodatenbanken.

Am Institut für Kartographie und Geoinformation der Universität Bonn wird zurzeit der Kern eines 3D-GIS auf Grundlage des DBMS Oracle entwickelt. Zwar existiert bereits die Implementierung eines Modells zur Integration von 3D-Stadtmodellen unter Berücksichtigung von geometrisch-topologischen Beziehung. Dieses berücksichtigt die Beschreibung des Reliefs durch DGM jedoch nicht. Gerade die Oberflächenrepräsentation ist aber für die Visualisierung von enormer Bedeutung. Die vorliegende Arbeit stellt die notwendige Erweiterung des Modells um Dreiecksnetze (*Triangular Irregular Network* – *TIN*) zur Beschreibung eines DGMs vor. Ein entsprechendes Datenbankschema zur Repräsentation von DGM wird ebenfalls erläutert. Dabei wird insbesondere die Beschreibung der Oberfläche durch farbliche Differenzierung oder durch Rasterbilder berücksichtigt.

Die Arbeit ist wie folgt gegliedert: Nach einer kurzen Erläuterung der Modellierung räumlicher Daten wird zur Abgrenzung und Einordnung dieser Arbeit Literatur vorgestellt, die sich mit 3D-GIS und 3D-Geodatenbanken beschäftigt. Das in dieser Arbeit entwickelte Datenmodell wird im Anschluss detailliert beschrieben, um dann auf Aspekte und Konzepte der Visualisierung dreidimensionaler Geometrien einzugehen. Dazu wird der Standard der *Virtual Reality Modeling Language* (*VRML*) vorgestellt. Außerdem wird die problembezogene Beschreibung der grafischen Darstellung raumbezogener Daten mit Orthophotos thematisiert. Es folgt eine Einführung in DBMS und die ausführliche Betrachtung des Oracle-Datentyps für räumliche Daten. Abschließend werden das Datenbankschema und die entsprechenden in *Java* implementierten Schnittstellen zum Datenimport und -export präsentiert.

Der Quelltext der *Java*-Klassen und deren *javadoc*-Beschreibung befinden sich auf der zu dieser Arbeit gehörenden CD-ROM.

Kapitel 2

3D-Geodatenbanken – Der Status quo

Viele Kommunen, private Dienstleister und große Unternehmen erstellen eigene 3D-Stadtmodelle, die u.a. von Architekten, Raumplanern, den Mobilfunkbetreibern und dem Katastrophenmanagement genutzt werden können. Nachdem zu Beginn der neunziger Jahre damit begonnen wurde, einen Großteil der Papierkarten in eine digitale Form zu überführen, wächst seit einigen Jahren der Bedarf an dreidimensionalen Daten (s. [Cie03]). Telefondienstleister planen Funknetzzellen für den Mobilfunk, Städte präsentieren virtuelle Stadtrundgänge durch Gebiete mit geplanter und bestehender Bebauung, und der Katastrophenschutz ermittelt zu evakuierende Häuser im Fall von Hochwasserereignissen. Für eine Vielzahl von Anwendungen scheint die Nutzung von 3D-Daten denkbar.

Mit Verfahren wie dem luftgestütztem Laserscanning¹ und leistungstarker Algorithmen zur Ableitung von Boden- oder Dachpunkten aus den Ergebnissen dieser Messungen stehen hoch effiziente Methoden zur Gewinnung der Datengrundlage von digitalen Geländemodellen (DGM) und 3D-Stadtmodellen zur Verfügung. Die Ableitung eines 3D-Stadtmodells ist noch nicht vollständig automatisiert und erfolgt manuell (computergestützt).

Darüberhinaus stellen unter anderem Zipf und Schilling [ZS02] und Cieslik [Cie03] Verfahren vor, mit denen 3D-Modelle aus 2D-Daten abgeleitet werden können. Letzteres wird seit einigen Jahren erfolgreich bei der Erstellung des 3D-Stadtmodells der Freien und Hansestadt Hamburg angewendet.

Die teilweise automatisierbare und somit kostengünstige Herstellung von DGM und 3D-Stadtmodellen ist ein weiterer Grund der stetig steigenden Nachfrage nach 3D-Geoinformation.

¹Von einem Flugzeug aus wird ein Fächer von Laserstrahlen ausgesandt. Aus der Laufzeit vom Laser bis zum Reflektionspunkt auf der Erdoberfläche (Straße, Haus, Baum o.ä.) und zurück zu dem im Flugzeug angebrachten Empfänger kann die Entfernung der Bodenpunkte vom Flugzeug errechnet werden. Bei bekannter Position des Flugzeugs ist somit auch die Position der Bodenpunkte bekannt und es kann ein 3D-Oberflächenmodell abgeleitet werden.

2.1 Modelle räumlicher Daten

Als Grundlage für die Diskussion über 3D-GIS-Systeme wird im Folgenden eine kurze Übersicht über verschiedene Modelle zur Repräsentation von 3D Daten gegeben. Grundsätzlich ist zu beachten, dass zum Ziel der redundanzfreien Speicherung raumbezogener Daten stets eine geometrisch-topologische Beschreibung bevorzugt werden sollte (vgl. [OSQZ02, Mol92]).

2.1.1 Digitale Geländemodelle

Digitale Geländemodelle dienen der Beschreibung der Erdoberfläche ohne Topographie, also der rein geomorphologischen Gestalt des Reliefs. Sie werden durch *Felder* beschrieben, die aus regelmäßig (*Raster*) oder unregelmäßig (*TIN*) verteilten, diskreten Punkten bestehen.

Jeder Punkt des Netzes hat genau einen Höhenwert als Funktionswert seiner Lage ($z = f(x, y)$). Aus diesem Grund werden Felder auch als 2,5D-Modelle bezeichnet. In Feldern kann einem Parameterpaar (x, y) nur ein Funktionswert (z) zugewiesen werden, was die Darstellung von senkrechten (vertikalen) Flächen unmöglich macht.

TIN

Ein *Triangulated Irregular Network (TIN)* ist ein spezielles Dreiecksnetz, bei dem aus der unregelmäßig (*irregular*) verteilten Punktwolke Dreiecke gebildet werden (*Triangulation*). Die Abbildung 2.1 zeigt eine solche Oberflächenbeschreibung.

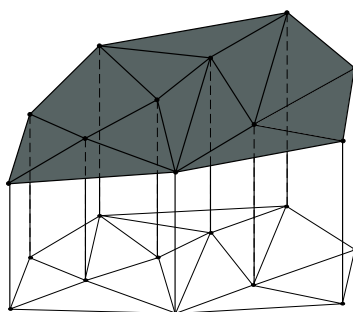


Abbildung 2.1: Triangulated Irregular Network (TIN).

Grundsätzlich ist die Anordnung der Dreiecke beliebig. Eine besonders gute Beschreibung der Oberflächenstruktur ergibt sich jedoch, wenn das *Delaunay*-Kriterium erfüllt ist. Es besagt, dass innerhalb des Umkreises² eines jeden Dreiecks kein Punkt eines anderen Dreiecks liegt. Eine geometrische Betrachtung dieser Forderung zeigt, dass sich die Reliefbeschreibung einer *Delaunay-Triangulation* dadurch auszeichnet, dass die entstehenden Dreiecke annähernd gleichschenkelig sind. Es entstehen keine sehr langen, spitzen Dreiecke, die das Gelände nur unzureichend beschreiben würden. Eine

²Der Umkreis eines Dreiecks ist der Kreis, der durch die drei Eckpunkte verläuft.

Delaunay-Triangulation kann durch verschiedenen Algorithmen implementiert werden, deren Laufzeitverhalten stark von der Anzahl der Punkte abhängig ist.

Eine weitere Spezialisierung der Triangulation stellen *Bruchkanten* dar. Bruchkanten üben Zwangsbedingungen auf die Triangulation aus, da sie zwingend als Dreieckskanten trianguliert werden müssen. Gradlinien von Gebirgen oder Gewässerkanten sind beispielsweise als Bruchkanten zu definiert, da sie im TIN nicht von Dreieckskanten geschnitten werden dürfen. Dieses Verfahren wird als *Constrained Delaunay Triangulation* bezeichnet (vgl. [OBSC99]).

Lenk [Len03] und Lenk und Heipke [LH02] stellen ein Verfahren zur Verschneidung eines digitalen Situationsmodells³ (DSM) mit einem TIN vor. Es werden die im DSM dargestellten Straßen- und Gewässerbegrenzungen in das TIN als Bruchkanten eingefügt, so dass die Fläche der Objekte als horizontale Dreiecke im TIN repräsentiert werden.

Raster

Ein Raster ist eine geordnete Menge von 3D-Punkten, deren Lagekoordinaten die Eckpunkte eines rechteckigen (meistens quadratischen) Gitters bilden. Durch die im Raum orientierten Vierecke kann das Geländere relief beschrieben werden. Es handelt sich, wie beim TIN, um eine 2,5D Darstellung. Abbildung 2.2 zeigt ein Beispiel der Oberflächenbeschreibung durch ein Raster.

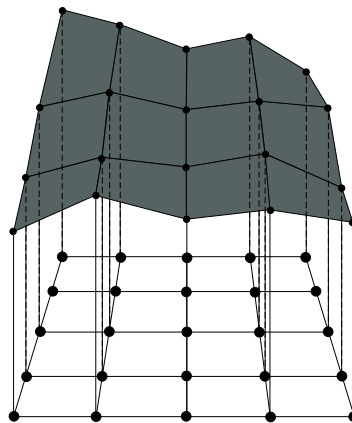


Abbildung 2.2: Ein Raster dient der Oberflächenbeschreibung eines Gebiets der Erdoberfläche.

Zur Visualisierung mit Rasterbildtexturen (s. Kapitel 4.2.2) eignet sich das Raster nicht, da die im Raum orientierten Vierecke nicht zwingend planar sein müssen. Planarität der Fläche ist im verwendeten *VRML*-Standard jedoch notwendige Voraussetzung für die Texturierung mit Rasterbildern⁴.

³Ein digitales Situationsmodell beschreibt nur Lageinformationen, z.B. das Amtliche Topographisch-Kartographische Informationssystem (ATKIS) (<http://www.atkis.de>).

⁴In *VRML* sind die Oberflächen der drei vordefinierten Volumengeometrien *Kugel*, *Kegel* und *Zylinder* die einzigen gekrümmten Flächen, die mit Rasterbildern texturiert werden können.

Die Qualität der Reliefbeschreibung hängt beim Raster stark von der vorgegebenen Rasterweite, der so genannten *Auflösung* ab. Ist diese zu groß gewählt, so können feinere Strukturen nicht erfasst und dargestellt werden. Ist die Auflösung zu klein, so vergrößert sich das Datenvolumen, ohne eine genauere Beschreibung der Oberfläche zu erreichen, da viele Vierecke koplanar sind.

Dagegen kann beim TIN in stark bewegten Gebieten oder in Bereichen, die aus anderen Gründen exakter beschrieben werden sollen, eine höhere Punktdichte gewählt werden. Dieser Übergang von einem großen zu einem kleinen Punktabstand hat keinen Einfluss auf die Triangulation. Zipf und Schilling [ZS02] stellen ein Verfahren für „dynamisch generierte Multiresolutions-Geländemodelle“ vor, das ein TIN mit konzentrischen Bereichen unterschiedlicher Punktdichten generiert.

2.1.2 3D-Stadtmodelle

Zur Modellierung einer Stadt müssen die Bauwerke, Verkehrswege, Straßenmöbel⁵, Vegetation usw. in geeigneter Form repräsentiert werden. Zur Vereinfachung wird die Darstellung in einem ersten Schritt jedoch auf oberirdische Gebäude beschränkt. Zur Steigerung des Realitätsgrads eines 3D-Stadtmodells können mehrere Detaillierungsstufen (*level of detail (LOD)*) unterschieden werden (vgl. [KG03]), die sich wie folgt gliedern:

- LOD1 – Klötzchenmodell aus Quadern,
- LOD2 – Simple Dachstrukturen und einfache Texturierung,
- LOD3 – Detaillierte Ausgestaltung (Gauben, Dachüberstände, Erker usw.), stark differenzierte Texturierung.

Einen ausführlichen Überblick über Datenstrukturen zur Modellierung allgemeiner 3D-Geometrien geben Foley et al. [FDFH95]. Einige dieser Modelle werden häufig zur Modellierung von 3D-Stadtmodellen verwendet. Zumindest namentlich seien hier die *konstruktive Festkörpergeometrie (constructive solid geometry – CSG)* und die *Randflächendarstellung (boundary representation – B-Rep)* erwähnt.

2.2 3D-Geoinformationssysteme

Mit zunehmendem Bedarf an 3D-Daten steigen auch die Anforderungen an Geoinformationssysteme. Diese müssen nicht nur in der Lage sein, große Mengen an Daten zu verwalten, sondern auch vollständige GIS-Funktionalität bieten, die sich nicht nur auf Volumen-, Umfang- oder Flächenberechnung beschränkt. Es müssen ebenfalls Informationen wie topologische Beziehungen oder Nachbarschaften abzufragen und die

⁵Als *Straßenmöbel* werden Gegenstände im Bereich von Straßen und Fußgängerwegen bezeichnet. Das sind z.B. Verkehrsschilder, Telefonzellen, Fahrradständer.

Datenkonsistenz zu prüfen sein.

Mit diesen Ansprüchen geht die Forderung nach einem geeigneten Datenformat einher. Es gilt somit ein Datenformat zur Beschreibung dreidimensionaler Stadtmodelle zu spezifizieren. Kolbe und Gröger [KG03] arbeiten mit der *Special Interest Group „3D“ (SIG 3D)* der Initiative „Geodaten Infrastruktur Nordrhein-Westfalen“ (*GDI NRW*⁶) an der Erstellung eines solchen Standards. Eine standardisierte Formatspezifikation zur Steigerung der Interoperabilität ist notwendig, da bisher nur Insellösungen realisiert wurden, die proprietäre Formate nutzen. Wichtig ist an dieser Stelle auch die Modellierung von unterirdischen Gebäuden wie Fußgängerunterführungen oder U-Bahnsystemen, die bislang häufig außer Acht gelassen wurde.

Zlatanova et al. [ZRP02] testen die 3D-Tauglichkeit der GIS-Systeme⁷ *ArcView 3D Analyst (ESRI)*, *Imagine VirtualGIS (ERDAS)*, *GeoMedia Terrain (Intergraph)*, *PAMAP GIS Topograph (PCIGeomatics)* und kommen zu dem Ergebnis, dass diese Systeme nur sehr spärliche 3D-GIS-Funktionalität bieten. Dieser Mangel führt dazu, dass 3D-Modelle häufig mit Programmen des Computer-Aided Design (CAD) oder mit Visualisierungs-Tools erstellt werden, die mit Datenbanken verbunden sind. Zlatanova et al. merken an, dass gerade die Kombination von CAD und DMBS häufig zu Inkonsistenz der Daten führt.

2.3 3D-Geodatenbanken – eine Alternative?

Da Geoinformationssysteme den 3D-Daten nicht in ausreichendem Maße gerecht werden und CAD keine GIS-Funktionalität bietet, müssen andere Lösungen zur Verwaltung und Nutzung von 3D-Daten gefunden werden. Es existieren eine Reihe von Arbeiten, welche die GIS-Funktionalität von Datenbankmanagementsystemen für 3D-Daten untersuchen.

Einige DBMS, wie das in dieser Arbeit verwendete System der Fa. *Oracle*, aber auch *IBM DB2*, *Informix* und *Ingres* bieten in den neusten Versionen Datentypen für räumliche Daten (vgl. [SZ03]). Diese sind aber auf Punkt-, Linien- und Flächen-Datentypen wie 3D-Punkt, 3D-Linie und 3D-Polygon beschränkt. Objekttypen zur Beschreibung von Volumen fehlen gänzlich.

Die Gruppe um van Oosterom am *Geo-Database Management Center (GDMC)*⁸ der TU Delft hat umfangreiche Versuche unternommen, Volumengeometrien mit Hilfe dieser Flächen-Datentypen in DMBS abzulegen und mit räumlichen Datenbank-Operatoren GIS-Funktionalität zu erreichen.

Grundsätzlich stellen van Oosterom et al. [OSQZ02] fest, dass nur objektorientierte DBMS aufgrund implementierbarer Methoden flexibel genug sind, um komplexe räumliche Daten zu verwalten. So ist es mit relationalen DBMS beispielsweise nicht möglich die Konsistenz der Daten zu prüfen.

⁶<http://www.gdi-nrw.org>

⁷Hersteller Angaben in Klammern. Alle Hersteller sind im Internet unter <http://www.<herstellername>.com> erreichbar.

⁸<http://www.gdmc.nl>

Quak et al. [QST03] beschreiben ein relationales Datenbankschema für *Oracle* auf Grundlage der „geflügelte Kanten“-Struktur. In diesem Schema werden die topologischen Zusammenhänge der Geometrien implizit gespeichert. Es kommen ausschließlich Standard-Datentypen wie `NUMBER` zur Anwendung.

Räumliche Analysen sind auf topologisch gespeicherten Daten nur sehr schwer durchzuführen. Quad et al. stellen eine implementierte Methode vor, welche die Topologien in den Oracle-Objekttyp für räumliche Daten überführt. Für diese Methode kann ein räumlicher Index erzeugt werden, der räumliche Analysen unterstützt.

Arens et al. [ASB03] verwenden in ihrer Arbeit ebenfalls ein topologisch orientiertes Datenbankschema, in dem auch räumliche Datentypen verwendet werden. Sie implementieren grundlegende GIS-Funktionalität durch in der Datenbank gespeicherte Methoden und Prozeduren. Zugleich weisen sie jedoch darauf hin, dass solche 3D-Algorithmen wesentlich komplexer sind als entsprechende 2D-Berechnungen. Daher muss besonders auf die Effizienz dieser Algorithmen geachtet werden.

Zusammenfassend kann aus allen vorgestellten Arbeiten gefolgert werden, dass auch 3D-(Geo)Datenbanken keinen ausreichenden Umfang an GIS-Funktionalität für 3D-Daten zur Verfügung stellen. Sie scheinen jedoch besser geeignet zu sein, als klassische GIS-Software. Unzulänglichkeiten der 3D-Datenbanken bestehen vor allem darin, dass keine echten 3D-Datentypen wie z.B. Polyeder existieren. Darüberhinaus können Funktionen wie Abstandsberechnung oder Ermittlung der topologischen Beziehung nur auf zweidimensionale Daten oder die Projektion von 3D-Daten in die Ebene angewendet werden. Aus diesem Grund müssen

- DB-Schemata entwickelt werden, welche die topologischen Aggregationen der Objekte explizit speichern, und
- Methoden implementiert werden, die benötigte Berechnungen auch auf 3D-Daten ausführen.

2.3.1 Abgrenzung dieser Arbeit

In dieser Arbeit wird das von Reuter [Reu03] vorgestellte und am Institut für Kartographie und Geoinformation der Universität Bonn (IKG) in einer Oracle-DB implementierte Datenbankschema untersucht und weiterentwickelt.

Es werden im Gegensatz zu den oben vorgestellten Arbeiten *Objekttabellen*⁹ für benutzerdefinierte Objekte wie *Knoten*, *Kante*, *Fläche*, *Volumen* verwendet. Höherdimensionale Geometrien (z.B. Volumen) werden mit Hilfe von *Collections* aus niedrigerdimensionalen Geometrien (Flächen) aggregiert.

Sämtliche Geometrien werden zusätzlich zur topologischen Aggregation in entsprechenden Oracle-Objekttypen gespeichert, um effiziente räumliche Abfragen zu ermöglichen. Zur Abfrage der Daten von der Datenbank werden DB-Funktionen für räumliche Daten

⁹Objekttabellen werden in Kapitel 5.1.2 ausführlich beschreiben. An dieser Stelle sei angemerkt, dass sie sich im Wesentlichen durch objektorientierte Ansätze wie abstrakte Objektdefinition, Attribute und Methoden von relationalen Tabellen unterscheiden.

verwendet, deren Ergebnisse dann durch externe Applikationen mit DB-Verbindung um die dritte Dimension ergänzt werden.

Das vorhandene DB-Schema wurde insbesondere um die Möglichkeit der Texturspeicherung erweitert. Flächengeometrien können Oberflächenbeschreibungen in Form von Farbdefinitionen oder Rasterbildern zugeordnet werden. Hierzu wird ein Oracle-eigener Datentyp für Rasterbilder verwendet.

Kapitel 3

Abstrakte Modellierung von DGM

Der Entwicklung einer Software oder eines Datenbankschemas sollte stets eine Modellierungsphase vorangehen, in der ein Datenmodell erstellt wird, das die Realwelt formalisiert (meist graphisch), vereinfacht und abstrakt darstellt.

Im Bereich des Software Engineerings für objektorientierte Programmiersprachen ist dies das Standardvorgehen, das zur Entwicklung spezifischer Werkzeuge wie der *Unified Modeling Language* (UML, [Obj03]) geführt hat.

Im Folgenden wird ein solches Datenmodell zur Beschreibung der Geländeoberfläche vorgestellt und anschließend in den Gesamtzusammenhang eines 3D-Stadtmodells eingegliedert.

3.1 Modell der Geländeoberfläche

Die formale und abstrakte Beschreibung der Geländeoberfläche ist in zwei Teile zu gliedern:

- Die Beschreibung der *Geometrie* der Oberfläche.
- Die Beschreibung der *Beschaffenheit* bzw. des Aussehens der Oberfläche.

Diese Teile werden nun erläutert und zueinander in Beziehung gesetzt.

3.1.1 Modellierung der Geometrie

Die geometrische Beschreibung des Reliefs ist Teil des von Plümer et al. vorgestellten „Datenmodells der 3D-Geo-Objekte für eine multi-funktionale Nutzung“ [PFK⁺02]. In diesem Modell wird die Geometrie der Geländeoberfläche als DGM repräsentiert, das aus mindestens einem TIN (vgl. Kapitel 2.1.1) besteht. Die TIN-Repräsentation ist mit Blick auf die spätere Visualisierung aufgrund der Planarität der Flächen sehr vorteilhaft (vgl. Kapitel 4).

Ein TIN besteht aus mindestens einem Dreieck, das eine spezielle Art eines Flächenobjekts ist. Flächengeometrien werden von Plümer et al. zur weiteren Abstraktion unter Berücksichtigung der topologischen Zusammenhänge aus Knoten, Kanten, Ringen und Maschen modelliert. Es ergibt sich das in Abbildung 3.1 gezeigte Modell, dessen Multiplizitäten wie folgt ausgedrückt werden können:

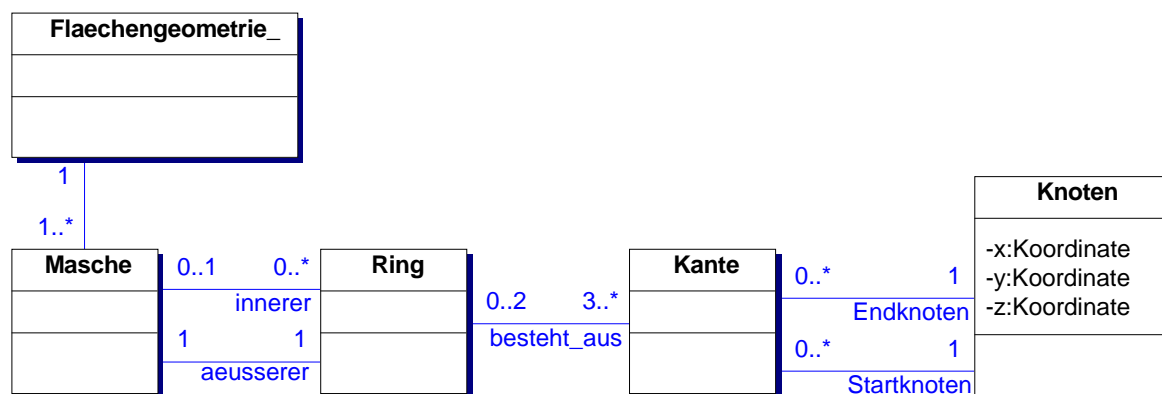


Abbildung 3.1: UML-Diagramm des topologischen Datenmodells für *allgemeine Flächengeometrien* nach Plümer et al. [PFK⁺02].

Eine Masche besitzt einen äußeren Umring und ggf. mehrere innere Ringe (Löcher), die aus mindestens drei Kanten mit jeweils einem Anfangs- und einem Endknoten bestehen. Diese explizite topologische Modellierung hat den Vorteil, dass Daten redundanzfrei gespeichert werden können und die Geometrie der Flächenobjekte implizit in den referenzierten Knoten gespeichert wird. So muss beispielsweise ein Knoten, der Eckpunkt mehrerer Maschen und somit Anfangs- oder Endpunkt mehrerer Kanten ist, nur einmal gespeichert werden. Alle Kanten verweisen auf die gespeicherte Instanz des Knotens und alle Ringe auf die entsprechenden Instanzen der Kanten. Somit wird auch klar, dass nur der Knoten Geometrieinformationen besitzt. Alle übrigen Objekte speichern lediglich topologische Informationen.

TIN-Dreiecke können jedoch gesondert modelliert werden, da sie spezielle Maschen sind, die keine inneren Ringe haben und deren äußerer Ring aus genau drei Kanten besteht. Daher entfallen die Klassen Ring und Masche und es entsteht eine direkte Aggregation aus Dreiecken und Kanten. Ein TIN besteht aus mindestens einem Dreieck und ist Bestandteil eines DGMs (vgl. Abbildung 3.2).

3.1.2 Modellierung der Beschaffenheit

In der Arbeit von Plümer et al. ist eine Aggregation der Klasse **Materialeigenschaft**, die der Beschreibung des Aussehens des Geländes dient, mit der Klasse **TIN** nicht vorgesehen. Diese wird dort nur mit der Klasse **Masche** aggregiert. An dieser Stelle wird das Modell ergänzt. Die Beschaffenheit der Geländeoberfläche wird ebenfalls mit der **Materialeigenschaft** aggregiert, die entweder eine Farbdefinition oder eine Kachel (s.u.) eines georeferenzierten Rasterbildes sein kann. Eine **Materialeigenschaft** wird einem

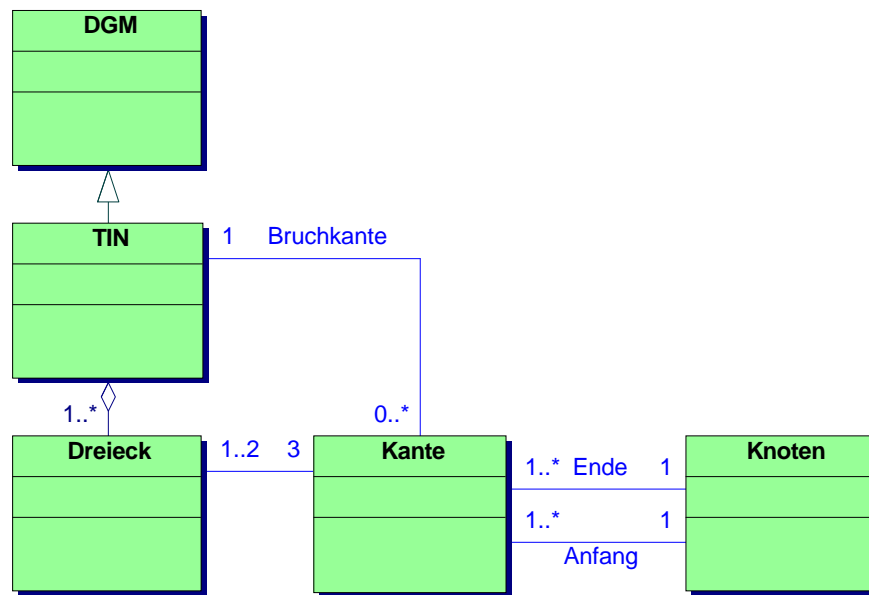


Abbildung 3.2: UML-Diagramm des topologischen Datenmodells der *Geometrie der Geländeoberfläche* nach Plümer et al. [PFK⁺02].

oder mehreren Dreiecken zugeordnet (vgl. Abbildung 3.3).

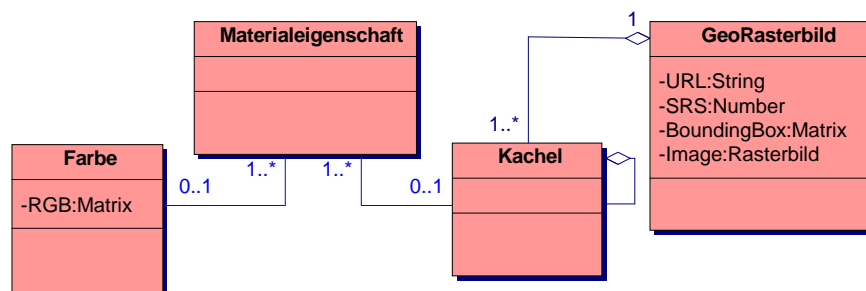


Abbildung 3.3: UML-Diagramm zum Datenmodell der *Beschaffenheit der Geländeoberfläche*.

Die Klasse des georeferenzierten Rasterbilds (*GeoRasterbild*) attribuiert nicht nur das Bild an sich, sondern auch dessen Bezug zu einem erdfesten Koordinatensystem (spatial reference system (*SRS*)) durch das Attribut *BoundingBox*. Das *SRS* kann beispielsweise durch die von der *European Petroleum Survey Group*¹ (*EPSG*) spezifizierte numerische Codierung angegeben werden. Das Gauß-Krüger-Koordinatensystem für den dritten Streifen trägt dort z.B. die Nummer 31467. Sämtliche Koordinatensysteme sind bereits im Datenbankmanagementsystem Oracle implementiert (vgl. Kapitel 5.3.2). Die *BoundingBox* gibt ein Koordinatenfenster an, dessen Inhalt vom Rasterbild abgedeckt wird. In der Regel wird es sich bei den georeferenzierten Rasterbildern um Orthophotos² handeln.

¹<http://www.epsg.org>

²Orthophotos sind entzerrte Luftbilder. Luftbilder entsprechen aufgrund der Radialverzerrung der

3.1.3 Zusammenführen der Modelle

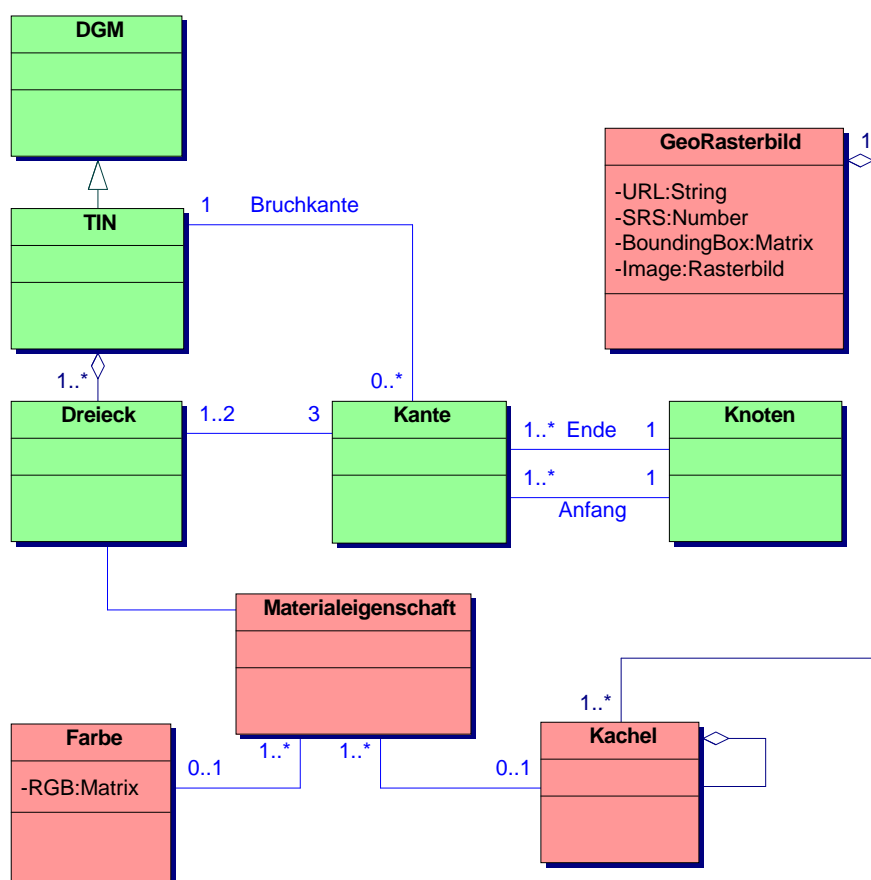


Abbildung 3.4: UML-Diagramm der zusammengeführten Datenmodelle zur Beschreibung der Geländeoberfläche durch ihre Geometrie und Beschaffenheit.

Die Abbildung 3.4 zeigt die Zusammenführung der beiden Teilbereiche der Oberflächenmodellierung. Die Klasse **Dreieck** stellt die Verbindung zwischen Geometrie und Beschaffenheit her, da ein Dreieck als Teil des TINs die Geometrie der Oberfläche beschreibt und ihm zugleich eine Materialeigenschaft zugewiesen werden kann.

3.2 Integration in das Gesamtmodell

Da diese Arbeit als Erweiterung der Arbeit von Reuter [Reu03] zu sehen ist, muss das oben vorgestellte Datenmodell an jenes von Reuter angepasst werden. Ebenso muss sich das in Kapitel 5.4 abgeleitete Datenbankschema an der am IKG implementierten DB orientieren. Die bestehende Datenbank und alle darauf zugreifenden Anwendungen dürfen hierzu lediglich ergänzt, nicht jedoch abgeändert werden.

Linse nur in der Nähe des Bildmittelpunktes einer Zentralprojektion. Unter Berücksichtigung der Aufnahmegeometrie und des Geländemodells werden sie derart verzerrt, dass jeder Bildpunkt einer Senkrechtaufnahme entspricht.

Reuter verwendet in seiner Arbeit das eingangs erläuterte Datenmodell von Plümer et al., das im Folgenden kurz als *Basismodell* bezeichnet wird. Da sich diese Arbeit auf die Integration des digitalen Geländemodells beschränkt wird auf eine detaillierte Erläuterung des Basismodells verzichtet. Eine ausführliche Beschreibung findet sich in den Arbeiten von Plümer et al. [PFK⁺02] und Reuter [Reu03]. Zur Integration des Modells der Geländeoberfläche in das Basismodell muss letzteres bzgl. der Modellierung der Materialeigenschaft modifiziert werden.

Im Basismodell ist die Materialeigenschaft mit der Klasse **Masche** aggregiert. Eine Flächengeometrie besteht entweder aus einer Aggregation von Maschen oder aus einer einzelnen Masche. Flächengeometrien können wiederum zu Volumenkörpern aggregiert werden, die i.d.R. Bauwerke repräsentieren. Es kann nun der Fall eintreten, dass eine Gebäudewand aus mehreren Maschen besteht. Eine terrestrische Fassadenaufnahme des Gebäudes wird jedoch typischerweise die gesamte Fassade darstellen. Demnach ist eine Texturierung nicht möglich, da nur Maschen Materialeigenschaften (Rasterbildtexturen) haben können. Es scheint also sinnvoll, die Klasse **Materialeigenschaft** nicht der Klasse **Masche**, sondern der Klasse **Flaechengeometrie** zuzuweisen. Zur Beschreibung einer Fassadenoberfläche mit Rasterbildern sind andere Informationen bzgl. der Transformation bzw. Verzerrung des Rasterbildes notwendig, als zur Beschreibung der Geländeoberflächen. Aus diesem Grunde muss die Klasse **Materialeigenschaft** derart verändert werden, dass sie terrestrische Fassadenaufnahmen und Orthophotos gleichermaßen repräsentiert. Zu diesem Zweck wird der Materialeigenschaft neben dem Farbwert eine allgemeine Klasse **Phototextur** zugewiesen. Diese kann sowohl eine Kachel eines georeferenzierten Rasterbilds, als auch eine Fassadentextur sein. Das Konzept der Kachelung wird in Kapitel 4.3.2 ausführlich abgehandelt, hier sei zum besseren Verständnis angemerkt, dass ein Rasterbild durch Kachelung im Sinne einer Tessellation in mehrere Einzelbilder zerlegt wird. Die Klasse des georeferenzierten Rasterbilds wird im UML-Diagramm in Abbildung 3.5 grafisch auf einer Höhe mit den Objekten der „Aggregationsebene“ (vgl. [PFK⁺02]) angeordnet und der abstarkten Oberklasse **OberflaechenBeschreibung** zugeordnet. Die Veränderungen gegenüber dem Basismodell sind rot gekennzeichnet.

Kapitel 4

Texturierte Darstellung von Geometrien

Digitale Geländemodelle bieten einen großen Mehrwert in Bereichen wie der Berechnung von Oberflächenabfluß oder der Erforschung schwer zugänglicher Geländebereiche (etwa Gletscher oder verseuchte Gebiete). Ohne Berücksichtigung der Geländeoberfläche wäre die Darstellung eines 3D-Stadtmodells äußerst ernüchternd.

Die grafische Darstellung des DGMs, die zur Steigerung des Wiedererkennungsgrades im besten Fall photorealistisch gestaltet sein sollte, wird im Folgenden näher betrachtet. Nach Vorstellung grundlegender Konzepte und des normierten *VRML*-Standards wird die Verwendung von Orthophotos zur Oberflächenbeschreibung thematisiert und ein Verfahren zur Optimierung des Speicherbedarfs vorgestellt.

4.1 Grundlagen der Texturierung

Zur grafischen Darstellung eines Volumenkörpers (3D-Objekt) können bildliche Darstellungen des Objekts an eine Fläche der Geometrie „angebracht“ werden. Dieser Vorgang, 2D-Rasterbilder auf 3D-Objekte zu projizieren, wird als *Texturierung* bezeichnet. Im Bereich der Computergrafik werden die folgenden Arten von Texturierungen unterschieden:

- **Texture Mapping** – Ein Rasterbild bzw. die Farbinformationen der Pixel werden auf die Geometrie projiziert. Es entsteht ein realistischer Eindruck.
- **Procedural Mapping** – Durch Algorithmen erzeugte Muster, die fortlaufend auf die Geometrien abgebildet werden. Dieses Verfahren dient der Texturierung von Oberflächen mit sich wiederholendem Muster.
- **Bump Maps** – Texturbilder enthalten Farbwerte, die Tiefeninformationen (Normalenvektor-Richtung) der Geometrieoberfläche entsprechen, also eine Art „Schattierungsbild“. So wird eine detailliertere Darstellung ermöglicht ohne die Geometrie selber mit hohem Detailgrad modellieren zu müssen.

Bei TINs beschränkt sich die Auswahl der zu texturierenden 3D-Objekte auf Dreiecke und es werden typischerweise Rasterbilder zur Texturierung verwendet. Jedem Dreieck müssen also Bildinformationen zugewiesen werden, die in Bilddateien gespeichert sind. Dies führt zu folgendem Problem: Rasterbilder können nur in rechteckiger Form gespeichert werden und haben somit mehr Bildinformationen als für die Beschreibung eines Dreiecks notwendig ist. Abbildung 4.1 verdeutlicht diese Diskrepanz.

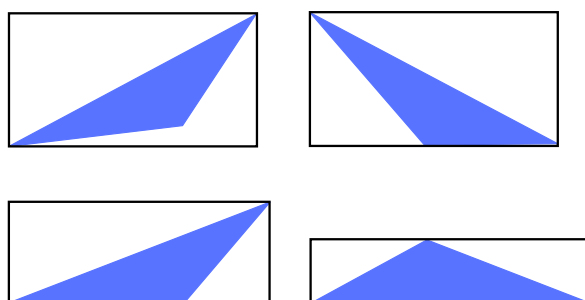


Abbildung 4.1: Dargestellt sind vier Möglichkeiten, dem Dreieck ein Rechteck zuzuordnen. Gesucht wird das Rechteck mit dem kleinsten Flächeninhalt. Die optimale Lösung liegt bei einem Flächenverhältnis von 1:2 zwischen Dreieck und Rechteck vor (rechts unten).

In allen vier Fällen handelt es sich um das selbe Dreieck. Im ersten Bild (li.o.) ist es beliebig orientiert. In den übrigen Bildern wird jeweils eine Seite des Dreiecks waagrecht orientiert. Die Fläche des Rasterbilds (hier als Rechteck dargestellt) ist im Bild links oben dreimal, im Bild rechts unten nur doppelt so groß wie die Fläche des Dreiecks. Zur Optimierung des Speicherbedarfs scheint es demnach sinnvoll das Rasterbild durch Rotation nach der längsten Dreiecksseite auszurichten und durch „Zurechtschneiden“ an dessen Grundseitenlänge und Höhe anzupassen. So läßt sich die Flächendifferenz zwischen Dreieck und Rechteck (Rasterbild) auf maximal 100% verringern¹. Demnach kann es trotz der beschriebenen Transformation der Rasterbilder bei der Texturierung eines großen TINs mit einigen 10.000 Dreiecken zu Effizienzproblemen und möglicherweise zur Überlastung des Grafikspeichers kommen, wenn ein Rasterbild pro TIN-Dreieck verwendet wird. Es werden mindestens doppelt so viele Bildinformationen geladen als für die Texturierung notwendig sind.

Dies soll lediglich das grundlegende Problem der Verwendung von Rasterbildern zur Texturierung von nicht rechteckigen Geometrien veranschaulichen. Da ein TIN ein zusammenhängendes Netz aus Dreiecken darstellt, können die „überflüssigen“ Bildinformationen des Texturbildes eines Dreiecks natürlich für die Texturierung des Nachbardreiecks verwendet werden. Eine Fortführung dieses Gedankens führt zu der Schlußfolgerung, dass ein gesamtes TIN mit nur einem Rasterbild texturiert werden könnte. Dieser Gedanke wird unten in Kapitel 4.3 wieder aufgegriffen. Der benötigte Speicherbedarf für die Texturierung mit Rasterbildern hängt natürlich auch von der Größe der Bilddateien ab. Da der Bildausschnitt mit den oben beschriebenen Transformationen auf die Geometrie angepasst wurde, bleibt die Auflösung des Bildes der entscheidende

¹Die Fläche eines Dreiecks berechnet sich nach der Formel $Fläche = 1/2 * Grundseite * Höhe$. Entspricht die Grundseite des Dreiecks einer Rechtecksseite und ist die Dreieckshöhe so lang wie die andere Rechtecksseite, dann ist die Fläche des Dreiecks halb so groß wie die Rechtecksfläche.

Faktor, der das Speichervolumen bestimmt. Es gilt also einen Kompromiss zwischen Qualität, Geschwindigkeit und Realismus der Darstellung zu finden.

Weinhaus und Devarajan [WD97] nennen neben *Texture Mapping* weitere Verfahren wie *Image Perspective Transformation (IPT)* oder *Perspective Photo Mapping* für die Abbildung von 2D-Grafiken auf 3D-Objekte. Anhand der Bezeichnungen wird deutlich, dass für diese Abbildungsverfahren zwei Transformationen notwendig sind (vgl. Abbildung 4.2).

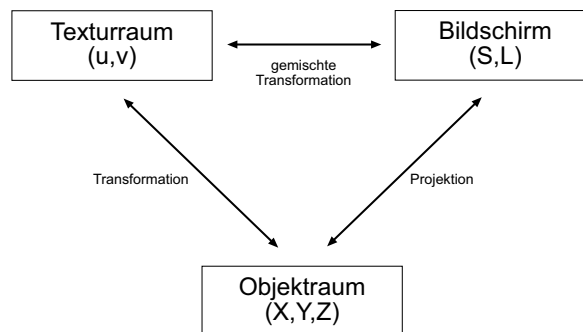


Abbildung 4.2: Bei der Texturierung notwendige Transformationen (aus [WD97]).

Im ersten Schritt müssen die Texturkoordinaten des Rasterbilds in den Objektraum der Geometrie transformiert werden (2D nach 3D). Im zweiten Schritt werden die Koordinaten der Geometrie im Objektraum in die Bildschirmenebene projiziert (3D nach 2D). Die Art der ersten Transformation (Texturkoordinaten – Objektraum) wird z.B. beim Programmieren einer *VRML*-Szene (vgl. Kapitel 4.2) explizit vom Benutzer angegeben. Die zweite Transformation (Objektraum – Bildschirm) berechnet eine entsprechende Viewer-Software in Abhängigkeit der Position des virtuellen Beobachters im Objektraum.

4.2 VRML97

Zur Beschreibung und Visualisierung dreidimensionaler Geometrien wurde vom *Web3D Consortium (W3C)* die *Virtual Reality Modeling Language* entwickelt, die 1997 als ISO-Standard² definiert wurde (*VRML97*, vgl. [Web97]). Im Dezember 2003 wurde der Nachfolgerstandard *Extensible 3D (X3D)* veröffentlicht, der eine Weiterentwicklung von *VRML97* darstellt.

In dieser Arbeit wird der *VRML97* Standard zur Beschreibung und Visualisierung des digitalen Geländemodells verwendet, da er insbesondere im Internet weit verbreitet ist. Dateien können aufgrund ihres Textformats mit jedem Texteditor bearbeitet und mit einem Internetbrowser³ dargestellt werden. *VRML* ist ein plattformunabhängiges,

²ISO/IEC 14772-1:1997

³Hierzu ist ein Plugin notwendig. Es existieren für den Zweck der *VRML*-Darstellung aber auch Standalone Applikationen. Für diese Arbeit wurde das *Cortona VRML Client Plugin Version 4.2 Release 93* der Fa. Parallelgraphics (<http://www.parallelgraphics>) in Verbindung mit dem InternetExplorer Version 6.0 der Fa. Microsoft (<http://www.microsoft.com>) verwendet.

interoperables Format.

4.2.1 Grundlagen

Eine *VRML*-Datei beschreibt eine dreidimensionale *Szene* durch eine Liste von Objekten, die als *Knoten* bezeichnet werden. Es wird zwischen *Gruppenknoten*, *Kindknoten* und *Objektknoten* unterschieden, die zusammen einem **SzeneGraph** zugeordnet sind. Einem Gruppenknoten können mehrere Kindknoten zugeordnet sein. Veränderungen der Eigenschaft des Gruppenknotens (z.B. Translation oder Rotation) wirken sich auf alle ihm zugeordneten Kindknoten aus (Vererbung). Objektknoten dienen der Erzeugung und Gestaltung der geometrischen Objekte.

Durch Knoten wie **Background**, **DirectionalLight**, **PointLight**, **SpotLight** und **NavigationInfo** ist es möglich, den Hintergrund, die Art der Beleuchtung oder die Betrachtungsweise (gehen, fliegen usw.) genauer zu spezifizieren.

Die Grundstruktur einer *VRML*-Datei soll anhand von Listing 4.1 verdeutlicht werden – Zeilenangaben des Listings in Klammern „()“.

Zu Beginn jeder Datei muss die Kopfzeile (1) angegeben sein, welche die Version der Sprachdefinition und das Codierungsformat beschreibt. Kommentare werden in *VRML* mit # eingeleitet.

Die dargestellte Szene besteht aus den zwei Gruppenknoten **Transform** (3-25) und **Background** (27-29). Ein Background-Knoten ist ein s.g. *bindable* Knoten, der nur einmal existieren darf. Er gibt die Hintergrundfarbe im RGB-Farbraum⁴ an (hier blau). Der Transform-Knoten legt die in den entsprechenden *Feldern* beschriebenen Transformations- (4) und Rotationsparameter (5) fest (hier beide 0), die für alle Kindknoten Geltung haben.

Es existiert nur der Kindknoten **Shape** (7-23), der durch den Objektknoten **appearance** (8-12) und das Feld **geometry** (13-22) näher beschrieben wird.

Der Appearance-Knoten kann durch den Objektknoten **Material** (9-11) ausgestaltet werden. Er dient zur Definition einer Oberflächenfarbe der Geometrie des Shape-Knotens (hier RGB-Wert für grün).

Der Geometry-Knoten kann die Geometrie des Shape-Knotens z.B. als **IndexedFaceSet**-Knoten (13) definieren. Das ist eine Fläche, die durch beliebige 3D-Punkte begrenzt wird. Zur Beschreibung eines TINs werden Dreiecke, also Polygone, benötigt. Diese werden durch den IndexedFaceSet-Knoten repräsentiert. Der Knoten muss zwingend die Felder **coord** (14-20) und **coordIndex** (21) enthalten.

Im ersten Feld werden die Koordinaten der Eckpunkte definiert. Koordinatentripel werden durch Kommata getrennt und (x,y,z) -Werte durch Leerzeichen. Die Reihenfolge ist hier beliebig. Im zweiten Feld (**coordIndex**) wird die Punktfolgenfolge des Polygons durch die Indizes der Koordinaten des coord-Feldes (beginnend bei 0) beschrieben. Aus allen Punkten können beliebig viele Flächen definiert werden. Dazu wird das Ende eines Polygons im CoordIndex-Feld mit '-1' gekennzeichnet, worauf beim letzten Poly-

⁴Im RGB-Farbraum wird eine Farbe durch Anteile von *Rot*, *Grün* und *Blau* festgelegt, die i.d.R. durch Werte im Bereich von 0 bis 255 gegeben werden. Das entspricht einer Farbtiefe von 8 Bit ($256 = 2^8$). In *VRML* beziehen sich alle Farbdefinitionen auf diesen Farbraum.

```

1 #VRML V2.0 utf8
2
3 Transform{
4   translation 0 0 0
5   rotation 0 0 0 0
6   children[
7     Shape { #Shape
8       appearance Appearance {
9         material Material{
10          diffuseColor 0 0.3 0
11          } #material
12        } #appearance
13       geometry IndexedFaceSet {
14         coord Coordinate {
15           point [
16             4 2 7,
17             3 0 5,
18             8 1 3
19           ] #point
20         } #coord
21         coordIndex [0 , 1 , 2]
22       } #geometry
23     } #Shape
24   ] #children
25 } #Transform
26
27 Background {
28   skyColor 0.0 0.0 0.5
29 } #Background

```

Listing 4.1: Beispiel einer VRML-Datei.

gon verzichtet werden kann.

Nach der „Daumenregel der rechten Hand⁵“ wird die Richtung des Normalenvektors festgelegt. Es wird die Seite der Geometrie texturiert, auf welcher der Normalenvektor steht.

4.2.2 Rasterbild-Texturen

Der *VRML*-Standard bietet genau spezifizierte Funktionalitäten zur Texturierung von Flächen oder Volumenkörpern. Der Benutzer muss notwendige Transformationen des Rasterbildes nicht vorab durchführen und somit das Rasterbild an sich verändern.

⁵Die Daumenregel der rechten Hand besagt in diesem Zusammenhang, dass bei geballter rechter Faust und ausgestrecktem Daumen die Umlaufrichtung der Eckpunkte einer Fläche in Richtung der Finger verläuft und der Normalenvektor durch die Richtung des Daumens festgelegt ist.

Sollen die Geometrien mit Rasterbildern⁶ texturiert werden, so ist im Appearance-Knoten der Kindknoten `texture` zu definieren. Dieser muss vom Typ `ImageTexture` sein und im Feld `url` den Speicherort der Rasterbilddatei festlegen (vgl. Listing 4.2).

```
# --- Beginn des Codeausschnitts ---

appearance Appearance {
  texture ImageTexture {
    url "meinRasterbild.jpg"
  } #texture
} #appearance

# --- Ende des Codeausschnitts ---
```

Listing 4.2: Struktur des ImageTexture-Knotens.

In *VRML* stehen drei Möglichkeiten der Texturierung zur Verfügung, die das Rasterbild auf unterschiedliche Art und Weise auf die Geometrie transformieren. Im Folgenden werden diese Methoden der Einfachheit halber für 2D-Objektkoordinaten ($z = 0$) beschrieben. Alle Konzepte können identisch für 3D-Objekte verwendet werden.

In den Beispielen wird die in Abbildung 4.3 gezeigte Textur (links) auf das Dreieck (rechts) abgebildet⁷. Zur Verdeutlichung der Transformationen wurde dieses symmetrische Texturbild gewählt, da transformierte Ausschnitte somit leichter zum Originalbild zugeordnet werden können.

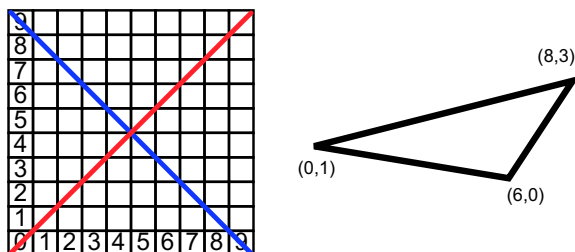


Abbildung 4.3: Ausgangssituation der folgenden Beispiele: Das Dreieck (rechts) soll mit dem nebenstehenden Rasterbild (links) texturiert werden.

⁶Ein Ausführung zur Verwendung von Rasterbilder in *VRML*: In der Literatur, die sich mit dem Erstellen von *VRML*-Welten befaßt, und in Internetforen zu diesem Thema ist immer wieder zu lesen, dass möglichst Rasterbilder verwendet werden sollen, deren Seiten eine Länge von 2^n haben. Dies habe den Hintergrund, dass anders formatierte Bilder im Viewer auf entsprechende Größe umgerechnet werden müssten (rendering). Um diesen Rechenaufwand zu umgehen, sollten stets Bilder mit der „richtigen“ Bildgröße verwendet werden.

Diese Behauptung sollte durch geeignete Daten empirisch untersucht werden. Das ist im Rahmen dieser Arbeit nicht geschehen. Es wurde, soweit wie möglich, aber stets versucht Rasterbilder mit einer Seitenlänge von 2^n zu verwenden. Eine Beschreibung der Einschränkungen finden sich in Kapitel 4.3.2.

⁷Anders als bei Weinhaus und Devarajan (vgl. Kapitel 4.1) werden in der *VRML*-Spezifikation die Achsen des Texturkoordinatensystems mit s, t bezeichnet.

Freie Texturierung

Die einfachste Möglichkeit das Dreieck zu texturieren, ist die Art der Texturierung nicht näher zu spezifizieren. Wird lediglich ein ImageTexture-Knoten erzeugt, so wird das Koordinatensystem des Texturraums durch die folgenden Schritte festgelegt:

- Bounding Box⁸ (BB) um das Dreieck bestimmen.
- Die s -Koordinatenachse ist die längste Seite der BB (max_1), die t -Achse die zweitlängste (max_2).
- Der Koordinatenursprung wird auf $(0, 0)$ gesetzt.
- Die s -Koordinatenachse hat den Maximalwert 1.
- Auf die t -Koordinatenachse wird der gleiche Maßstab wie auf die s -Achse angewendet. Der dem Ursprung diagonal gegenüberliegende Punkt der BB hat demnach die Koordinaten $(1, max_2/max_1)$.

Die Abbildung 4.4 macht die Skalierung der Achsen deutlich (die BoundingBox hat die Größe 8×3). Die s -Achse wird auf 1 und die t -Achse auf $3/8 = 0,375$ skaliert, da auf beide Achsen der gleiche Maßstab angewendet wird. Der VRML-Code dieses Beispiels findet sich in Anhang A.1.

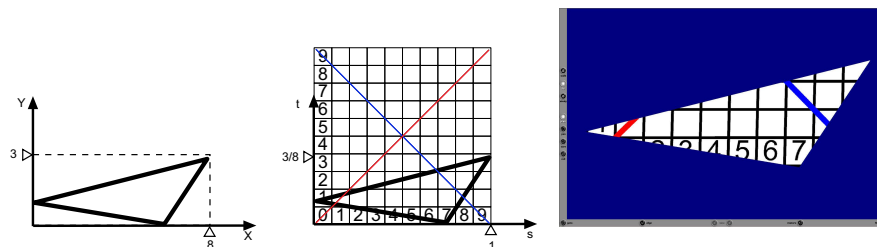


Abbildung 4.4: VRML-Texturierung ohne Angabe von Parametern. Bestimmen der BoundingBox (links), Skalieren der Achsen des Texturkoordinatensystems (Mitte) und Ergebnis im Browser (rechts).

Im 3D-Fall ist die BoundingBox tatsächlich als 3D-Objekt zu verstehen, das um die 3D-Fläche gelegt wird. Das Texturkoordinatensystem wird dann an der längsten und der zweitlängsten Seite der BoundingBox ausgerichtet. Beispielsweise ist bei einer senkrechten Wand eines dreidimensional modellierten Gebäudes die BoundingBox wieder eine Fläche, da sie die Tiefe 0 hat. Die Textur wird also im Sinne einer Orthogonalprojektion auf die Geometrie abgebildet, denn die (s, t) und (X, Y) Koordinatenachsen liegen übereinander. Die Zuordnung der Achsen zueinander ist evtl. vertauscht ($s \rightarrow Y, t \rightarrow X$), da sie von den Ausdehnungen der Wand bzw. BoundingBox abhängen (s.o). Ein TIN-Dreieck hingegen liegt i.d.R. schräg in seiner BoundingBox, an der die

⁸Die Bounding Box ist im 2D-Fall das kleinste, das Objekt umschließende Rechteck, dessen Seiten parallel zu den Koordinatenachsen liegen. Im Dreidimensionalen wird ein Quader verwendet.

Textur ausgerichtet wird. Somit wird die Textur im Sinne einer projektiven Abbildung auf die Textur transformiert.

Vorgabe von Texturkoordinaten

Die Abbildung des Rasterbildes auf das Dreieck kann alternativ auch derart erfolgen, dass im Geometry-Feld das Feld `TextureCoordinate` eingefügt wird, in dem Koordinaten im Texturkoordinatensystem definiert werden. Die so definierten Texturpunkte entsprechen i.d.R. einem Eckpunkt der zu texturierenden Fläche. Hier sollte die Punkt-reihenfolge der Geometriedefinition angepasst sein, damit die Texturpunkte 1 : 1 den Geometriepunkten zugeordnet werden können.

Der Koordinatenbereich des Texturraums ist auf Werte im Intervall $[0, 1]$ beschränkt. Werden Werte außerhalb dieses Intervalls verwendet, so wird die Textur periodisch am Rand fortgesetzt und durch eine Affintransformation auf die Geometrie abgebildet (vgl. Abbildung 4.5). Der *VRML*-Code dieses Beispiels findet sich in Anhang A.2. Da das durch die Texturkoordinaten beschriebene Dreieck und das zu texturierende Dreieck nicht kongruent sind, ergibt sich die Verzerrung der Textur (links).

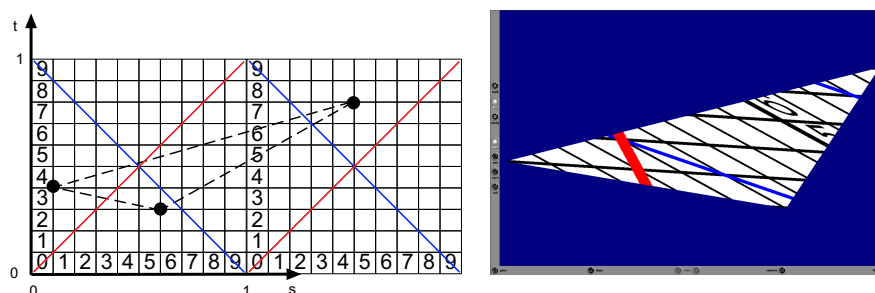


Abbildung 4.5: VRML-Texturierung durch Texturkoordinaten. Im linken Bild sind die Texturkoordinaten $(0.1/0.4, 0.6/0.3, 1.5/0.8)$ vorgegeben. Das Rasterbild wird bei Verwendung von Koordinaten außerhalb des Intervalls $(0, 1)$ am Rand periodisch fortgesetzt. Im rechten Bild ist die Visualisierung im *VRML*-Browser dargestellt.

Zur Implementierung der *VRML*-Schnittstelle der Datenbank wird der gerade beschriebene Ansatz verwendet. Da die Koordinaten der Dreiecke des TINs und die zur Texturierung verwendeten Orthophotos im gleichen Koordinatensystem vorliegen, müssen deren Koordinatenwerte zur Bestimmung von Texturkoordinaten lediglich „auf 1 normiert werden“.

Transformation der Textur

Ergänzend sei die Möglichkeit der Texturierung durch den Knoten `TextureTransform` genannt. Dieser wird als Kindknoten in den Appearance-Knoten eingefügt. Es können die Felder `center`, `rotation`, `scale` und `translation` definiert werden. Sie beschreiben Transformationen des Texturkoordinatensystems, die in der Reihenfolge

- Translation,

- Rotation um den Center-Punkt und
- Skalierung um den Center-Punkt

ausgeführt werden. Es ist zu beachten, dass diese Operationen auf das Koordinatensystem der Textur angewendet werden. Beispielsweise bewirkt eine Skalierung um den Faktor 2 eine Verdopplung der Länge der Achsen, nicht jedoch eine Verdopplung der Größe des Rasterbildes, welches nach der Skalierung im Intervall $[0, 0.5]$ liegt. Der Skalierungsfaktor kann auch als „Anzahl der Wiederholungen der Textur auf der Geometrie“ interpretiert werden. Der `TextureCoordinate`-Knoten kann mit dieser Texturierungsart kombiniert werden.

Ein *VRML*-Code für diese Art der Texturierung findet sich in Anhang A.3.

4.3 Orthophotos

Zur photorealistischen Visualisierung von DGM werden üblicherweise digitale Orthophotos verwendet. Im Bereich der Landesvermessung liegen beispielsweise für Nordrhein-Westfalen flächendeckend digitale Orthophotodaten im Blattschnitt der Deutschen Grundkarte 1:5000 (DGK5) ($2km \times 2km$) mit einer Bodenauflösung von $30cm$ (160 Pixel/cm) vor (DOP5)⁹. Diese können in schwarz/weiß (ca. 42MB/Datei) oder farbig (ca. 125MB) bezogen werden.

4.3.1 Georeferenzierung

Um den Bezug zwischen digitalem Orthophoto und den Koordinaten der Landesvermessung des abgebildeten Bereichs der Erdoberfläche herzustellen, müssen folgende Daten vorliegen:

- dem jeweiligen Koordinatensystem entsprechende Koordinaten einer Bildecke (i.d.R. links oben) – x_{LiO} , y_{LiO} ,
- Breite eines Pixels (picture element = pel) in X-Richtung in Metern und Pixel-Höhe in Y-Richtung – pel_B , pel_H und
- Rotationsparameter der Seiten des Orthophotos gegenüber den Koordinatenachsen der Landesvermessung – rot_X , rot_Y .

Diese Daten werden zeilenweise in der Reihenfolge (pel_B , rot_X , rot_Y , pel_H , x_{LiO} , y_{LiO}) in einer ASCII-Textdatei mit dem Dateinamen der entsprechenden Orthophotodatei und der Endung *wld*, *tfw*, *jpw* oder *pnw* gespeichert. Die letzten drei Endungen lassen auf das jeweilige Dateiformat der Orthophoto-Datei schließen (tiff, jpg, png).

Im Beispiel der DOP5 können die Rotationsparameter wegfallen, da die Orthophotos

⁹Quelle: Landesvermessungsamt NRW – <http://www.lverma.nrw.de>.

im Blattschnitt der DGK5 vorliegen. Somit sind die Seiten des Orthophotos achsparallel zum Gauß-Krüger-System. Die Koordinaten der linken unteren Ecke eines Orthophotos mit der Auflösung von $n \times m$ berechnen sich in diesem Fall nach den Formeln

$$\begin{aligned}x_{LiU} &= x_{LiO} , \\y_{LiU} &= y_{LiO} * m * pel_H .\end{aligned}$$

4.3.2 Optimierung des Speicherbedarfs

Wie oben erwähnt deckt ein Orthophoto im Maßstab 1:5000 eine Bodenfläche von $2km \times 2km$ ab. Wird in *VRML* mit diesem Orthophoto ein TIN texturiert, von dem nur einige wenige Dreiecke im Deckungsbereich des Orthophotos liegen, so muss trotzdem das gesamte Orthophoto in den Speicher des *VRML*-Viewers geladen werden. Dies führt bei einer Dateigröße von $125MB$ unausweichlich zum Überlauf des Grafikspeichers.

Ein weiteres Problem tritt auf, wenn ein TIN-Dreieck nur partiell von einem Orthophoto bedeckt wird. Am Bildrand der Rasterbilddateien ist dies die Regel. Die Texturierung einer solchen Situation ist im *VRML*-Standard nicht vorgesehen und muss umgangen werden. Aus diesem Grund werden nun Konzepte zur Lösung der beiden geschilderten Probleme vorgestellt.

Kachelung

Falls nur wenige Dreiecke eines TINs von einem Orthophoto beschrieben werden, ist es ineffizient, die gesamte Orthophotodatei laden zu müssen. Hilfe bietet das Verfahren der *Kachelung*, das in der Computergrafik und der Photogrammetrie häufig Anwendung findet. Hierzu wird das Rasterbild in mehrere kleine Teile zerlegt (vgl. Abbildung 4.6), die in einzelnen Dateien abgespeichert werden.

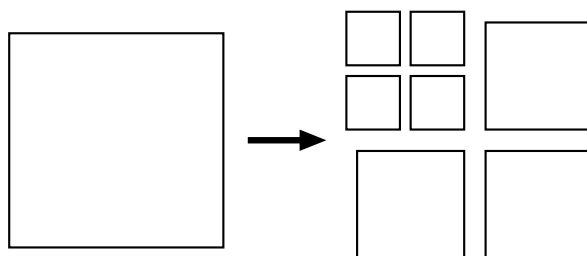


Abbildung 4.6: Bei der Kachelung wird das Orthophoto in gleich große Kacheln zerlegt.

Somit muss eine wesentlich kleinere Bilddatei geladen werden. Diese beschreibt die wenigen Dreiecke besser, als das gesamte (Original)Orthophoto. Häufig wird das Bild geviertelt, denn so bleiben die Seitenverhältnisse des Originalbildes erhalten. Die Kachelung kann rekursiv programmiert werden, wenn mehrere *Kachelungsstufen* erzeugt werden sollen. Beim Zerteilen des Bildes in vier Teile entstehen in der n -ten Stufe $4^n + 1$ Bilder; rechnet man das Originalbild (Stufe 0) hinzu.

Neben dem Orthophoto muss auch die Georeferenzierungsdatei an das neue Bildformat angepasst und gespeichert werden.

Die Kachelung eines Orthophotos kann dazu führen, dass die (unbestätigte) Forderung, ausschließlich Rasterbilder mit einer Seitenlänge 2^n zu verwenden, nicht erfüllt werden kann. Liegt das Originalbild in einem entsprechenden Seitenverhältnis vor und wird es in jeder Stufe geviertelt, so ist die Forderung erfüllt. In allen anderen Fällen (Ausgangsbild mit beliebiger Seitenlänge oder Kachelung in $x \neq 4$ Teile) ist eine Erfüllung der Forderung eher zufällig.

Verschneidung von Orthophoto und TIN

Unabhängig von der Kachelung der Orthophotos ist bei der Verwendung des *VRML*-Standards zur Texturierung eines TINs mit Orthophotos ein weiteres Problem zu lösen. Hat das TIN eine größere Flächenausdehnung als das Orthophoto, so müssen einem Dreieck, das am Rand des Orthophotos bzw. im Schnitt von vier Orthophotos liegt, bis zu vier unterschiedliche Texturen zugewiesen werden (vgl. Abbildung 4.7), was in der *VRML*-Spezifikation nicht vorgesehen ist.

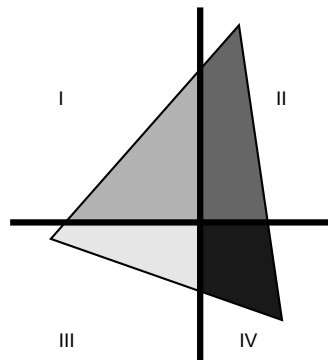


Abbildung 4.7: Liegt ein Dreieck im Blattschnitt vier Orthophotos, so müssen ihm mehrere Texturen zugeordnet werden. Dies ist mit *VRML* nicht möglich.

Bei näherer Betrachtung der von *ArcScene*¹⁰ generierten *VRML*-Ausgabe eines phototexturierten TINs ist Folgendes festzustellen: Das Rasterbild wurde gekachelt und liegt in mehreren Dateien abgespeichert vor. Dem TIN wurden entlang der Rasterbildkanten neue TIN-Kanten hinzugefügt. Somit wird das Problem, dass einer Fläche mehrere Rasterbilder zugeordnet werden müssen, umgangen. Es existieren nämlich Kanten, die denen des Orthophotos entsprechen. Somit gibt es nur noch Flächen, die von genau einem Rasterbild beschrieben werden. Es wird nun ein Verfahren vorgestellt, das diese Methode verfolgt.

Zum Einfügen der Orthophotokanten in das TIN ist eine Retriangulation mit den Orthophotokanten als Bruchkanten zu aufwendig, denn es kann ein einfacherer Weg gefunden werden. Es sind lediglich die Schnittpunkte der Projektion der Orthopho-

¹⁰Softwareprodukt der *ArcGIS Extensions* der Fa. *ESRI* (<http://www.esri.com>)

tokanten auf die Dreieckskanten zu berechnen und in das TIN einzufügen. In einem zweiten Schritt werden entsprechende Kanten eingefügt (vgl. Abbildung 4.8).

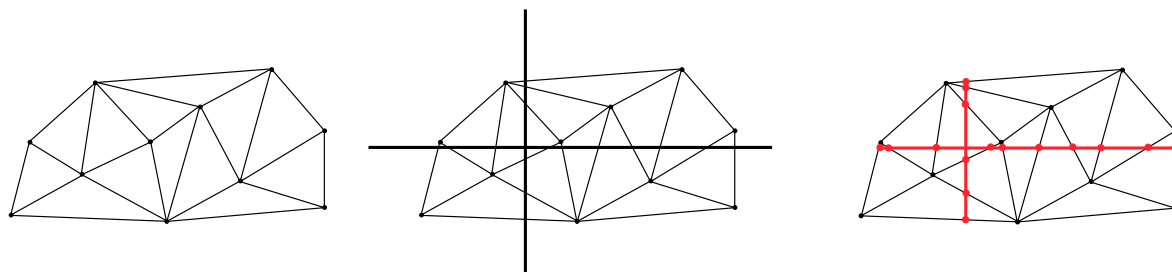


Abbildung 4.8: Für Dreiecke, die im Blattschnitt der Orthophotos liegen (links, Mitte) müssen neue Knoten und Kanten (rot) in das TIN eingefügt werden (rechts).

Wie in der Abbildung zu erkennen ist, handelt es sich nach dem Einfügen der Kanten nicht mehr um ein TIN, da Vierecke entstanden sind. Es stellt sich die Frage, ob eine Retriangulation notwendig ist. Zum einen kann angemerkt werden, dass durch das Hinzufügen weiterer Kanten eine Verdichtung des Netzes vorgenommen wird, die nicht zur Verschlechterung der Oberflächenbeschreibung führen kann. Eine Neugenerierung des Dreiecksnetzes unter Berücksichtigung des Delaunay-Kriteriums führt also vermutlich nicht zu einer besseren Oberflächenbeschreibung. Zum anderen wurden die Kanten ausschließlich zur Visualisierung des DGM eingefügt. Somit steht fest, dass eine Retriangulation nicht nötig ist, da in *VRML* beliebige planare Flächen texturiert werden können. Die Forderung der Planarität der Flächen ist erfüllt, da die eingefügten Kanten in der durch das jeweilige Dreieck beschriebenen Ebene liegen.

Bei der Betrachtung der implementierten Schnittstellen (vgl. Kapitel 5.5) wird deutlich, dass dieses Verfahren nur auf Daten angewendet wird, die zur Erstellung eines *VRML*-Modells aus der Datenbank abgefragt werden. In den Datenbestand der Datenbank werden durch dieses Verfahren keine neuen Knoten oder Kanten eingefügt!

Bislang wurde mit der Annahme argumentiert, dass ein Orthophoto einen Großteil des TINs überdeckt. Offen bleibt das Problem, welches bei Orthophotos in sehr großem Maßstab und DGM mit sehr großen Maschenweiten entsteht. Unter diesen Voraussetzungen kann es dazu führen, dass ein Dreieck größer ist als ein Orthophoto. Es sind zwei Fälle zu unterscheiden (vgl. Abbildung 4.9): Entweder ist das Dreieck vollständig durch ein Raster von Orthophotos überdeckt (links), oder es liegen nur vereinzelt Orthophotos vor (rechts).

Im ersten Fall (links) können solange mehrere Orthophotos zu einem Rasterbild zusammengefaßt werden (*merging*), bis diese Konstellation nicht mehr auftritt und das oben beschriebene Verfahren angewendet werden kann.

Im zweiten Fall (rechts) können die Schnittpunkte der Orthophotoecken mit dem entsprechenden Dreieck unter der o.g. Voraussetzung der 3D-Konsistenz berechnet werden. Es entstehen Rechtecke, die mit dem Orthophoto-Rasterbild texturiert werden können.

An dieser Stelle ist es unerheblich, dass die Rechtecke topologisch gesehen Löcher des Dreiecks darstellen, denn dies bereitet bei der Texturierung keine Probleme. Ande-

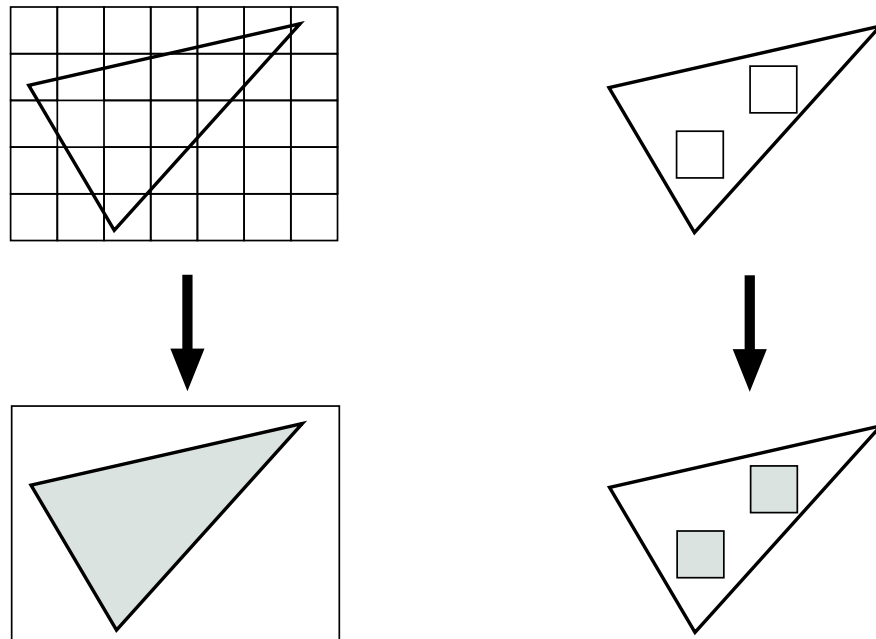


Abbildung 4.9: Ist das Orthophoto kleiner als das Dreieck, so sind diese beiden Konstellationen denkbar: Entweder ist das Dreieck komplett mit Orthophotokacheln bedeckt (links), oder es liegen nur einzelne Orthophots innerhalb der Dreiecksfläche vor (rechts).

rerseits sollte die Frage der Sinnhaftigkeit einer solchen Texturierung gestellt werden, der hier jedoch nicht nachgegangen wird.

Kapitel 5

Abbildung des Modells auf eine 3D-Geodatenbank

Zu Beginn dieser Arbeit wird die Notwendigkeit der gleichzeitigen Visualisierung von Geländeoberfläche und 3D-Stadtmodell unterstrichen. Die Darstellung des digitalen Geländemodells ist im wahrsten Sinne des Wortes die *Grundlage* einer real wirkenden Präsentation von 3D-Stadtmodellen. Bei der Entwicklung effizienter Systeme zur Verwaltung und Nutzung dieser Daten ist deren großes Datenvolumen zu beachten. Aus diesem Grund bieten sich Datenbankmanagementsysteme (DBMS) als Grundlage eines solchen Systems an. Datenbanken wurden speziell für effiziente Speicherung und Abfrage großer Datenmengen entwickelt. Wie in Kapitel 2 angeführt, verfügen einige DBMS über spezielle Datentypen für räumliche Daten, die auch eine gewisse GIS-Funktionalität unterstützen. Diese DBMS werden daher als *3D-Geodatenbanken* bezeichnet.

Zur Nutzung einer Geodatenbank wird aus dem bereits vorgestellten abstrakte Datenmodell zur Beschreibung von Geländeoberflächen ein Datenbankschema abgeleitet. Zudem werden Schnittstellen spezifiziert und implementiert, die zum Datenimport und -export genutzt werden.

Zum besseren Verständnis der verwendeten Geodatenbank wird einleitend ein allgemeiner Überblick über DBMS gegeben.

5.1 Datenbankmanagementsysteme

Der Aufgabe, große Datenmengen zu speichern und zu verwalten, werden Datenbankmanagementsysteme in hohem Maße gerecht. Umgangssprachlich werden sie oft als „Datenbanken“ bezeichnet, was ihrer umfangreichen Funktionalität allerdings nicht gerecht wird. Datenbank (DB) ist jener Teil eines DBMS, in dem die Daten gespeichert werden. Das DBMS stellt darüber hinaus vielfältige Werkzeuge und Methoden zur Verfügung, um Daten zu laden, zu verwalten, zu analysieren oder zu manipulieren, und um sie abzufragen. In der Literatur zum Thema Datenbanken werden die Begriffe *Datenbank* und *Datenbankmanagementsystem* ebenfalls nicht immer klar getrennt.

In diesem Zusammenhang sei darauf hingewiesen, dass oft mehrere Bezeichnungen für Standard-Datenbankelemente parallel verwendet werden. So wird beispielsweise eine Tabelle auch als *Datenbank*, *Entität*, *Relation*, *Datenbestand* oder *Datei* bezeichnet. Weitere wichtige Termini des Datenbankjargons, die sich weitestgehend selbst erklären und daher nicht näher erläutert werden, sind: *Zeile*, *Spalte*, *Feld* und *Wert*.

Grundlage dieser Arbeit bildet das objekt-relationale DBMS *ORACLE* in der Programmversion 9.2.0.2 mit der *Spatial* Erweiterung für räumliche Daten in der Version 9.2.0 (s. Kapitel 5.3.2). Aus diesem Grund wird hier verstärkt auf Syntax und Struktur von Oracle eingegangen.

Im Folgenden werden die am häufigsten verwendeten Typen von Datenbanksystemen vorgestellt, um einen Überblick über ihre Grundprinzipien zu geben.

5.1.1 Relationale Datenbanksysteme

Die Funktionsweise eines relationalen DBMS (RDMBS) basiert auf Tabellen, die durch so genannte *Schlüssel* miteinander verknüpft sind und typischerweise aus Spalten und Zeilen bestehen. Jede Zeile stellt einen Datensatz dar, dessen Attribute bzw. Elemente in den einzelnen Spalten geführt werden. Eine relationale Tabelle kann aus beliebig vielen Spalten bestehen. Bei der Spaltendeklaration muss der Datentyp der Spalte festgelegt werden. Hier können Elementartypen wie Zeichenketten oder Dezimalzahlen verwendet werden. Selbst bestehende Tabellen können in einigen DBMS um Spalten erweitert werden.

Der Umgang mit relationalen Tabellen soll an dem geometrischen Beispiel aus Abbildung 5.1 veranschaulicht werden.

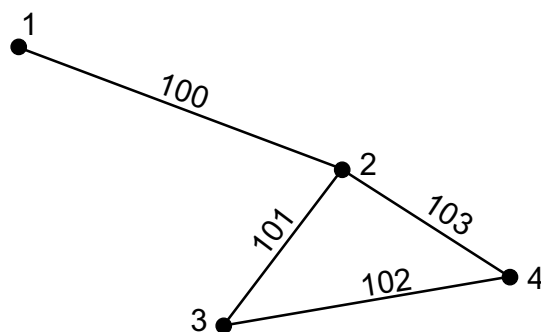


Abbildung 5.1: Beispiel einer Knoten-Kanten-Struktur.

Es wird die topologische Beziehung zwischen Knoten und Kanten beschrieben, nach der ein Knoten eine Koordinate und eine Kante einen Start- und einen Endknoten besitzt. Werden die Knoten und Kanten aus der Abbildung in relationalen Tabellen gespeichert, so ergibt sich das in den Tabellen 5.1 dargestellte Datenbankschema. Die Erweiterung `_reltab` im Tabellennamen weist darauf hin, dass es sich um eine relationale Tabelle handelt.

Damit die Zahlen der Spalte `Startknoten_Id` der Tabelle `Kanten_reltab` in Zusammenhang mit der Tabelle `Knoten_reltab` gebracht werden können, muss diese

Table Knoten_reltab

Knoten_ID	X_Wert	Y_Wert	Z_Wert
Number(10,0) PK	NUMBER(10,3)	NUMBER(10,3)	NUMBER(10,3)
1	0	4	2
2	4	2	7
3	3	0	5
4	8	1	3

Table Kanten_reltab

Kante_ID	Startknoten_Id	Endknoten_Id
Number(10,0) PK	NUMBER(10,0) FK	NUMBER(10,0) FK
100	1	2
101	2	3
102	3	4
103	2	4

Tabelle 5.1: Beispiel für relationale Tabellen anhand von Knoten (oben) und Kanten (unten). *PK*, *FK* deklarieren diese Spalte als Primär bzw. Fremdschlüsselspalte. *NUMBER(x,y)* definiert den Datentyp der Spalte. Hier handelt es sich um Dezimalzahlen mit x Gesamtstellen, die y Nachkommastellen beinhalten.

Verbindung explizit hergestellt werden. Dies geschieht über s.g. Schlüssel. Es werden *Primärschlüssel* (primary key, PK) und *Fremdschlüssel* (foreign key, FK) unterschieden, die einer Spalte oder einer Gruppe von Spalten explizit zugeordnet werden. Diese Zuordnung erfolgt bei der Tabellendeklaration. Ein Primärschlüssel ist eindeutig und identifiziert genau einen Datensatz der Tabelle. Im Beispiel sind die ID-Spalten der Tabellen als Primärschlüssel deklariert, da jede ID höchstens einmal pro Tabelle vorkommt. Für Spalten, deren Werte auf die Primärschlüssel anderer Tabellen verweisen, wird ein Fremdschlüssel deklariert. Werte in Fremdschlüsselspalten müssen zwingend auch in der zugewiesenen Primärschlüsselspalte existieren, da der Verweis ansonsten keinen Sinn machen würde.

Im Beispiel verweisen die Einträge der Start- und Endknoten-Spalte (Fremdschlüsselspalten) auf die jeweiligen Datensätze der Knotentabelle mit gleichem Wert in der Primärschlüsselspalte. Durch die PK-FK-Beziehung können jeder Kante die beiden entsprechenden Datensätze der Knotentabelle zugeordnet werden. Da ein Knoten Anfangs- oder Endknoten mehrerer Kanten sein kann, wird hier eine $1 : n$ Beziehung (ein Knoten – mehrere Kanten) beschrieben.

Um $n : m$ Beziehungen in relationalen Datenbanken darzustellen, muss eine weitere Tabelle angelegt werden, die diese Zuordnungen explizit vornimmt. Beispielsweise die Zuordnung von Eigentümern zu Grundstücken stellt eine $n : m$ Beziehung dar. Ein Grundstück kann mehreren Eigentümern gehören, jeder dieser Eigentümer kann aber auch mehr als ein Grundstück besitzen (vgl. Tabellen 5.2). In der Flurstückstabelle *Flurstück_reltab* wird jedem Flurstück ein in der Tabelle *Person* gespeicherter

Eigentümer zugewiesen. Dazu muss die PK-FK-Beziehung deklariert sein. Besitzt ein Eigentümer mehrere Grundstücke, so kann in der entsprechenden Spalte mehrfach auf die jeweilige Person verwiesen werden. Dies zeigt, dass Daten redundanzfrei gespeichert werden können. Eine Zuordnung von mehreren Eigentümern zu einem Grundstück ($n : m$) kann nicht realisiert werden. Abhilfe schafft die Tabelle `Eigentum_reltab`, in der unter fortlaufender Nummerierung jede Eigentümer-Flurstück-Beziehung gespeichert wird.

Person_reltab		Flurstück_reltab			Eigentum_reltab		
ID	Name	Nr	Eigentümer	Fläche	Nr	Eigentümer	Flurstück
PK		PK	FK		PK	FK	FK
1	Dagobert	12	1	12.033	1	1	12
2	Daisy	29	1	30.219	2	1	29
3	Donald	45	1	21.142	3	1	45
		99	2	450	5	2	99
					6	3	99

Tabelle 5.2: Beispiel zur Realisierung von $n : m$ Beziehungen zwischen Eigentümern (links) und Flurstücken (Mitte) über die zusätzliche Tabelle Eigentum (rechts).

5.1.2 Objektorientierte Datenbanksysteme

Wie in der Welt der objektorientierten Programmiersprachen wie *C++* oder *Java* können in entsprechenden Datenbankmanagementsystemen Objekte mit Attributen und Methoden definiert werden (ODBMS). Ziel dieser Abstraktion ist es, Objekte und ihre Attribute zu kapseln und über Methoden zugänglich zu machen. So werden klare Schnittstellen geschaffen, mit denen andere Entwickler die Objekte nutzen können, ohne deren interne Struktur zu kennen. Ein Realweltobjekt wird dabei auf ein abstraktes Objekt abgebildet, indem es in seine „atomaren“ Bestandteile zerlegt wird, deren Eigenschaften mit Attributen und Methoden beschrieben werden. Im Gegensatz zu relationalen DBMS, die lediglich Daten *speichern*, ist es aufgrund der Methoden in objektorientierten DBMS möglich, benutzerdefinierte *Berechnungen* auf den Daten auszuführen und dem Benutzer die Ergebnisse zu liefern. In objektorientierten DBMS existieren keine Tabellen.

5.1.3 Objekt-relationale Datenbanksysteme

Neben relationalen und objektorientierten DBMS existieren auch solche, die beide Ansätze kombiniert verfolgen. Wie oben bereits erwähnt ist das in dieser Arbeit verwendete *Oracle* ein solches Datenbankmanagementsystem. Im Folgenden werden die objekt-relationalen Möglichkeiten vom verwendeten DBMS *Oracle 9i* beschrieben. Es können sowohl klassische relationale Tabellen als auch Objekttypen mit entsprechenden Objekttabellen erzeugt werden. Eine Verknüpfung der Ansätze ist möglich,

indem (benutzerdefinierte) Objekttypen als Datentyp von Spalten relationaler Tabellen definiert werden (Objektspalten).

Table Kanten_objtab (of Kante_objtyp)

Kante_ID	Startknoten_REF	Endknoten_REF
Number	REFERENCE	REFERENCE
NUMBER(10,0)	Knoten_objtyp	Knoten_objtyp

Tabelle 5.3: Beispiel der Objekttable für Kanten, die auf der Definition des Objekttypen Kante beruht. Start- und Endknoten werden durch Zeiger (*Reference*) auf den Knoten-Objekttyp realisiert. So können Daten redundanzfrei gespeichert werden.

Im Beispiel des vorherigen Abschnitts würde anstatt von relationalen Tabellen für Knoten und Kanten Objekte definiert. Denkbar wären ein Objekt **Knoten** mit den Attributen *Id*, *x*, *y* und *z* und ein Objekt **Kante** mit den Attributen *Id*, *Anfangsknoten* und *Endknoten* (vgl. Tabelle 5.3). Der Datentyp **Reference** wird unten näher erläutert. Er ist das objektorientierte Gegenstück zu einer PK-FK-Beziehung und realisiert einen Zeiger auf ein Objekt.

Zur Ergänzung des Objekts können Methoden zur Bestimmung des Abstandes zweier Knoten oder zur Umkehrung der Richtung einer Kante implementiert werden.

Objekttypen

In Oracle werden abstrakte Objekte als *Objekttypen* (vgl. [Ora02a]) bezeichnet und im Folgenden durch die Erweiterung `_objtyp` gekennzeichnet. Solche Objekttypen können zur Definition von Objekttabellen (`_objtab`), deren Spalten genau den Attributen des Objekttypen entsprechen, verwendet werden. Im Gegensatz zu relationalen Tabellen können bestehenden Objekttabellen keine weiteren Spalten hinzugefügt werden. Dazu müsste der Objekttyp geändert und ihm ein neues Attribut zugewiesen werden. Das ist nicht möglich, sobald eine von diesem Typ abhängige Objekttable existiert.

Referenzen

Wie in Tabelle 5.3 zu erkennen, werden in Oracle Beziehungen zu anderen Objekttypen bzw. zu Instanzen dieser Typen durch Objekte vom Typ **REFERENCE** (wie z.B. das Attribut `Startknoten_REF` des Objekttypen **Kante**) realisiert. Diese Zeiger verweisen auf existierende Instanzen des entsprechenden Objekttyps.

Ein Beispiel für das Erzeugen einer Referenz findet sich in Listing 5.3. Durch den Befehl **DEREF** kann mit der „“-Schreibweise auf Attribute des referenzierten Objekts zugegriffen werden. In der unten beschriebenen Datenbanksprache *SQL* müsste der folgende Befehl integriert werden, um aus der Kanten-Table auf die ID des Startknotens einer Kante zuzugreifen: `DEREF(Startknoten_REF).Knoten_Id`. Dieser Ausdruck ist zur Anwendung in einen *SQL*-Befehl zu integrieren.

Hinweis: An dieser Stelle sei auf ein bei den Forschungen zu dieser Arbeit aufgetretenes Problem hingewiesen! Mit der verwendeten Oracle-Version ist es nicht möglich,

ein Objekt vom Typ `REFERENCE` auf eine Instanz eines Objekttypen `typ1_objtyp` zu erzeugen, wenn diese Instanz in einem Feld einer relationalen Tabelle gespeichert ist, dessen Tabellenspalte vom Typ `typ1_objtyp` ist. Referenzen können also nicht auf Felder relationaler Tabellen verweisen.

Collections

Eine weitere wichtige Gruppe der Oracle-Datentypen stellen die *Collections* dar. Sie können mehrere Elemente eines Datentyps aufnehmen und ermöglichen so eine explizite 1:n Beziehung. Collections können in Oracle wie alle Objekttypen sowohl als Objektattribute als auch als Spaltentypen verwendet werden. Sie werden in `VARRAY` und `NESTED TABLE` unterteilt. Beide Typen müssen explizit deklariert werden (`_vartyp`, `_ntabtyp`), wobei ihnen ein Datentyp zugewiesen wird, dem alle enthaltenen Elemente entsprechen.

Dem `VARRAY` wird bei der Deklaration eine maximale Länge vorgegeben. Er beinhaltet eine geordnete und festgelegte Menge von Elementen, die durch einen Index eindeutig zu identifizieren sind. Er ist mit einem *Array* objektorientierter Programmiersprachen zu vergleichen.

Im Gegensatz dazu speichert eine `NESTED TABLE` eine ungeordnete Menge von Elementen. Mit Hilfe einer Nested Table können n:m-Beziehungen realisiert werden, denn bildlich gesprochen handelt es sich hier um Tabellen in Tabellen. Ein Feld eines Datensatzes kann somit eine Tabelle mit beliebiger Anzahl von Datensätzen beinhalten. Durch verschachtelte (und kombinierte) Deklaration können Multilevel-Collections erzeugt werden. Beispielsweise kann ein `VARRAY` wiederum aus `VARRAYs` bestehen.

5.2 Datenbankkommunikation

5.2.1 SQL

Die *Structured Query Language (SQL)* ist eines der wichtigsten Hilfsmittel beim Umgang mit DBMS. Diese Programmiersprache kann in die zwei Bereiche **Datendefinition** (*DDL - Data Definition Language*) und **Datenabfrage** (*DML - Data Manipulation Language*) unterteilt werden. Von besonderer Bedeutung ist die problemlose Integration von *SQL*-Befehlen in *Java*. Werden *SQL*-Statements typischerweise mit einer Art Kommandozeilen-Konsole (z.B. das zur Oracle-Installation gehörige *SQLPlus Worksheet*) ausgeführt, bietet die in Kapitel 5.2.3 beschriebene Schnittstelle zu *Java* die elegantere Möglichkeit zur Verarbeitung großer Datenmengen innerhalb eines Programmsystems mit Datenbankanschluss.

Im Folgenden soll ein kleiner Einblick in die *SQL*-Syntax geben werden, die weitestgehend selbsterklärend ist. Dabei wird auf die *SQL*-Spezifikation von Oracle [Ora02d] zurückgegriffen. Zur Steigerung der Lesbarkeit des Codes werden alle *SQL*-Schlüsselwörter in Großbuchstaben und alle benutzerdefinierten Namen in Kleinbuchstaben geschrieben. Intern wandelt Oracle jedoch alle Namen in Großbuchstaben um. Durch

```

CREATE TABLE knoten_reltab (
  knoten_id NUMBER(10,0),
  x_wert    NUMBER(10,3),
  y_wert    NUMBER(10,3),
  z_wert    NUMBER(10,3),
  PRIMARY KEY(knoten_id)
)
/

```

Listing 5.1: SQL-Befehl zum Erstellen einer relationalen Knotentabelle.

```

CREATE OR REPLACE TYPE kante_objtyp
AS OBJECT (
  kante_id          NUMBER(10,0),
  startknoten_ref  REF knoten_objtyp,
  endknoten_ref    REF knoten_objtyp
)
/
CREATE TABLE kanten_objtab
OF kante_objtyp (
  CONSTRAINT kanten_idx
  PRIMARY KEY(kante_ID),
  FOREIGN KEY (startknoten_ref)
  REFERENCES knoten_objtab,
  FOREIGN KEY (endknoten_ref)
  REFERENCES knoten_objtab
)
OBJECT IDENTIFIER IS PRIMARY KEY
/

```

Listing 5.2: SQL-Befehl zum Erstellen des Kanten-Objekttyps (erster Befehl) und der entsprechenden Objekttable (zweiter Befehl).

konsequente Zeilenumbrüche und Einrückung wird die Lesbarkeit ebenfalls gesteigert. Ein Befehl wird durch einen Schrägstrich (/) oder ein Semikolon (;) abgeschlossen.

Das Grundgerüst eines *DDL*-Befehls wird in Listing 5.1 am Beispiel der in Kapitel 5.1.1 verwendeten relationalen Tabelle für Knoten vorgestellt. Durch diesen Befehl wird die Tabelle, die Datentypen und Namen der Spalten und die Primärschlüsselspalte definiert.

Listing 5.2 zeigt am Beispiel des in Kapitel 5.1.3 verwendeten Objekttyps für Kanten die Definition des Objekttypen (erster Befehl) und der entsprechenden Objekttable (zweiter Befehl). Es wird in der Definition der Objekttable explizit der Name des Primärschlüssels (*kanten_idx*) angegeben. Die Zeiger auf die Knotentabelle müssen beim Erstellen des Objekttypen als Fremdschlüssel gekennzeichnet und ihre „Zielta-
belle“ angegeben werden.

Beispiele für *DML*-Befehle zeigt Listing 5.3. Der erste Befehl fragt die *x*-Koordinate

```

SELECT x_wert
FROM knoten_reltab
WHERE knoten_id = 2
/

INSERT INTO kanten_objtab
  SELECT
    100,
    REF(knoten1),
    REF(knoten2)
  FROM knoten_objtab knoten1, knoten_objtab knoten2
  WHERE knoten1.knoten_id = 1 AND knoten2.knoten_id = 2
/

```

Listing 5.3: SQL-Befehle zum Abfragen (erster Befehl) oder Füllen (zweiter Befehl) von Tabellen.

des Punktes 2 aus der relationalen Knotentabelle ab. Mit dem zweiten Befehl wird die Kante mit der ID 100 in `kanten_objtab` eingefügt. Hier wird deutlich, dass ein `REFERENCE` ein Zeiger auf eine existierende Instanz eines Objekttypen ist, die mit einer Unterabfrage selektiert wird.

5.2.2 PL/SQL

Bei komplexen Datenbankabfragen stößt der Benutzer schnell an die Grenzen von *SQL*, da es keine prozeduralen Fähigkeiten besitzt. Alternativ bzw. ergänzend steht *PL/SQL* (Procedural Language / Structured Query Language) als *SQL*-Erweiterung zur Verfügung, die diesen Anforderungen durch Schleifenstrukturen (*if*, *while*, *for*), Modularisierbarkeit und lokale, zur Laufzeit veränderbare Variablen gerecht wird.

Ein *PL/SQL* Skript besteht aus mindestens einem Block der folgenden Struktur:

- Deklaration (*optional*, eingeleitet durch `DECLARE`) - Hier werden Konstanten und Variablen deklariert.
- ausführbarer Abschnitt (gekennzeichnet durch `BEGIN` und `END`) - Hier können sowohl *SQL* Anweisungen als auch *PL/SQL* Anweisungen ausgeführt werden.
- Fehlerbehandlung (*optional*, Schlüsselwort `EXCEPTION`) - Hier werden Fehler, die bei der Ausführung des Skripts auftreten können, abgefangen.

Soll ein Skript nur einmal ausgeführt werden, so genügt es, dieses als transiente *anonyme Prozedur* auszuführen. Darüber hinaus besteht die Möglichkeit, *Prozeduren* und *Funktionen* zu erzeugen und mit einem einleitenden `CREATE PROCEDURE` bzw. `CREATE FUNCTION` persistent in der Datenbank zu speichern. Diese Konstrukte unterscheiden sich von anonymen Prozeduren dadurch, dass sie einen eindeutigen Namen haben und ihnen Parameter übergeben werden können. Eine Funktion liefert im

Gegensatz zur Prozedur einen Rückgabewert eines bestimmten Datentyps. Gruppen verwandter Prozeduren und Funktionen können in *Paketen* zusammengefasst werden. Wie oben erwähnt, können innerhalb des ausführbaren Abschnitts gewöhnliche *SQL* Befehle genutzt werden. Das führt zu einem Problem, da *PL/SQL* Strukturen nur in der Lage sind, einzelne Werte zu verarbeiten, nicht jedoch ein Abfrageergebnis mit mehreren Zeilen. Zur Lösung bietet *PL/SQL* den Datentyp *CURSOR* an, der beliebig viele Zeilen aufnehmen kann. Er kann in einer *for*-Schleife sequenziell durchlaufen werden, um die einzelnen Ergebniszeilen zu verarbeiten.

In Anhang C findet sich ein Beispiel eines *PL/SQL* Skripts. Es dient dazu, die Referenzobjekte zu definieren, die Orthophotos und Dreiecken miteinander referenzieren (vgl. Kapitel 5.4). Die ausführliche Spezifikation von *PL/SQL* findet sich in [Ora01b]. In [LN02] wird eine grundlegende Einführung gegeben.

5.2.3 JDBC

Mit Hilfe des JDBC-Treibers (Java Database Connectivity) lassen sich aus einem *Java*-Programm heraus Verbindungen zu Datenbanken aufbauen, *SQL*-Befehle ausführen und Abfrageergebnisse bearbeiten. Dem Nutzer steht hier das in *Java 1.4* implementierte Package *java.sql* oder das von Oracle angebotene Package *ojdbc14* [Ora02c] und [Ora02b] zur Verfügung. OJDBC bietet die gesamte JDBC Grundfunktionalität, ist aber um Klassen zur Verwendung Oracle eigener Objekttypen erweitert. Anhand von Listing 5.4 sollen die benötigten Methoden und Klassen kurz vorgestellt werden. Wegen der Kommentare im Code (eingeleitet durch Doppel-Slash (//)) wird dieser nicht weiter erläutert.

Besonders wichtig scheint es, den Unterschied der beiden Möglichkeiten zum Ausführen eines *SQL*-Befehls herauszustellen. Es können die Klassen **Statement** (Zeilen 30–37) und **PreparedStatement** (Zeilen 18–29) genutzt werden. Beide müssen für eine bestehende Verbindung explizit instanziiert werden (Zeilen 16 und 19). Hier liegt der große Unterschied der zwei Klassen:

Ein **Statement** kann geschaffen werden, ohne ihm einen bestimmten *SQL*-Ausdruck zuzuordnen. Somit kann ihm stets ein beliebiger *SQL*-Befehl übergeben werden, der zur Datenbank geschickt und dort ausgeführt wird. Im Gegensatz dazu wird ein **PreparedStatement** für eine bestimmte *SQL*-Anweisung instanziiert und der Datenbank übermittelt. Variablen dieser *SQL*-Anweisung werden dabei durch Fragezeichen (?) ersetzt, denen bei bestehender DB-Verbindung jederzeit neue Werte zugewiesen werden können. Das **PreparedStatement** wird mit diesen Werten auf der DB ausgeführt (Zeilen 23–27). Müssen viele Datensätze in die gleiche Tabelle geladen werden, so hat diese Möglichkeit den Vorteil, dass der *SQL*-Ausdruck, der als String übergeben wird, nicht für jeden Befehl neu generiert und zur Datenbank gesendet werden muss. Es genügt, die jeweils aktuellen Variablenwerte zu übergeben. Zur weiteren Performancesteigerung können alle Parameter-Updates in einem Batchstapel gespeichert werden, der in einem Stück ausgeführt wird.

Anhand von Listing 5.4 wird die Funktion des **PreparedStatement** deutlich. Es werden 99 Datensätze über ein **PreparedStatement** in die Knotentabelle 5.1 geladen (Zeilen

```

1 import java.sql.*;
2
3 public static void main(String [] args) {
4     Connection con = null;
5     Statement stmt = null;
6     PreparedStatement pstmt = null;
7     ResultSet rs =null;
8
9     try { //Treiber laden
10        DriverManager.registerDriver (new oracle.jdbc.OracleDriver ());
11    } catch (SQLException e1) {System.out.println(e1);}
12    try { //Verbindung aufbauen
13        con = DriverManager.getConnection(
14            "jdbc:oracle:thin:" +
15            "benutzer/passwort@127.0.0.1:1521:datenbankname" );
16        stmt = con.createStatement(); //Statement initialisieren
17    } catch (SQLException e2) {System.out.println(e2);}
18    try { //PreparedStatement erzeugen
19        pstmt = con.prepareStatement(
20            "INSERT INTO knoten_reltab " +
21            "VALUES(?,?,?,?)");
22        for(int i = 1; i < 100; i++){ //Paramerter updaten
23            pstmt.setLong(1, i);
24            pstmt.setDouble(2, Math.random());
25            pstmt.setDouble(3, Math.random());
26            pstmt.setDouble(4, Math.random());
27            pstmt.executeUpdate(); //PreparedStatement ausführen
28        }
29    } catch (SQLException e3) {System.out.println(e3);}
30    try { //Daten abfragen
31        rs = stmt.executeQuery(
32            "SELECT knoten_id " +
33            "FROM knoten_reltab;");
34        while(rs.next()){ //Daten ausgeben
35            System.out.println(rs.getLong(1));
36        }
37    } catch (SQLException e4) {System.out.println(e4);}
38    try { //Verbindung trennen
39        stmt.close();
40        pstmt.close();
41        con.close();
42    } catch (SQLException e5) {System.out.println(e5);}
43 }

```

Listing 5.4: Javacode zur Datenbankkommunikation mit dem JDBC Treiber.

18-29). Dabei wird die ID bei eins beginnend hochgezählt und die Koordinatenwerte werden als Zufallszahlen erzeugt. Anschließend werden die Knoten-IDs mit einem `Statement` abgefragt und ausgegeben (Zeilen 30-37).

5.3 Spezielle Objekttypen

5.3.1 ORDImage

Oracle *interMedia* [Ora01a] bietet die Möglichkeit, multimediale Daten (digitale Bilder, Audio und Video) in einer Datenbank zu speichern. Im Rahmen dieser Arbeit wird es zur Speicherung der Orthophotos in der Datenbank verwendet. Dies geschieht mit Hilfe des Objekttypen `ORDImage`, der Methoden zur Dateiformatkonvertierung, Bildformatkonvertierung und Bildkompression zur Verfügung stellt. In dieser für digitale Rastergrafiken optimierten Funktionalität liegen die Vorteile gegenüber der Speicherung der Orthophotos als Objekte vom Typ `BLOB` (binary large object) oder `BFILE` (binary file).

Listing 5.5 zeigt, wie das Bild `c:/test/testbild.jpg` in die Tabelle `photos_reltab` einer Datenbank geladen werden kann. Zunächst muss ein leeres Objekt als Platzhalter eingefügt werden (Zeilen 7-11). In Zeile 13-17 werden lokale Variablen deklariert, wobei auf die Großschreibung zu achten ist! Das eigentliche „Laden“ erfolgt in Zeile 28, wo dem selektierten Dummy die Bilddatei übergeben wird. Abschließend muss der Vorgang mit `UPDATE` und `COMMIT` in die Datenbank übernommen werden.

Es ist zu beachten, dass sich alle angegebenen Verzeichnisse und Dateien auf den Rechner, auf dem das DBMS installiert ist, beziehen. Zu Bildern, die auf einem Clientrechner gespeichert sind, kann demnach kein Pfad angegeben werden. Unter Windows erzeugte Netzlaufwerksverbindungen können auch nicht verwendet werden, weil diese bei einem Neustart des DB-Rechners erst dann wieder hergestellt werden, wenn der Oracle Dienst vom Betriebssystem bereits gestartet ist. Die Netzlaufwerke existieren beim Start von Oracle demnach noch nicht. Der Versuch, anstatt eines lokalen UNC¹ Pfads eine URL² bei der Variablendeklaration von `MYDIR` (Zeile 16, Listing 5.5) anzugeben schlug fehl.

Zur Lösung des Problems, Bilder von einem Client- auf den DB-Rechner zu laden und in umgekehrter Richtung wieder abzufragen, wurde das auf der Oracle Homepage³ zur Verfügung gestellte Java-Package `ordim817` verwendet, das Klassendefinitionen für `ORDImage`-Objekte anbietet. Diese Klassen haben Methoden, mit denen selektierte Objekte vom Typ `ORDImage` als Rasterbilder auf dem Clientrechner gespeichert oder lokal vorliegende Bilder als `ORDImage` Objekte zur Datenbank hinzugefügt werden können. Diese werden im Kapitel 5.5 im Zusammenhang mit den implementierten Schnittstellen erläutert.

¹Universal Naming Convention

²Uniform Resource Locator

³<http://otn.oracle.com> – Das Package wird als „Java Classes for Oracle8i on NT“ bezeichnet!

```

1  CREATE TABLE photos_reltab(
2     photo_id  NUMBER PRIMARY KEY,
3     photo     ordsys.ordimage
4  )
5  /
6
7  INSERT INTO photos_reltab VALUES(
8     1,
9     ordsys.ordimage.init()
10 )
11 /
12
13 CREATE OR REPLACE DIRECTORY MYDIR AS 'c:/test';
14
15 DECLARE
16     MYIMAGE ordsys.ordimage;
17     ctx RAW(4000) := NULL;
18 BEGIN
19     INSERT INTO photos_reltab VALUES(
20         1,
21         ordsys.ordimage.init()
22     );
23
24     SELECT photo INTO MYIMAGE FROM photos_reltab
25     WHERE photo_id = 1
26     FOR UPDATE;
27
28     MYIMAGE.importFrom(ctx, 'file', 'MYDIR', 'TESTBILD.JPG');
29
30     UPDATE photos_reltab
31     SET photo = MYIMAGE
32     WHERE photo_id = 1;
33
34     COMMIT;
35 END;
36 /

```

Listing 5.5: PL/SQL Skript zum Laden eines Bildes in die Datenbank.

5.3.2 Oracle Spatial

Die Spatial-Erweiterung [Ora02e] des DBMS Oracle (im Folgenden kurz als „Spatial“ bezeichnet) dient der effizienten Verwaltung räumlicher Daten und soll hier näher beschrieben werden.

Räumlichen Daten sind durch ihre Koordinatengrundlage in einem wohldefinierten Bezugssystem gekennzeichnet. Sie lassen sich in die in Abbildung 5.2 gezeigten Typen einteilen, die der *Simple Features Specification* des *OpenGIS Consortium* (OGC) [Ope99] entsprechen. Die Verwendung von Typen dieser Spezifikation bedeutet eine enorme Steigerung der Interoperabilität des DBMS, da die OGC-Spezifikationen in der von proprietären Formaten geprägten GIS-Welt immer mehr an Bedeutung gewinnen.

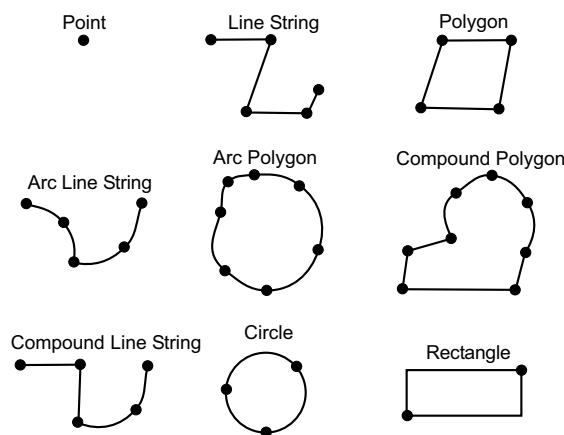


Abbildung 5.2: Geometrietypen in Oracle Spatial (aus [Ora02e]).

Es sind lediglich Punkt-, Linien und Flächengeometrien spezifiziert. Aus diesem Grunde fehlen in Oracle Datentypen für raumbezogene Volumengeometrien. Dieses Problem wurde zu Beginn der Arbeit diskutiert und wird später nochmals aufgegriffen.

Der Objekttyp MDSYS.SDO_GEOMETRY

Jede Flächengeometrie lässt sich als Objekt vom Typ `MDSYS.SDO_GEOMETRY` (oder kurz Spatial-Objekt) beschreiben. Die komplexe Syntax zur Beschreibung eines solchen Objekts sei hier anhand von Listing 5.6 erläutert.

Das Objekt hat fünf Attribute, die kurz skizziert werden. Eine ausführliche Beschreibung bietet [Ora02e].

1. `SDO_GTYPE` (Zeile 2) – Typ der Geometrie: Eine vierstellige Zahl **abcd**:
 - **a**: Dimension des Objektes (hier 3D, da die Eckpunkte Höhenwerte haben).
 - **b**: ID der Referenzdimension des linearen Referenzsystems (LRS) (0 bezeichnet die letzte Dimension des LRS, hier die z-Achse).

```

1  MDSYS.SDO_GEOMETRY(
2  3003,
3  NULL,
4  NULL,
5  MDSYS.SDO_ELEM_INFO_ARRAY(1, 1003, 1),
6  MDSYS.SDO_ORDINATE_ARRAY(
7    4, 2, 7,
8    3, 0, 5,
9    8, 1, 3,
10   4, 2, 7
11 )

```

Listing 5.6: Beschreibung einer Masche eines TINs als Objekt vom Typ MDSYS.SDO_GEOMETRY.

- **cd:** zweistellige ID des Geometrietyps (z.B 01 für POINT oder wie hier 03 für POLYGON).
2. **SDO_SRID** (Zeile 3) – ID des Referenz-Koordinatensystems. Die ID muss einem Wert der Tabelle MDSYS.CS_SRS entsprechen und in USER_SDO_GEOM_METADATA eingefügt sein. Hier können beliebige räumliche Bezugssysteme definiert werden. Es stehen sämtliche geodätische Daten, Referenzellipsoide und Projektionen zur Verfügung. Alle Größen können auch frei definiert werden. NULL verwendet ein rechtwinkeliges, kartesisches Koordinatensystem.
 3. **SDO_POINT** (Zeile 4) – Punktobjekt zur Speicherung von Punkten. Dies könnte auch über den Geometrietyp 2001 oder 3001 erfolgen. Die explizite Verwendung des Punktobjektes ist aber effizienter. Hier ist der NULL-Wert gesetzt, da ein Polygon beschrieben wird.
 4. **SDO_ELEM_INFO** (Zeile 5) – Info über SDO_ORDINATES mit beliebig vielen Trippeln von Zahlen (**o**, **p**, **q**), die jeweils eine Teilgeometrie näher spezifizieren:
 - **o:** SDO_STARTING_OFFSET Index des MDSYS.SDO_ORDINATE_ARRAY (beginnend bei 1). Ab diesem Index beginnt eine neue Teilgeometrie des Gesamtobjekts (z.B. ein Loch in einer Masche).
 - **p:** SDO_ETYPE Typ der Teilgeometrie (z.B. 1003 für Polygon).
 - **q:** SDO_INTERPRETATION:
 - Beschreibt **p** eine Assoziation von Geometrien, dann wird hier die Anzahl der Unterelemente dieser Assoziation angegeben.
 - Beschreibt **p** keine Assoziation von Geometrien, dann wird hier die Art der Linienverbindungen zwischen den Eckpunkten angegeben (z.B. 2 für Kreisbögen oder hier 1 für gerade Linien).
 5. **SDO_ORDINATES** (Zeile 6-10) – Koordinatenwerte durch Kommata getrennt. Der Zeilenumbruch zwischen den Punkten, wie im Beispiel, ist nicht notwendig und

dient nur der Lesbarkeit. Der letzte Punkt einer Geometrie muss dem ersten Punkt entsprechen.

Anhand dieser Struktur wird deutlich, dass Spatial-Objekte keinerlei topologische Zusammenhänge von Knoten, Kanten und Flächen wiedergeben, sondern lediglich Geometrien beschreiben. Beispielsweise muss die gemeinsame Kante zweier benachbarter Dreiecke bzw. deren zwei Knoten zur Beschreibung als Spatial-Objekt in beiden Geometrien explizit angegeben werden. Diese Informationen liegen also redundant vor. Trotz des Auftretens der zwei Punkte in beiden Dreiecken sind keinerlei Nachbarschaftsbeziehungen der Objekte gespeichert oder ableitbar.

Spatial vs. 3D

Oracle Spatial unterstützt in der verwendeten Version dreidimensionale Daten nur äußerst spärlich. Alle Geometrien können mit 3D-Koordinaten (x,y,z) erzeugt werden. Die Nutzung von Operatoren und Funktionen, die in den folgenden Kapiteln beschrieben werden, ist jedoch stark eingeschränkt. Sie können entweder nur auf 2D-Daten angewendet werden, oder sie lassen die dritte Dimension völlig außer Acht und arbeiten mit einer Projektion der Daten in die (x,y) Ebene.

Wie bereits erläutert sind die Geometrietypen auf Punkt-, Linien- oder Flächengeometrien beschränkt. Dieser Umstand führt dazu, dass an entscheidenden Stellen große Anstrengungen und Tricks nötig sind, um eine 3D-Funktionalität zu erreichen. Beispielsweise können Flächengeometrien mit Punkten des dreidimensionalen Raums erzeugt werden, so dass ein Polygon mit 3D-Punkten nicht zwingend planar ist. Aufgrund dieser Tatsache kann beispielsweise ein Quader als ein 3D-Polygon beschrieben werden. (vgl. Abbildung 5.3).

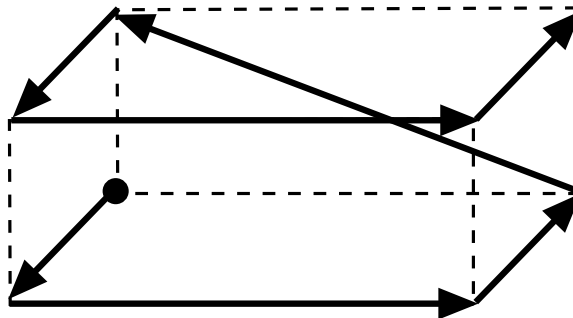


Abbildung 5.3: Beschreibung eines Quaders durch ein 3D-Polygon.

Indizierung räumlicher Daten

Die Verwendung von Spatialobjekten hat den Vorteil, dass ein höchst effizienter räumlicher Index dieser Daten erzeugt werden kann, der kürzere Anfragezeiten ermöglicht. Dieser Index wird in einer Baumstruktur als **Quadtree** oder als **R-Baum** generiert.

Das Generieren eines Quadtree erfolgt durch (rekursive) Unterteilung der Gesamtfläche aller Geometrien in Kacheln (vgl. Abbildung 5.4). Diese Zerlegung wird als

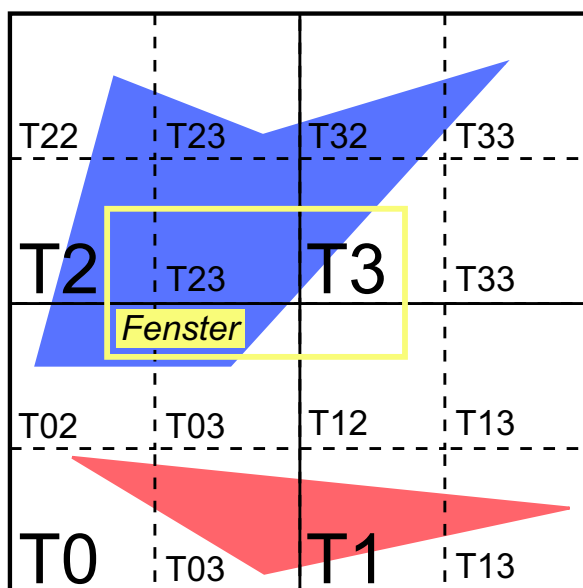


Abbildung 5.4: Zum Generieren des Quadtree wird die Fläche (rekursiv) im Sinne einer Tesselation unterteilt.

Tesselation bezeichnet. Die Idee des Quadtree ist eine symmetrische Dekomposition einer Fläche in vier Teile – daher auch **Quadtree**. Von diesem Prinzip weicht Oracle Spatial ab, da die Kachelgröße oder die Gesamtzahl von Kacheln vom Benutzer defubiert wird. Es ergibt sich demnach nicht immer das in der Abbildung dargestellte quadratisch-symmetrische Erscheinungsbild.

Zur Selektion aller Geometrien in einem bestimmten Suchfenster müssen lediglich die Kacheln, die vom Suchfenster geschnitten werden, auf Berührungspunkte mit den Geometrien überprüft werden. Und eben diese Überprüfung findet bereits beim Generieren des Indexes statt, so dass zum Zeitpunkt der Selektion bereits bekannt ist, welche Geometrie mit welcher Kachel interagiert. Bei näherer Betrachtung des obigen Beispiels wird deutlich, dass die Wahl der Kachelgröße bzw. -anzahl entscheidenden Einfluss auf die Exaktheit des Abfrageergebnisses hat. Sollen alle Geometrien im Abfragefenster selektiert werden, so wird in der ersten Kachelungsstufe (Kacheln T_x) auch das Dreieck ausgewählt, weil die Kacheln T_0 und T_1 sowohl das Abfragefenster als auch das Dreieck berühren oder schneiden. In der zweiten Kachelungsstufe (Kacheln T_{xy}) wird das Dreieck nicht mehr selektiert, weil keine räumliche Schnitt-Beziehung zwischen Kacheln und Abfragefenster herrscht.

Der R-Baum (vgl. [Gut84]) beruht auf stufenweiser Generierung der Minimum Bounding Rectangles (MBR) (vgl. *BoundingBox* – Kapitel 4.2.2). Wie in Abbildung 5.5 dargestellt, werden die MBRs „vom Großen ins Kleine“ für alle Geometrien erzeugt und der R-Baum somit schrittweise angelegt. Suchbaumelemente werden üblicherweise in Wurzel, Knoten und Blätter unterteilt. Im hier gezeigten Beispiel ist *root* das Wurzelement, *A* und *B* sind Knotenelemente und *a*, *b*, *c*, *d* die Blätter. Gerade in den

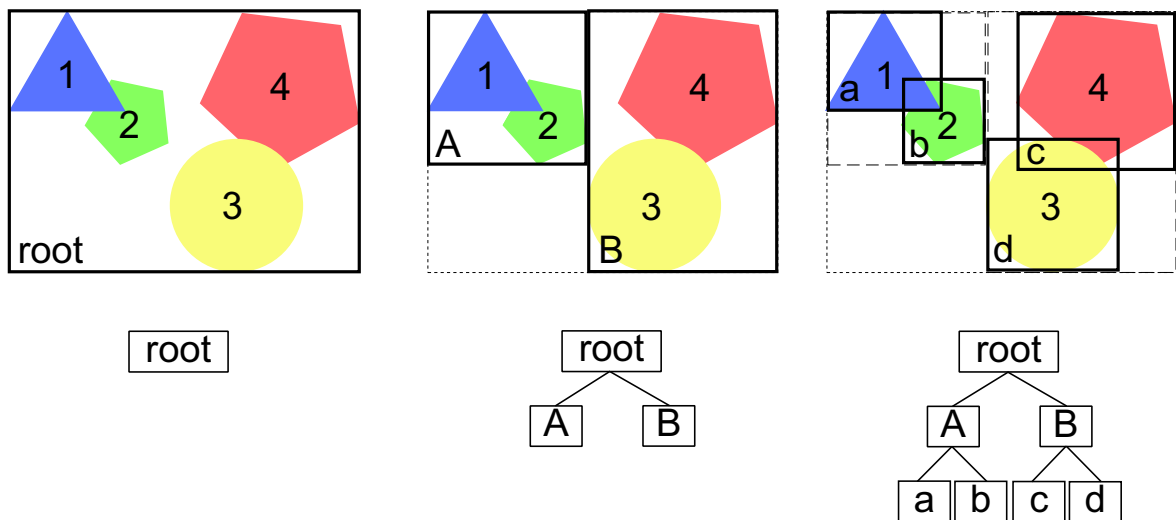


Abbildung 5.5: Der R-Baum wird schrittweise über die Minimum Bounding Rectangles erzeugt.

Blättern werden Verweise auf die enthaltenen Geometrien gespeichert.

In Oracle wird der Index mit einem `CREATE INDEX`-Befehl für jede Tabelle mit Spatialobjekten explizit generiert. Werden bei bestehendem Index weitere Geometrien in die Tabelle geschrieben, so kann der Index mit einem `ALTER INDEX`-Befehl neu erzeugt werden. Diese Rechenoperation ist für einen R-Baum aufwendiger als für einen Quadtree (s.u.). Damit ein Spatial-Index einer Tabelle erzeugt werden kann, muss für diese Tabelle die in Kapitel 5.3.2 erwähnte Metadatendefinition im View `USER_SDO_GEOM_METADATA` vorliegen.

Da der Quadtree-Index nur auf zweidimensionale Daten angewendet werden kann, wird für das unten vorgestellte Datenbankschema ein dreidimensionaler R-Baum-Index erzeugt. Dieser beschreibt die Geometrien nicht mit Rechtecken, sondern mit Quadern. Wie in Kapitel 2.3 angeführt genügt in den meisten Fällen zwar ein 2D-Index [ASB03], mit einem dreidimensional erzeugten Index stehen aber alle Abfragemöglichkeiten offen. Der 3D-Index hat zwar den Nachteil, dass er bei großen Update-Tätigkeiten rechenintensiv erneuert werden muss, es steht aber keine Alternative zur Verfügung. Das unvorteilhafte Update-Verhalten liegt darin begründet, dass der R-Baum bei jedem Update umsortiert werden muss, bis er ausgeglichen⁴ ist.

Hinweis: Auch in diesem Zusammenhang wurden im Laufe dieser Arbeit Unzulänglichkeiten der verwendeten Oracle-Spatial Version deutlich. Es ist nicht möglich, einen räumlichen Index für eine Objekttablelle zu definieren. Das bedeutet, dass ein benutzerdefiniertes Objekt mit einem Attribut vom Typ `MDSYS.SDO_GEOMETRY` zwar definiert, eine aus ihm abgeleitete Objekttablelle aber nicht indiziert werden kann. Das Hinzufügen eines Spatialobjekts zur Tabelle ist durchaus möglich, macht aber wegen des fehlenden Indexes wenig Sinn. Demnach müssen zur Verwaltung räumlicher Daten durch Objekte vom Typ `MDSYS.SDO_GEOMETRY` stets relationale Tabellen verwendet

⁴Ein Suchbaum ist ausgeglichen, wenn alle Blätter das gleiche Niveau haben.

werden.

Operatoren und Funktionen

Spatial bietet zum Umgang mit räumlichen Daten *Spatial-Operatoren* und *Geometrie-Funktionen*. Operatoren prüfen, ob zwei oder mehrere Objekte in einer bestimmte Beziehung zueinander stehen. Funktionen hingegen führen räumliche Berechnungen auf Objekten durch.

So prüft beispielsweise der Operator `SDO_RELATE`, ob zwei Geometrien in einer vorgegebenen topologischen Beziehung zueinander stehen. Die Funktion `SDO_GEOM.RELATE` gibt an, in welcher topologischen Beziehung zwei Geometrien zueinander stehen. Der Unterschied zwischen Operatoren und Funktionen wird hier klar ersichtlich.

An dieser Stelle sind die neun von Egenhofer und Franzosa [EF91] aufgestellten topologischen Relationen zwischen Maschen von besonderer Bedeutung (vgl. Tabelle. 5.4). Die Erklärungen sind in Englischer Sprache abgedruckt, weil diese ebenfalls in den Bezeichnungen der Geometrie-Funktionen verwendet werden.

$\partial \cap \partial$	${}^{\circ} \cap {}^{\circ}$	$\partial \cap {}^{\circ}$	${}^{\circ} \cap \partial$	
\emptyset	\emptyset	\emptyset	\emptyset	A and B are disjoint
$\neg \emptyset$	\emptyset	\emptyset	\emptyset	A and B touch
$\neg \emptyset$	$\neg \emptyset$	\emptyset	\emptyset	A equals B
\emptyset	$\neg \emptyset$	$\neg \emptyset$	\emptyset	A is inside of B or B contains A
$\neg \emptyset$	$\neg \emptyset$	$\neg \emptyset$	\emptyset	A is covered by B or B covers A
\emptyset	$\neg \emptyset$	\emptyset	$\neg \emptyset$	A contains B or B is inside of A
$\neg \emptyset$	$\neg \emptyset$	\emptyset	$\neg \emptyset$	A covers B or B is covered by A
\emptyset	$\neg \emptyset$	$\neg \emptyset$	$\neg \emptyset$	A and B overlap with disjoint boundaries
$\neg \emptyset$	$\neg \emptyset$	$\neg \emptyset$	$\neg \emptyset$	A and B overlap with intersecting boundaries

Tabelle 5.4: Die neun von Egenhofer und Franzosa [EF91] definierten topologischen Beziehungen der Ränder (∂) und Inneren (${}^{\circ}$) zweier Maschen.

Typischerweise wird eine Spatialabfrage als zweistufiges Modell formuliert, das aus einem Primär- und einem Sekundärfilter besteht. Der Primärfilter nutzt die Operatoren zur groben Vorfilterung der Daten, damit die exakte Berechnung des Ergebnisses mit Funktionen auf einer kleineren Datenmenge durchgeführt werden kann. Durch die Vorfilterung wird die Datenmenge erheblich reduziert und die Effizienz der Abfrage gesteigert. Die exakte Berechnung muss nämlich nur noch auf einer kleinen Auswahl möglicher Kandidaten (*candidate set*) ausgeführt werden.

Operatoren können – bis auf den `SDO_FILTER` – nur auf Spatialobjekte angewendet werden, die mit zweidimensionalen Koordinaten beschrieben werden. Sie sind für andere Geometrien deaktiviert. Der `SDO_FILTER` beachtet alle Dimensionen. Weitere Operatoren sind:

- `SDO_FILTER` – ermittelt alle Geometrien, die in einer gegebenen topologischen Relation zueinander stehen,

- `SDO_NN` – ermittelt die nächsten Nachbarn einer gegebenen Geometrie,
- `SDO_RELATE` – ermittelt, ob zwei gegebene Geometrien in einer bestimmten topologischen Beziehung zueinander stehen.

Funktionen lassen bei ihren Berechnungen die dritte Dimension einfach außer Acht. Sie berücksichtigen nur die (x,y) -Ebene der Geometrien. Eine Gruppe von Funktionen untersucht die Beziehungen *Vereinigung*, *Differenz*, *Schnittmenge usw.* der Mengenlehre:

- `SDO_GEOM.SDO_DIFFERENCE` - liefert das Differenzobjekt (A *MINUS* B).
- `SDO_GEOM.SDO_INTERSECTION` - liefert das Schnittobjekt (A *AND* B).
- `SDO_GEOM.SDO_UNION` - liefert das Vereinigungsobjekt (A *OR* B).

5.4 Datenbankschema

Im Folgenden wird das aus dem Datenmodell (s. Kapitel 3.2) abgeleitete Datenbankschema vorgestellt. Es basiert auf dem von Reuter [Reu03] beschriebenen Schema, das am Institut für Kartographie und Geoinformation in einer Oracle-Datenbank implementiert ist. Diese Datenbank wird im Bereich der Texturierung stark modifiziert und um Objekttypen für TIN und Dreieck ergänzt.

Entsprechend der Aufgabenstellung dieser Arbeit soll das Datenbankschema nur erweitert werden. Daher stellt das unten vorgestellte Schema eine *Ergänzung* dar, die bestmöglich in das vorhanden Schema „eingepaßt“ wird. Im Anschluss wird eine mögliche *Änderung* des Schemas skizziert, die im Rahmen dieser Arbeit jedoch nicht zu realisieren war. Diese Änderung ist durch konsequente Nutzung von objektorientierten Ansätzen wie Vererbung gekennzeichnet. Zusammen mit einer Implementierung von benutzerdefinierten Methoden könnten eine bessere Performance der Datenbank erreicht werden. Dies ist außerhalb dieser Arbeit näher zu untersuchen.

5.4.1 Erweiterung des vorhandenen DB-Schemas

Abbildung 5.6 zeigt das implementierte Schema im Stil eines *UML*-Diagramms. Dabei entsprechen die Namen der Objekttypen den Klassennamen. Referenzen bzw. Primär- und Fremdschlüssel werden als Attribute geführt und durch Assoziationen ausgedrückt (eine Abbildung im DIN A3 Format findet sich in Anhang B.3 auf Seite 96). Zur besseren Lesbarkeit und klaren Unterscheidung der Datentypen sind diese durch Farben unterschieden: *rot* – relationale Tabellen, *blau* – Objekttypen, *grün* – Collections. Das Diagramm ist horizontal verschiedene Ebenen zu gliedern (von oben nach unten): *Spatial-Typen*, *Aggregation*, *Topologie* und *Material*. Ebenso kann es vertikal in die Spalten *Volumen*, *Flächen*, *Linien* und *Punkte* (von links nach rechts) unterteilt werden. Die unterste Ebene ist jedoch von der vertikalen Teilung ausgenommen.

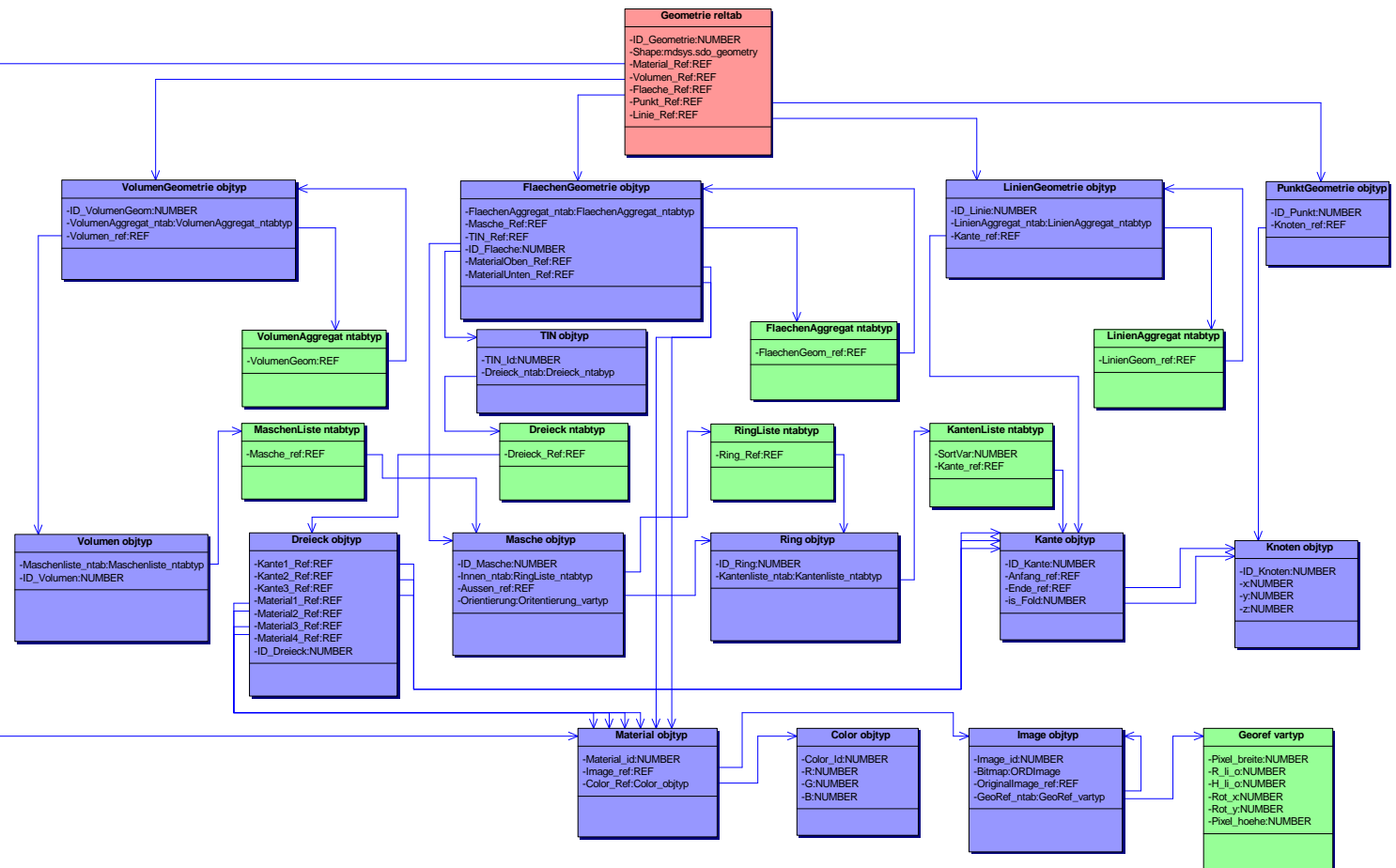


Abbildung 5.6: Datenbankschema als abgewandeltes UML-Diagramm. Objekttypen sind *blau*, Collections *grün* und relationale Tabellen *rot* gekennzeichnet. Das Diagramm ist horizontal in die Ebenen *Spatial-Typen*, *Aggregation*, *Topologie* und *Material* und vertikal in die Bereiche *Volumen*, *Flächen*, *Linien* und *Punkte* unterteilt. Die unterste Ebene ist von der vertikalen Teilung ausgenommen.

In der Datenbank können parallel Daten von 3D-Stadtmodellen und digitalen Geländemodellen verwaltet werden. Dazu werden vier Grunddatentypen verwendet, die hier kurz aufgeführt werden sollen.

- **Punkt_Geometrie** besteht aus Knotenobjekten, die neben den Spatialobjekten als einziger Datentyp Geometrieinformationen enthalten (x-, y- und z-Koordinaten).
- **Linien_Geometrie** verweist auf ein Kantenobjekt und kann aus einer rekursiven Aggregation von Linien_Geometrien bestehen.
- **Flaechen_Geometrie** verweist auf ein Maschenobjekt und kann aus einer rekursiven Aggregation von Flaechen_Geometrien bestehen.
- **Volumen_Geometrie** verweist auf ein Volumenobjekt und kann aus einer rekursiven Aggregation von Volumen_Geometrien bestehen.

Alle Geometrien können selber aus Geometrien dieser Art aggregiert sein, die wiederum Aggregate von Geometrien dieses Typs sein können. Diese rekursive Aggregation wird durch Nested Tables (vgl. Kapitel 5.1.3) erreicht, die Referenzen auf den jeweiligen Geometriotyp speichern.

Zur differenzierten Betrachtung einiger Objekttypen werden diese anhand tabellarischer Übersichten (Objekttabellen) erläutert, die an die Notation des Oracle Handbuchs für objekt-relationale Eigenschaften [Ora02a] angelehnt sind.

Der Objekttyp Flächengeometrie

Table FLAECHENGEOMETRIE_OBJTAB (of FLAECHENGEOMETRIE_OBJTYP)

ID_FLAECHE	FLAECHENAGGREGAT_NTAB	MASCHE_REF
Number NUMBER(10,0)	Nested Table FLAECHENAGGREGAT_NTABTYP	Reference MASCHE_OBJTYP
PK		FK
TIN_REF	MATERIALOBEN_REF	MATERIALUNTEN_REF
Reference TIN_OBJTYP	Reference MATERIAL_OBJTYP	Reference MATERIAL_OBJTYP
FK	FK	FK

Tabelle 5.5: Die Objekttabelle für Flächengeometrie. Ergänzt wurden die Referenzen auf die Objekttypen TIN und Material.

In Tabelle 5.5 sind die Modifikationen des Objekttypen für Flächengeometrien abgebildet. Der bestehende Objekttyp `Flaechengeometrie_objtyp` wird um die Referenzattribute `TIN_Ref` und `MaterialOben_Ref` und `MaterialUnten_Ref` erweitert, die auf entsprechende Objekttypen verweisen (s.u.). Eine Flächengeometrie ist also eine Masche, ein TIN oder ein Flächenaggregat. In der Objekttabelle müssen demnach immer zwei Attribute NULL gesetzt werden, weil eine Flächengeometrie nicht gleichzeitig

zwei der genannten Typen entsprechen kann. Oracle sieht für solche Bedingungen keine Automatismen vor, so dass ein *PL/SQL*-Skript implementiert werden muss, dass das Einhalten dieser Bedingungen überwacht.

Table TIN_OBJTAB (of TIN_OBJTYP)

ID_TIN	DREIECK_NTAB
Number NUMBER(10,0)	Nested Table DREIECK_NTABTYP
PK	

Column DREIECK_NTAB (of DREIECK_NTABTYP (as Table of TIN_OBJTYP))

DREIECK_REF
Reference DREIECK_OBJTYP

Tabelle 5.6: Objekttable TIN (oben) und die im Objekttyp TIN eingebettete Nested Table zur Referenzierung der Dreiecke (unten).

Der Objekttyp `TIN_objtyp` (vgl. Tabellen 5.6) besitzt das Attribut `ID` und die Nested Table `Dreiecke_ntab`. Ein TIN ist durch seine ID spezifiziert und ihm können durch die Nested Table, die Referenzen auf `Dreieck_objtyp` speichert, beliebig viele Dreiecke zugewiesen werden.

Der Objekttyp Dreieck

Table DREIECK_OBJTAB (of DREIECK_OBJTYP)

DREIECK_ID	KANTE1_REF	KANTE2_REF	KANTE3_REF
Number NUMBER(10,0)	Reference KANTE_OBJTYP	Reference KANTE_OBJTYP	Reference KANTE_OBJTYP
PK	FK	FK	FK

MATERIAL1_REF	MATERIAL2_REF	MATERIAL3_REF	MATERIAL4_REF
Reference MATERIAL_OBJTYP	Reference MATERIAL_OBJTYP	Reference MATERIAL_OBJTYP	Reference MATERIAL_OBJTYP
FK	FK	FK	FK

Tabelle 5.7: Objekttable für Dreiecke.

Der Objekttyp `Dreieck_objtyp` (vgl. Tabellen 5.7) besitzt das Attribut `ID_Dreieck` zur Identifizierung des Dreiecks. Darüber hinaus noch drei Referenzattribute `KanteX_Ref`, die auf die Kanten des Dreiecks verweisen und vier Referenzattribute `MaterialX_Ref`, die auf die Materialien verweisen, die diesem Dreieck zugeordnet sind. Wie in Kapitel 4.3.2 angeführt, sind bis zu vier Referenzen auf Materialien notwendig, falls ein Dreieck im Schnitt von vier Luftbildern liegt. In allen anderen Fällen müssen die nicht

benötigten Referenzattribute NULL gesetzt werden. Die Referenzen auf die Dreiecks-kanten zeigen auf den bereits in der DB vorhandenen Objekttypen `Kante_objtyp`.

Alternativ hätte der Objekttyp `Masche` entsprechend erweitert werden können, so dass er TIN-Dreiecke in der oben beschriebenen Form repräsentiert. Dazu hätte für jedes Dreieck der äußere Ring, bestehen aus den drei Kanten, explizit als Ring-Objekt gespeichert werden müssen. Ebenso hätte ein TIN als Flächengeometrie-Objekt gespeichert werden müssen, damit die Aggregation von Dreiecken und TIN repräsentiert werden kann.

Diese alternative Modellierung des DB-Schemas wurde nicht umgesetzt, da das zuvor vorgestellte Schema näher am Datenmodell orientiert ist. Darüber hinaus blieben bei Umsetzung des alternativ vorgestellten Schemas einige Attribute der Objekttypen `Masche` und `Flaechengeometrie` unbesetzt. ADiesen Typen müßten im Gegensatz dazu neue Attribute hinzugefügt werden, die wiederum von allgemeinen Maschen nicht besetzt würden. Die genannten Gründe könnten zu einer Performanceverschlechterung führen. Dies wäre zu untersuchen.

Der Objekttyp Material

Table MATERIAL_OBJTAB (of MATERIAL_OBJTYP)

MATERIAL_ID	COLOR_REF	IMAGE_REF
Number NUMBER(10,0)	Referenece COLOR_OBJTYP	Reference IMAGE_OBJTYP
PK		FK

Table COLOR_OBJTAB (of COLOR_OBJTYP)

COLOR_ID	ROT	GRUEN	BLAU
Number NUMBER(10,0)	Number NUMBER(3,2)	Number NUMBER(3,2)	Number NUMBER(3,2)
PK			

Tabelle 5.8: Objekttable der Materialien (oben) und Farbdefinitionen (unten).

Die Objekttable für Materialien (vgl. Tabelle 5.8) enthält die Attribute `Color_Ref` als Verweis auf eine Farbdefinition und eine Referenz zu einem Image-Objekt `Image_Ref`. Das Color-Objekt besitzt eine ID und drei Attribute, die eine Farbdefinition im RGB-Farbraum durch Anteile von rot, grün und blau beschreiben. Um die Konsistenz zu VRML zu gewähren, werden die Anteile auf Werte von 0 bis 1 normiert. Durch das Erzeugen eines eigenständigen Color-Objekts muss jede Farbe nur einmal definiert und dort gespeichert werden. Sämtliche Materialien können dann auf dieses Objekt verweisen. Es wird redundante Datenspeicherung vermieden.

Wie Cieslik [Cie03] am Beispiel des 3D-Stadtmodells von Hamburg beschreibt, kann das gesamte Modell (Gelände und Stadt) durch die Farbdefinition der Maschen texturiert werden. Liegen die Maschen z.B. differenziert nach Dachfläche, Gebäudewand, Fahrbahn, Gehweg, Grünstreifen und Gewässerfläche vor, so können den entsprechen-

den Maschen geeignete Farbwerte zugeordnet werden.

Der Material-Objekttyp kann von den Objekttypen für Dreiecke und Flächengeometrien gleichermaßen referenziert werden, da beide zur Visualisierung texturiert werden müssen. Flächengeometrien repräsentieren Gebäudefassaden, die möglicherweise aus mehreren Maschen aggregiert sind, für die aber i.d.R. nur eine Texturfarbe bzw. terrestrische Fassadenaufnahme zur Verfügung steht. Somit stellt der Objekttyp `Material_objtyp` eine Erweiterung der Datenbank dar. Er kann für die Texturierung von 3D-Stadtmodellen und digitalen Geländemodellen gleichermaßen verwendet werden kann.

Der Objekttyp Image

Table IMAGE_OBJTAB (of IMAGE_OBJTYP)

IMAGE_ID	BITMAP	GEOREF_VAR	ORIGINAL_IMAGE_REF
Number NUMBER(10,0)	Image ORDSYS.ORDImage	Varray GEOREF_VARTYP	Reference IMAGE_OBJTYP
PK			

Tabelle 5.9: Objekttable der Phototexturen.

Der Objekttyp `Image_objtyp` (vgl. 5.9) repräsentiert eine Phototextur und besitzt die Attribute `Bitmap`, `Georef_var` und `Original_Image_Ref`. Das Rasterbild wird im Oracle-Objekttyp `ORDImage` gespeichert (vgl. Kapitel 5.3.1), die Parameter zur Georeferenzierung (vgl. Kapitel 4.3.1) in einem `VARRAY`. Die Referenz zum Originalbild verweist rekursiv auf den Objekttypen `Image_objtyp`.

Die Speicherung des Originalbildes ist aus zwei Gründen sinnvoll. Beispielsweise bei einer nachträglichen Änderung der Kachelgröße kann das Originalbild zur Neugenerierung der Kacheln verwendet werden. Würde das Originalbild nicht gespeichert, so müssten die Kacheln im umgekehrten Prozess erst wieder zum Originalbild zusammengefasst werden. Ebenso wichtig ist das Beibehalten des Originalbildes für die Fassadentexturierung⁵, um die Projektion rückgängig machen zu können bzw. sie nachträglich zu verändern.

Geometrie_reltab

Die in Kapitel 5.3.2 angeführten Einschränkungen bzgl. der räumlichen Indizierung führen dazu, dass `Geometrie_reltab` (vgl. Tabelle 5.10) die einzige relationale Tabelle des ansonsten objektorientiert angelegten Schemas ist. Hier werden die Spatialgeometrien aller Objekte gespeichert. Da die Geometrieinformationen der Objekte bereits in der topologischen Beschreibung durch Knoten, Kanten und Maschen codiert sind,

⁵Parallel zu dieser Arbeit wird am Institut für Kartographie und Geoinformation eine Anwendung zur projektiven Verzerrung von Fassadenaufnahmen zur Texturierung von 3D-Gebäudemodellen entwickelt, die ebenfalls das hier beschriebene Datenbankschema zur Speicherung der Fotos nutzt. Es werden neben den verzerrten Fotos die Originalbilder und ihre Projektionsparameter gespeichert.

Table GEOMETRIE_RELTAB

ID_GEOMETRIE	SHAPE	MATERIAL_REF
Number NUMBER(10,0)	Spatial MDSYS.SDO_GEOMETRY	Reference MATERIAL_OBJTYP
PK		FK

Punkt_REF	Linie_REF
Reference PUNKTGEOMETRIE_OBJTYP	Reference LINIENGEOMETRIE_OBJTYP
FK	FK

Flaeche_REF	Volumen_REF
Reference FLAECHEGEOMETRIE_OBJTYP	Reference VOLUMENGEOMETRIE_OBJTYP
FK	FK

Tabelle 5.10: Relationale Tabelle der Spatial Geometrien.

führt die zusätzliche Speicherung als Spatialobjekt zu doppelter Datenführung. Das redundante Speichern stellt auf der anderen Seite aber die einzige Möglichkeit dar, sowohl die topologischen Zusammenhänge der Objekte zu repräsentieren als auch die Vorteile des räumlichen Indexes zu nutzen.

An dieser Stelle wird das vorhandene Datenbankschema ebenfalls modifiziert. Es werden neue Referenzattribute in `Punktgeometrie_objtyp` und in `Material_objtyp` eingefügt, damit auch deren Darstellung als Spatialobjekte gespeichert werden kann.

Eine Repräsentation als Spatialobjekt ist auch für Punktgeometrien sinnvoll, da z.B. Kanaldeckel oder Straßenschilder als Punktobjekte in die DB eingefügt werden könnten und dann mit einem Spatialindex verwaltet werden sollten. Eine Repräsentation der Eckpunkte eines 3D-Stadtmodells oder der Flächenpunkte eines TINs als Spatialobjekte bringt hingegen keinen Vorteil, da eine isolierte Abfrage dieser Daten wenig Sinn macht.

Werden die Materialobjekte bzw. die ihnen zugeordneten Orthophotos ebenfalls als Spatialgeometrien gespeichert, so können sämtliche Vorteile räumlicher Abfragen auch für Luftbilder genutzt werden. Auf diese Weise lassen sich sehr leicht einzelne Dreiecksmaschen den entsprechenden Luftbildern zuordnen. Zum Zeitpunkt des Einfügens eines digitalen Geländemodells in die Datenbank kann nicht vorausgesetzt werden, dass auch Luftbilder dieses Gebietes zur Verfügung stehen. Umgekehrt können Orthophotos unabhängig von einem ihrem Deckungsgebiet entsprechendem DGM in die DB aufgenommen werden. Mit Hilfe des in Anhang C gelisteten *PL/SQL*-Skripts kann die Verknüpfung zwischen Material-Objekttabelle und der relationalen Tabelle `Geometrie` nachträglich generiert werden.

Die Repräsentation eines Orthophotos als Spatialobjekt ist ein Rechteck, dessen Eckpunkte leicht aus den Georeferenzierungsinformationen (vgl. Kapitel 4.3.1) berechnet werden können.

5.4.2 Mögliche Änderungen des DB-Schemas

Da das oben vorgestellte Datenbank-Schema auf Grundlage einer bestehenden Datenbank erstellt werden musste, konnten keine Änderungen am vorhandenen Schema vorgenommen werden. Dieses hat jedoch folgende Einschränkung: TIN, Flächenaggregat und Masche sind spezielle Flächengeometrien, die in der objektorientierten Programmierung als *Unterklassen* bezeichnet werden und aus der *Oberklasse* abgeleitet werden können. Im abstrakten Datenmodell in Kapitel 3.2 ist der Sachverhalt entsprechend modelliert. Im DB-Schema war diese Idee allerdings nicht umsetzbar, da hierzu das gesamte DB-Schema hätte abgeändert werden müssen. An dieser Stelle wird die Idee der Änderung des DB-Schemas jedoch kurz skizziert. In späteren Untersuchungen sollte diese Änderung implementiert, getestet, und mit dem oben vorgestellten Schema auf Performance-Unterschiede untersucht werden.

Wird eine Unterklasse aus einer Oberklasse abgeleitet, so werden alle Eigenschaften (Attribute und Methoden) der Oberklasse in die Unterklasse kopiert (vererbt). In relationalen DBMS ist das Vererbungs-Konzept nicht vorgesehen. Oracle als objektrelationale Datenbank unterstützt dieses Konzept jedoch. Es könnte also auf die oben beschriebenen Objekttypen angewendet werden. Der entsprechende Teil des DB-Schemas könnte wie in Abbildung 5.7 modelliert werden.

Die Objekttypen `Masche_objtyp` und `FlaechenAggregat_objtyp` könnten als Unterobjekte (Subtypen) des Objekttyps `FlaechenGeometrie_objtyp` (Supertyp) erzeugt werden. Dazu wäre der Zusatz `UNDER` im `SQL`-Befehl notwendig:

```
CREATE OR REPLACE masche_objtyp UNDER flaechenaggregat_objtyp.
```

Die Subtypen erben alle Attribute, die jedoch nicht besetzt werden müssen. Ihnen können auch weitere Attribute hinzugefügt werden. Aus diesen Subtypen würden `Dreieck_objtyp` und `TIN_objtyp` als weitere Subtypen abgeleitet (vgl. Abbildung). Ein Dreieck wäre demnach eine besondere Masche und ein TIN ein besonderes Flächenaggregat, wobei alle Typen Subtypen von Flächengeometrie wären.

Gegenüber dem o.g. Schema wäre die Nested Table zur Aggregation der Flächen nicht mehr als Attribut der Flächengeometrie gespeichert, sondern durch ein eigenständiges Objekt `Flaechenaggregat` repräsentiert. Das wäre vorteilhaft, da das Objekt `Masche`, das aus Flächengeometrie abgeleitet ist, diese Nested Tables zur Aggregation ebenso wenig benötigt wie die daraus abgeleiteten Dreiecke. Die Referenzattribute von `Flaechengeoemtrie` zur den Objekttypen für Maschen und TINs würden wegfallen, da diese spezielle Flächengeometrien wären.

Diese Modellierung durch Vererbung kann ebenso auf die Objekttypen `Volumengeometrie` und `Liniengeometrie` angewendet werden.

Ein möglicher Nachteil der hier vorgestellten Modellierung wäre die Tatsache, dass die Oberklassen viele Attribute vererben, die in den Unterklasse nicht benötigt werden und somit `NULL` gesetzt werden müssen. Dieser Nachteil ist aber gegen die Vorteile der umfangreichen Ausnutzung der objektorientierten Möglichkeiten von Oracle abzuwägen.

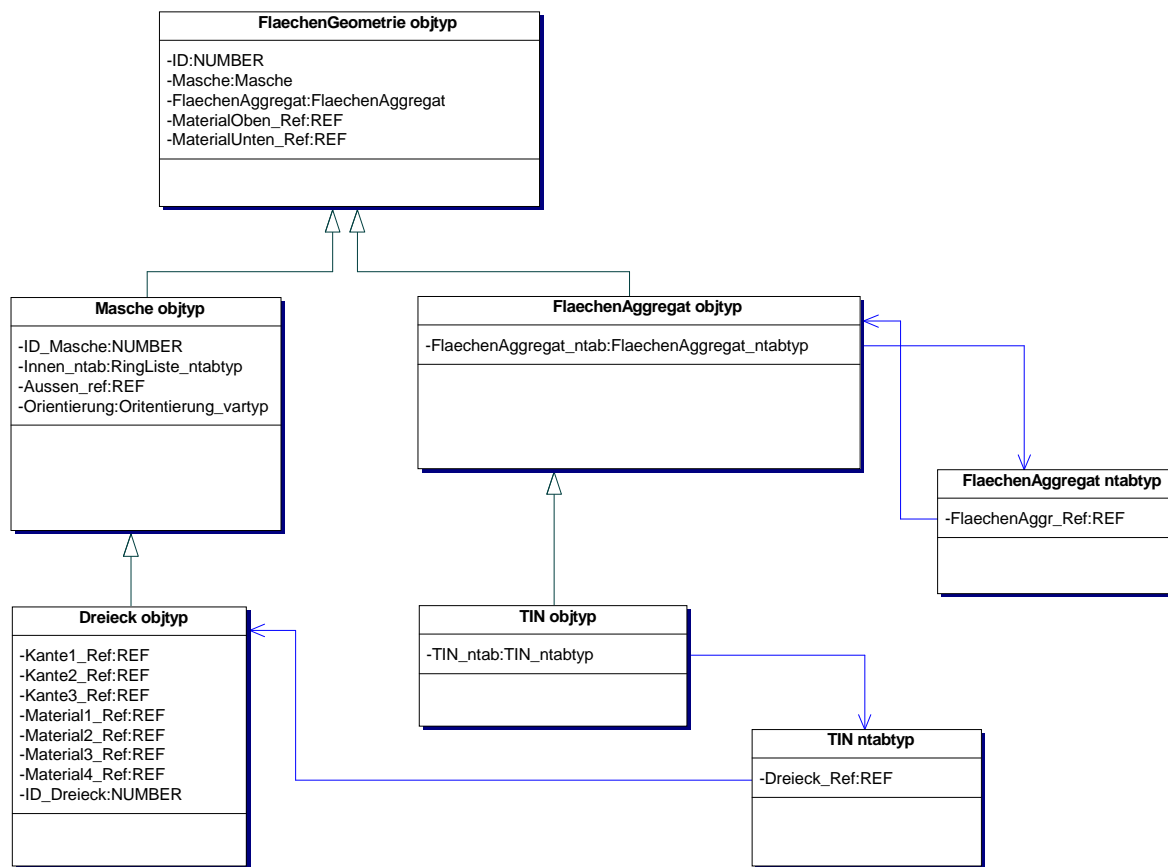


Abbildung 5.7: Mögliche Änderung des DB-Schemas durch Nutzen der Vererbung von Objekttypen.

5.5 Schnittstellen

Zum Im- und Export von Daten wurden Schnittstellen in *Java* implementiert, deren Klassen im UML-Diagramm in Abbildung 5.8 angeführt sind (in Anhang B.4 findet sich eine Übersicht im DIN A3 Format). Der Quellcode aller Klassen und eine abgeleitete *javadoc*-Beschreibung befinden sich auf der zu dieser Arbeit gehörenden CD-ROM (vgl. D).

Grundlage der Schnittstellen-Funktionalität bildet die Klasse *Db*, die Methoden zur Datenbankkommunikation zur Verfügung stellt. Dies sind zum einen die Methoden `connect` oder `disconnect`, die Verbindung zur Datenbank herstellen oder trennen, und zum anderen Methoden wie `writeNodes` und `getNode`s, die Daten in die Datenbank schreiben oder aus der DB lesen.

Die Klassen *NodeTopo*, *EdgeTopo* und *TriangleTopo* repräsentieren die topologischen Zusammenhänge der Geometrien. Zusätzlich steht für Dreiecke die Klasse *TriangleGeom* zur Verfügung, die ein Dreieck durch die Koordinaten seiner Eckpunkte beschreibt.

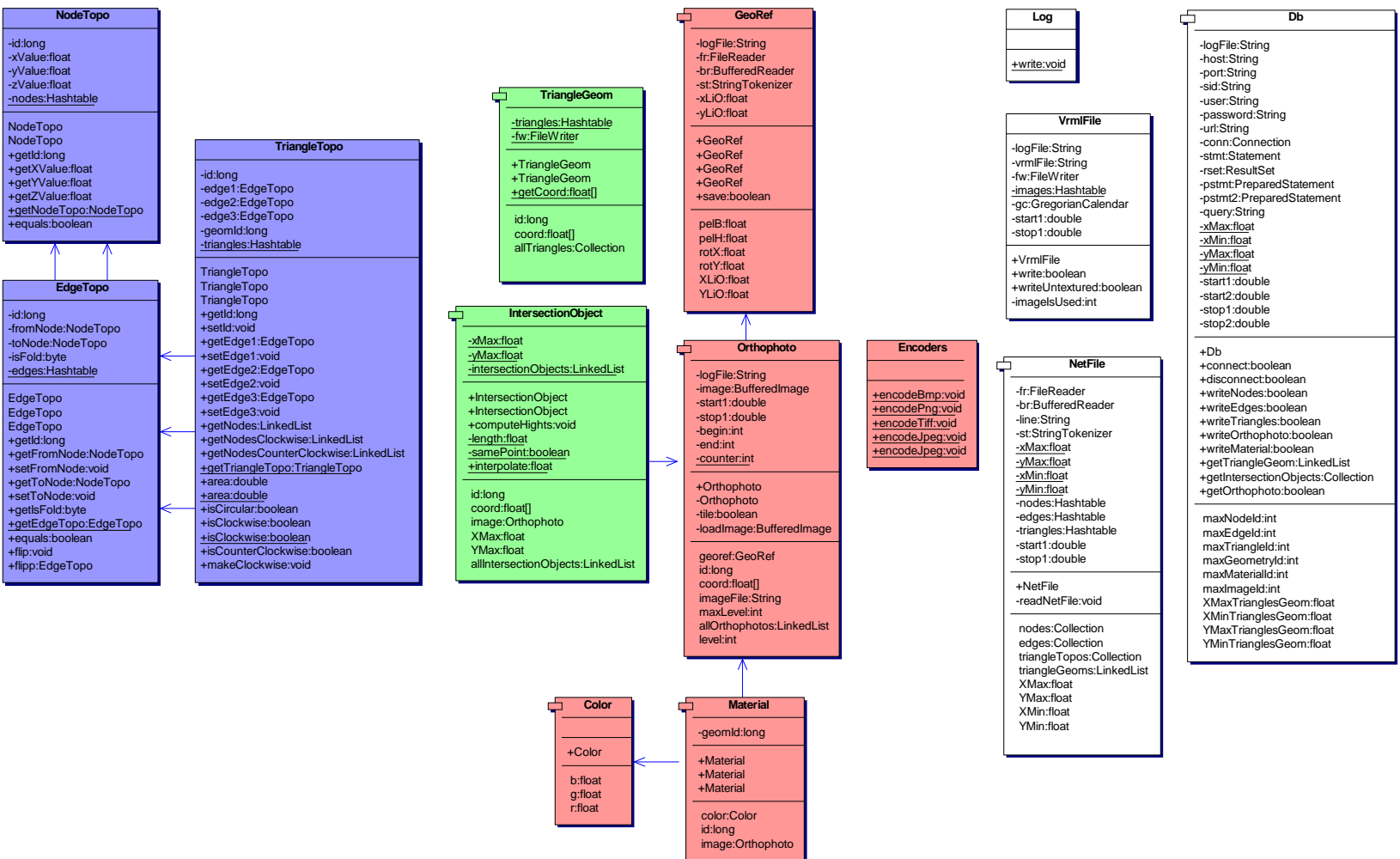


Abbildung 5.8: UML-Klassendiagramm der für die Schnittstellenfunktionalität implementierten Klassen.

```

NODES
2 0 4 2
3 3 0 5
4 8 1 3
EDGES
101 2 3 1
102 3 4 1
103 4 2 1
TRIANGLES
10001 101 102 103

```

Listing 5.7: Beispiel des *net* Formats zur Beschreibung von TINs.

5.5.1 Import in die Datenbank

Als Importformat der DGM-Daten wird das *.net* Format des Programmsystems *ArcInfo* der Fa. *ESRI*⁶ unterstützt. Orthophotos können theoretisch in beliebigen Rasterbildformaten verwendet werden. Es wurde aber lediglich eine Unterstützung des *tif*-Formats (Tag(ged) Image File) implementiert.

DGM

Das *net* Format ist ein Textformat, das ein TIN durch seine topologischen Bestandteile beschreibt (vgl. Listing 5.7).

Die vier Elemente einer Zeile entsprechen im jeweiligen Abschnitt folgenden Daten:

- **NODES** – ID des Knotens, X-Koordinate, Y-Koordinate, Z-Koordinate.
- **EDGES** – ID der Kante, ID des Startknotens, ID des Endknotens, Typ der Kante (1 = Innenkante des TINs, 0 = Außenkante des TINs).
- **TRIANGLES** – ID des Dreiecks, ID der Kante1, ID der Kante2, ID der Kante3.

Die Klasse `NetFile` bietet die Methode `readNetFile`, welche die ihr zugewiesene Datei ausliest und entsprechende Instanzen der o.g. Topologie- und Geometrieklassen und der Klasse `TriangleGeom` erzeugt. Diese werden in statischen `Hashtables`⁷ verwaltet und sind über `getX` Methoden zugänglich.

Die so gelesenen Daten können mit den `writeX` Methoden der Klasse `Db` in die entsprechenden Tabellen der Datenbank geschrieben werden. Dazu werden die in Abschnitt 5.2.3 erläuterten `PreparedStatements` verwendet, da sie eine enorme Geschwindigkeitssteigerung der Datenübertragung gegenüber `Statements` bieten.

⁶<http://www.esri.com>

⁷In `Hashtables` werden Schlüssel (hier ID der Geometrie) bestimmte Werte (hier Instanzen von `xTopo`) zugeordnet (key value pairs). Auf die Werte kann einzeln über den Schlüssel oder im Ganzen in Form einer `Collection` zugegriffen werden.

Orthophotos

Die Wahl des *tif*-Formats für Orthophotos beruht auf zwei Argumenten. Zum einen handelt es sich um ein plattformunabhängiges, verlustfrei komprimierbares Format. Zum anderen stand die *Java*-Klasse `Encoders` der Fa. *lat/lon* zur Kodierung dieses Dateiformats zur Verfügung⁸.

Orthophotos und ihre Georeferenzierungsinformationen sind in den Klassen `Orthophoto` und `GeoRef` implementiert. Die Klasse `Orthophoto` speichert das entsprechende Rasterbild als ein `BufferedImage` im Attribut `image`. Dem Konstruktor wird neben dem Dateinamen der *tif*-Datei auch ein Wert für die Anzahl der Kachelungsstufen übergeben, die durch rekursiven Aufruf automatisch erzeugt werden.

Das Rasterbild muss nicht zwingend auf dem Clientrechner vorliegen. Die *Java*-Klasse kann leicht dahingehend angepasst werden, dass Bilder über eine URL spezifiziert werden, und Bilder aus dem Internet geladen werden können. Es wäre durchaus denkbar, Bilder über einen frei verfügbaren WebMapServer (WMS⁹) zu beziehen. Mit entsprechenden Methoden könnten Karten oder Orthophotos der benötigten Größe von solchen Servern geladen werden, indem der Parameter für den räumlichen Ausschnitt mit einer *for*-Schleife iterativ verändert wird.

Zur Kachelung wird das Bild in jedem Schritt in vier gleich große Teile zerlegt, die in eigenen Dateien gespeichert werden. Parallel werden die Georeferenzierungsdaten berechnet und in einer gleichnamigen *tfw*-Datei gespeichert. Der Name der jeweiligen Datei setzt sich auch dem Dateinamen des Bildes der Kachelungsstufe 0 und dem entsprechenden Index des in Abbildung 5.9 dargestellten Schemas zusammen.

Die Kachelungsstufe wird bei Objekten der Klasse `Orthophoto` als Attribut geführt. In dieser Klasse wird ebenfalls eine statische Variable vom Typ `LinkedList` geführt, die Zeiger auf alle erzeugten Instanzen verwaltet. Auf die Liste der Instanzen kann mit der Methode `getAllOrthophotos` zugegriffen werden kann.

Um die Orthophotos in die Datenbank zu schreiben wird die Methode `writeOrthophoto` der bereits vorgestellten Klasse `Db` verwendet. Dieser Methode wird unter anderem die Liste mit Zeigern auf die erzeugten Instanzen und die `Orthophoto` Instanz der Kachelungsstufe 0 übergeben. Die Kacheln und das Originalbild werden dann in die DB geschrieben. An dieser Stelle kann kein `PreparedStatement` verwendet werden, da Rasterbilder mit Hilfe einer *PL/SQL*-Anweisung in die Datenbank geschrieben werden, die nur als `Statement` ausgeführt werden kann.

5.5.2 Export nach VRML

Die Exportfunktionalität ist auf das Erzeugen einer *VRML* Datei beschränkt, die durch die Klasse `VrmlFile` repräsentiert wird. Bereits beim Aufruf des Konstruktors wird

⁸<http://www.latlon.de>

⁹Ein WMS ist ein durch das OpenGIS Consortium spezifizierter und somit interoperabler Kartenserver, der Karten in Form von Rasterbildern liefert. In der URL können unter anderem Parameter über den räumlichen Ausschnitt (z.B. Koordinatensystem, Boundingbox), die gewünschten Layer und das Grafikformat der Ausgabedatei spezifiziert und dem Server übermittelt werden.

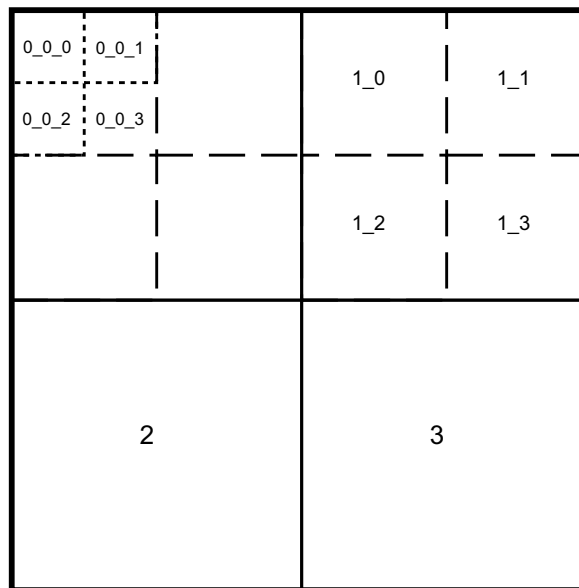


Abbildung 5.9: In jeder Kachelungsstufe wird das Rasterbild in vier gleich große Teile zerlegt. Es werden die gezeigten Indizes zur Erweiterung des Dateinamen verwendet.

eine Datei mit gegebenem Dateinamen angelegt, und ein Dateikopf mit Metadaten wie Erstellungsdatum erzeugt. Die Definition des `Background`- und `NavigationInfo`-Knotens wird ebenfalls schon beim Aufruf des Konstruktors geschrieben. Es werden standardmäßig die Hintergrundfarbe `blau` und Navigationsart `Examine` gewählt. Durch die beiden Methoden `writeUntextured` und `write` wird ein Viewpoint erzeugt, von dem aus die gesamte Szene betrachtet werden kann. Darüber hinaus werden die Geometrie-Knoten erzeugt. Da der *VRML*-Standard keine topologischen Beziehungen der Objekte unterstützt, genügt es, die Spatial-Objekte der Geometrien aus der Datenbank abzufragen. Wie in Kapitel 4.2 ausgeführt, können nur Koordinatenwerte mit einer bestimmten Anzahl von Stellen verwendet werden. Daher werden Gauß-Krüger-Koordinaten zur Visualisierung um die Koordinatenwerte des linken unteren Punkts gekürzt.

Beide Texturierungsmethoden greifen auf die Methode `getTriangleGeom` der Klasse `Db` zurück, welche die drei Eckpunkte der Spatial-Repräsentation von Dreiecken ausliest und deren Reihenfolge gegen den Uhrzeigersinn (counter clockwise (ccw)) orientiert. Die zwingende ccw-Orientierung der Dreiecke geht auf die Texturierung von Polygonen in *VRML* nach „Daumenregel der rechten Hand“ zurück. Werden die TIN-Dreiecke in die (x, y) -Ebene projiziert¹⁰, so kann die Umlaufrichtung der Eckpunkte durch die Gauß'sche Flächenformel bestimmt werden:

$$F = \frac{1}{2} \sum_{i=1}^3 (x_i - x_{i+1})(y_i + y_{i+1}).$$

Wird $x_4 = x_1$ und $y_4 = y_1$ gesetzt, so sind die Punkte ccw orientiert, wenn $F > 0$ gilt. Die Methode `getTrianglesGeom` prüft die Orientierung und kehrt die Reihenfolge der

¹⁰Da es in TINs keine senkrechten Flächen gibt, haben in der Projektion alle Dreiecke eine endliche Fläche.

Punkte ggf. um. Durch dieses Vorgehen ist sichergestellt, dass auch bei Berücksichtigung der z -Koordinate die „oben“ liegende Seite des Dreiecks texturiert wird.

Einheitliche Textur

Die Methode `writeUntextured` wird gewählt, wenn auf die Fototextur verzichtet wird, keine Orthophotos des gewünschten Bereichs in der DB vorliegen oder keine Farbdefinitionen für die Masche vorhanden ist. Alle im angegebenen Auswahlfenster liegenden Spatialrepräsentationen von Dreiecken werden mit der Methode `getTriangleGeom` abgefragt und der Methode `writeUntextured` übergeben, die für jedes Dreieck einen Kindknoten des Transform-Knotens erzeugt. Die Reihenfolge der Punkte eines Dreiecks wird bereits bei der Datenbankabfrage entgegen dem Uhrzeigersinn orientiert. Alle Dreiecke erhalten eine voreingestellte Farbe.

Differenzierte Textur

Sind die Dreiecke der DB mit einem Materialobjekt verknüpft, das ein Farbattribut besitzt, können diese RGB-Farbdefinitionen zusammen mit den Dreiecken abgefragt und im `Material`-Knoten angebracht werden.

Phototextur

Sollen Orthophotos zur Texturierung verwendet werden, so wird die Methode `write` verwendet, der ebenfalls nur Koordinaten des darzustellenden räumlichen Ausschnitts zu übergeben sind.

In der Methode wird das in Kapitel 4.3.2 vorgestellte Konzept zur speicheroptimierten Verwendung von Phototexturen implementiert. Das Problem dieses Ansatzes liegt in der Berechnung der Schnittpunkte von Orthophoto und TIN sowie im Einfügen dieser Punkte und der entsprechenden Kanten in das TIN. Zur Lösung des Problems wird die in Kapitel 5.3.2 beschriebene Spatialfunktion `SDO_GEOM.SDO_INTERSECTION` verwendet. Sie wird auf alle Orthophotos und Dreiecke im Auswahlfenster angewendet, und gibt deren Schnittobjekte zurück. Dabei wurde die Toleranz zur Unterscheidung zweier Punkte auf $1mm$ gesetzt, was der Genauigkeit von Gauß-Krüger-Koordinaten entspricht. Es müssen noch weitere Untersuchungen angestellt werden, die die Zuverlässigkeit dieses Verfahrens betrachten, wenn die Fläche der Schnittobjekte in der Größenordnung dieses Toleranzbereichs liegt.

Das Problem der Texturierung von Flächen mit mehr als einem Rasterbild wird durch diesen Ansatz (auf ein kleineres Problem) reduziert. Die Funktion wird nur auf die Projektion der Daten in x,y -Ebene angewendet. Demnach sind die Schnittobjekte ebenfalls nur zweidimensional¹¹. Zur Beseitigung dieser Mängel werden neben den Schnittobjekten die entsprechenden „Ursprungsdreiecke“ und die ID der zugeordneten Materialien

¹¹Die mit der Funktion berechneten Schnittobjekte haben zwar eine z -Koordinate, diese ist jedoch falsch (vgl. [Ora02e]).

von der Datenbank abgefragt. Somit ist es möglich, die Dreiecke mit den Schnittobjekten zu vergleichen und die Höhen der Eckpunkte entweder zu übernehmen oder zu interpolieren (vgl. Abbildung 5.10).

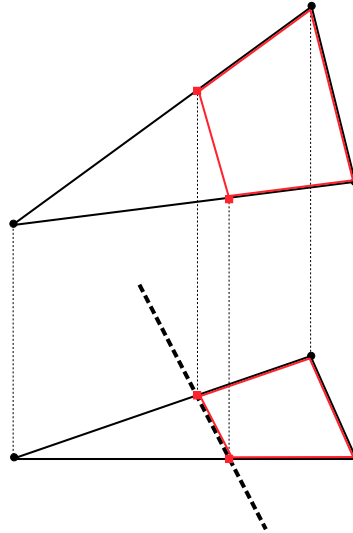


Abbildung 5.10: Schnittobjekte (Viereck) werden von Spatial nur in 2D ausgegeben. Daher müssen die Höhen aus den ursprünglichen Daten (Dreieck) übernommen oder interpoliert werden

Dazu werden die Lagekoordinaten des Schnittobjekts mit denen des Dreiecks verglichen. Sind diese identisch, so kann die Höhe des Dreieckspunktes übernommen werden, andernfalls muss sie interpoliert werden. Da die Punktereihenfolge in beiden Objekten gegen den Uhrzeigersinn orientiert ist (Spatialvorgabe!), sind die Punkte zwischen denen interpoliert werden muss eindeutig festgelegt. Die so berechneten 3D-Schnittobjekte, die Drei- oder Vierecke sein können, werden durch die Klasse `IntersectionObjects` repräsentiert.

Die benötigten Orthophotos bzw. Kacheln werden in einem zweiten Schritt über die ID der Materialien abgefragt. Zur Optimierung der Rechenzeit sollte versucht werden, durch nur eine Datenbankabfrage alle gewünschten Daten zu bekommen (vgl. `Statement` vs. `PreparedStatement`).

Zum Generieren der *VRML*-Datei werden der Methode `write` alle erzeugten Instanzen von `IntersectionObjects` übergeben, deren Koordinaten als `IndexedFaceSet` in einen Geometry-Knoten geschrieben werden.

Die Texturierung von Flächen kann in *VRML* nur erfolgen, wenn die Flächen planar sind. Dreiecke sind ebene Flächen, da drei Punkte eine Fläche definieren. Schnittobjekte mit vier Eckpunkten sind bis auf die endliche Rechengenauigkeit von *Java* ebenfalls planar, da ein oder zwei Punkte durch Interpolation berechnet wurden. Die hier beschriebenen Elemente der Klasse `IntersectionObject` entsprechen also diesen Planaritätsanforderungen.

Zur Texturierung werden `TextureCoordinate`-Knoten erzeugt, deren Koordinaten leicht zu berechnen sind, da Dreieckspunkte und Orthophoto-Eckpunkte im Gauß-Krüger-System vorliegen.

Zur Effizienzsteigerung wird der **Appearance-Knoten** bei der erstmaligen Verwendung eines Orthophotos explizit definiert (**DEF**). Wird das gleiche Orthophoto ein zweites Mal verwendet, muss lediglich diese Definition aufgerufen werden (**USE**).

Kapitel 6

Zusammenfassung und Ausblick

Diese Arbeit stellt ein abstraktes Datenmodell zur Beschreibung des Geländereiefs durch ein digitales Geländemodell vor, das durch ein Dreiecksnetz (*Triangular Irregular Networks – TIN*) repräsentiert wird und auf die topologischen Zusammenhängen seiner Bestandteile Knoten, Kanten und Dreiecke aufbaut. Desweiteren wurde die Beschreibung der Oberflächenbeschaffenheit durch Farbwerte oder Rasterbilder in das Modell integriert, das mit einem bereits existierenden Datenmodell für 3D-Stadtmodelle vereinigt wurde. Aus dem Gesamtmodell wurde ein entsprechendes Datenbankschema abgeleitet, das eine Erweiterung der am Institut für Kartographie und Geoinformation der Universität Bonn implementierten Datenbank darstellt.

Ein Problem stellte die Optimierung des Speicherbedarfs bei der Verwendung von Orthophotos dar. In der hier vorgestellten Lösung wird das Orthophoto unter Berücksichtigung der Georeferenzierung (rekursiv) gekachelt. Die texturierte Visualisierung der Geländeoberfläche wird auf der Grundlage des interoperablen Standards der *Virtual Reality Modeling Language (VRML)* realisiert. Um die Texturierung von TIN-Dreiecken, die von zwei oder vier Luftbildern beschrieben werden, dem *VRML*-Standard entsprechend zu ermöglichen, wird eine 2D-Darstellung der Orthophoto-Kacheln mit dem TIN verschnitten und entsprechend neue TIN-Kanten eingefügt. Die so entstehenden Schnittobjekte werden vollständig von einem Orthophoto beschrieben und können texturiert werden.

Das vorgestellte Datenbankschema wurde im DBMS *Oracle 9i* unter Nutzung der Spatial-Erweiterung für räumliche Daten (Geodatenbank) implementiert. Es werden Objekttypen und -tabellen zur Beschreibung der topologischen Zusammenhänge verwendet. Alle Geometrien werden zusätzlich in einer relationalen Tabelle im Spatial Datentyp für raumbezogene Objekte gespeichert, um effiziente Abfragen zu ermöglichen. Da Orthophotos ebenfalls als Spatial-Objekte repräsentiert werden, kann die oben beschriebene Verschneidung von Orthophotos und TIN durch die *Intersection*-Funktion von der DB gelöst werden. Das Nichtberücksichtigen der dritten Dimension wird durch ein entsprechendes *Java*-Programm aufgefangen, das den Schnittobjekten die entsprechenden Höhenwerte zuweist bzw. diese interpoliert. Somit ist die Planarität auch bei viereckigen Schnittobjekten gewährleistet.

Werden Orthophotos und TINs unabhängig voneinander in die DB eingefügt, so kann eine Referenzierung der Dreiecke zum jeweiligen Orthophoto nicht direkt erzeugt werden. Die gegenseitige Referenzierung wird durch ein *PL/SQL*-Skript gelöst, das nach dem Einfügen von neuen Daten aufgerufen werden kann. Es nutzt Oracle Spatial zur Suche sich schneidender Objekte. Dieses Vorgehen ermöglicht den zeitversetzten Import von Orthophotos und TINs.

Abschließend wurden Schnittstellen zum Datenimport von *ARCInfo* und Datenexport nach *VRML* vorgestellt, die in *Java* implementiert wurden. Orthophotos können zum Import mit Hilfe der entsprechenden Klasse durch Vorgabe einer Anzahl von Kachelungsstufen rekursiv zerlegt und in die Datenbank geschrieben werden. Zum Datenexport werden die vom DBMS berechneten Schnittobjekte von TIN und Orthophoto mit Phototexturen in *VRML* dargestellt. Durch Nutzen des *VRML*-Standards wird ein plattformunabhängiges Format verwendet, das zur Interoperabilität des Gesamtsystems beiträgt.

Über die Aufgabenstellung dieser Arbeit hinaus bleiben einige Probleme unbeantwortet, die weiterer Untersuchungen bedürfen.

So sind z.B. das Datenbankschema und die verwendeten SQL-Statements hinsichtlich ihrer Performance zu untersuchen und ggf. zu optimieren. Ansatzpunkte wären die Verwendung von *PreparedStatements* beim Laden von Luftbildern oder die Verwendung eines *Batch-Stacks* beim Import der Geometrien. Auch sollte das Laufzeitverhalten der Anfragen mit alternativen Formulierungen der SQL-Befehle evaluiert werden. Zur Optimierung der *Java*-Klassen wäre eine eingehende Untersuchung und Verwendung des Packages *oracle_sdo* nötig, das Klassen und Methoden zur Repräsentation von Spatial-Objekten bietet. Außerdem muss die Georeferenzierung der Orthophotos, wie im Datenmodell gefordert, so angepaßt werden, dass die Angabe unterschiedlicher SRS möglich ist. In der vorliegenden Version ist diese auf das Gauß-Krüger-System beschränkt.

Dem Datenbankschema können Objekte oder Attribute hinzugefügt werden, die das Speichern von Metadaten wie Adressen ermöglichen. Points of Interest (POI) liegen in großer Zahl in digitaler Form vor und werden von WebFeatureServern (WFS) ähnlich einem WebMapServer angeboten. Werden diese POI durch eine Spatial-Repräsentation in die Datenbank eingefügt, so ist es möglich, dem Metadatenattribut einer Volumenometrie (Gebäude) die Informationen des POI zuzuordnen, der innerhalb dieses Volumenkörpers liegt. Diese Zuordnung kann in ein *PL/SQL* Skript implementiert werden.

Neue Versionen des DBMS Oracle sind dahingehend zu untersuchen, ob die angemerkteten Unzulänglichkeiten der Version 9i behoben sind. Das sind zum einen die Fähigkeit, einen Spatial-Index für Objekttabellen zu erzeugen, und zum anderen die Implementierung eines Datentypen für Volumenobjekte. Alternativ können benutzereigene Spatial-Funktionen in die DB integriert werden, die eine bessere Unterstützung für 3D-Daten bieten.

In Bezug auf die Visualisierung ist der angesprochene *X3D*-Standard auf mögliche Vorteile oder Performance-Verbesserungen zu untersuchen. Darüberhinaus wären u.a.

im *VRML*-Modell anklickbare *Touch-Sensors* denkbar, welche die zu einem Objekt gespeicherten Metadaten anzeigen oder vordefinierte und im *VRML*-Browser auswählbare *Viewpoints* für jedes Gebäude, das POI-Informationen besitzt.

Ein sehr umfangreicher weiterer Schritt wäre eine echte Verschneidung von DGM und 3D-Stadtmodell. Bislang werden beide Teile getrennt entwickelt, so dass die Unterkante eines Gebäudes nicht als (Bruch)Kante des TINs berechnet oder modelliert wurde. Dieses Vorgehen kann dazu führen, dass die Gebäude knapp über dem TIN schweben oder in halber Höhe von diesem geschnitten werden. Eine solche Schnittberechnung sollte unter Zuhilfenahme der Spatial-Funktionen erfolgen, da eine numerische Berechnung aufgrund des hohen Datenvolumens zu aufwendig wäre.

Anhang A

VRML-Beispiele

VRML-Code der Beispiele aus Kapitel 4.2

A.1 Texturierung ohne Angabe von Parametern

```
#VRML V2.0 utf8
```

```
Transform{  
  children [  
    Shape { #Shape0  
      appearance Appearance {  
        texture ImageTexture{  
          url "textur1024.jpg"  
        }  
      } #appearance  
      geometry IndexedFaceSet {  
        coord Coordinate {  
          point [  
            0 1 0  
            6 0 0  
            8 3 0  
          ] #point  
        } #Coordinate  
        coordIndex [0, 1, 2]  
      } #IndexedFaceSet  
    } #Shape0  
  ] #children  
} #Transform
```

```
Background {  
  skyColor 0.0 0.0 0.5  
} #Background
```

A.2 Texturkoordinaten

```
#VRML V2.0 utf8
```

```
Transform{
  children [

    Shape { #Shape0
      appearance DEF tex1 Appearance {
        texture ImageTexture{
          url "textur1024.jpg"
        }
      } #appearance
      geometry IndexedFaceSet {
        coord Coordinate {
          point [
            0 1 0
            6 0 0
            8 3 0
          ] #point
        } #Coordinate
        coordIndex [0, 1, 2]
        texCoord TextureCoordinate {
          point [
            0.1 0.4
            0.6 0.3
            1.5 0.8
          ]
        } #TextureCoordinate
      } #IndexedFaceSet
    } #Shape0

  ] #children
} #Transform

Background {
  skyColor 0.0 0.0 0.5
} #Background
```

A.3 Transformation der Rasterbildtextur

```
#VRML V2.0 utf8
```

```
Transform{
  children [

    Shape { #Shape0
      appearance DEF tex1 Appearance {
        texture ImageTexture{
          url "textur1024.jpg"
        }
        textureTransform TextureTransform {
          center 0.2 0.4
          scale 2 0.7
          translation 0.3 0.7
        } #TextureTransform
      } #appearance
      geometry IndexedFaceSet {
        coord Coordinate {
          point [
            0 1 0
            6 0 0
            8 3 0
          ] #point
        } #Coordinate
        coordIndex [0, 1, 2]
      } #IndexedFaceSet
    } #Shape0

  ] #children
} #Transform

Background {
  skyColor 0.0 0.0 0.5
} #Background
```


Anhang B

Datenbankschema

B.1 SQL-Befehle zum Generieren des DB-Schemas

```
/*=====
   Datenbankschema auf Grundlage von Objekttypen und Objekttabellen
   zur Repräsentation und Verwaltung von TINs.

   Erstellt am 23.03.2004 von Joerg Schmittwilken
   =====*/

/*#####
   ##          ##
   ## Knoten  ##
   ##          ##
   #####*/

/*-----
   Objekt-Typ für Knoten erzeugen
   -----*/
CREATE OR REPLACE TYPE knoten_objtyp
AS OBJECT (
  id.Knoten NUMBER(10,0),
  x NUMBER(10,3),
  y NUMBER(10,3),
  z NUMBER(10,3)
)NOT FINAL
/

/*-----
   Objekt-Tabelle für Knoten erzeugen
   -----*/
CREATE TABLE knoten_objtab
```

```

OF knoten_objtyp(
  CONSTRAINT knoten_idx PRIMARY KEY(id_Knoten)
)
OBJECT IDENTIFIER IS PRIMARY KEY
/

```

```

/*#####
##          ##
##  Kanten  ##
##          ##
#####*/

```

```

/*-----
  Objekt-Typ für Kanten erzeugen
-----*/

```

```

CREATE OR REPLACE TYPE kante_objtyp
AS OBJECT (
  id_Kante    NUMBER(10,0),
  anfang_ref  REF knoten_objtyp,
  ende_ref    REF knoten_objtyp,
  is_Fold     NUMBER(1,0)
)NOT FINAL
/

```

```

/*-----
  Objekt-Tabelle für Kanten erzeugen
-----*/

```

```

CREATE TABLE kante_objtab
OF kante_objtyp(
  CONSTRAINT kante_idx
    PRIMARY KEY(id_Kante),
  FOREIGN KEY (anfang_Ref)
    REFERENCES knoten_objtab
    ON DELETE CASCADE,
  FOREIGN KEY (ende_Ref)
    REFERENCES knoten_objtab
    ON DELETE CASCADE
)
OBJECT IDENTIFIER IS PRIMARY KEY
/

```

```

/*#####
##          ##
##  Photo-Textur  ##
##          ##
#####*/

```

```

/*-----

```

```

    Varray-Typ für Georeferenzierung erzeugen
    _____ */
CREATE OR REPLACE TYPE georef_vartyp
AS VARRAY(6)  --pelB, pelH, xLiO, yLiO, rotX, rotY
OF NUMBER(10,3)
/

/* _____
    Objekt-Typ für Photo-Textur erzeugen
    _____ */
CREATE OR REPLACE TYPE image_objtyp
AS OBJECT (
    image_Id          NUMBER(10,0),
    bitmap            ordsys.ordimage,
    original_Image_Ref REF image_objtyp,
    georef_var        georef_vartyp
)NOT FINAL
/

/* _____
    Objekt-Tabelle für Photo-Texturen erzeugen
    _____ */
CREATE TABLE image_objtab
OF image_objtyp(
    CONSTRAINT image_idx
        PRIMARY KEY (image_ID)
)
OBJECT IDENTIFIER IS PRIMARY KEY
/

/* #####
    ##           ##
    ##  Farbe   ##
    ##           ##
    #####*/

/* _____
    Objekt-Typ für Farbe erzeugen
    _____ */
CREATE OR REPLACE TYPE color_objtyp
AS OBJECT (
    color_Id NUMBER(10,0),
    rot      NUMBER(3,2),
    gruen    NUMBER(3,2),
    blau     NUMBER(3,2)
)NOT FINAL
/

```

```

/*-----
  Objekt-Tabelle für Farbe erzeugen
-----*/
CREATE TABLE color_objtab
OF color_objtyp(
  CONSTRAINT color_idx
    PRIMARY KEY (color_Id)
)
OBJECT IDENTIFIER IS PRIMARY KEY
/

/*#####
##          ##
##  Material  ##
##          ##
#####*/

/*-----
  Objekt-Typ für Material erzeugen
-----*/
CREATE OR REPLACE TYPE material_objtyp
AS OBJECT (
  material_Id  NUMBER(10,0),
  color_Ref    REF color_objtyp ,
  image_Ref    REF image_objtyp
)NOT FINAL
/

/*-----
  Objekt-Tabelle für Material erzeugen
-----*/
CREATE TABLE material_objtab
OF material_objtyp(
  CONSTRAINT material_idx
    PRIMARY KEY (material_ID),
  FOREIGN KEY (color_Ref)
    REFERENCES color_objtab
    ON DELETE CASCADE,
  FOREIGN KEY (image_Ref)
    REFERENCES image_objtab
    ON DELETE CASCADE
)
OBJECT IDENTIFIER IS PRIMARY KEY
/

```

```

/*#####*/
##      ##
##   Dreiecke   ##
##      ##
#####*/

/*-----
   Objekt-Typ für Masche erzeugen
-----*/
CREATE OR REPLACE TYPE masche_objtyp
AS OBJECT(
  masche_id NUMBER(10,0)
)NOT FINAL
/

/*-----
   Objekt-Tabelle für Masche erzeugen
-----*/
CREATE TABLE masche_objtab
OF masche_objtyp
/

/*-----
   Objekt-Typ für Dreiecke erzeugen
-----*/
CREATE OR REPLACE TYPE dreieck_objtyp
AS OBJECT (
  dreieck_Id      NUMBER(10,0),
  kante1_Ref     REF kante_objtyp ,
  kante2_Ref     REF kante_objtyp ,
  kante3_Ref     REF kante_objtyp ,
  material1_Ref  REF material_objtyp ,
  material2_Ref  REF material_objtyp ,
  material3_Ref  REF material_objtyp ,
  material4_Ref  REF material_objtyp
)NOT FINAL
/

/*-----
   Objekt-Tabelle für Dreiecke erzeugen
-----*/
CREATE TABLE dreieck_objtab
OF dreieck_objtyp(
  CONSTRAINT dreieck_idx
  PRIMARY KEY (dreieck_Id),
  FOREIGN KEY (kante1_Ref)
  REFERENCES kante_objtab
  ON DELETE CASCADE,

```

```

FOREIGN KEY ( kante2_Ref)
  REFERENCES kante_objtab
  ON DELETE CASCADE,
FOREIGN KEY ( kante3_Ref)
  REFERENCES kante_objtab
  ON DELETE CASCADE,
FOREIGN KEY ( material1_Ref)
  REFERENCES material_objtab
  ON DELETE CASCADE,
FOREIGN KEY ( material2_Ref)
  REFERENCES material_objtab
  ON DELETE CASCADE,
FOREIGN KEY ( material3_Ref)
  REFERENCES material_objtab
  ON DELETE CASCADE,
FOREIGN KEY ( material4_Ref)
  REFERENCES material_objtab
  ON DELETE CASCADE
)
OBJECT IDENTIFIER IS PRIMARY KEY
/

/* #####
   ##      ##
   ##  TIN  ##
   ##      ##
   #####*/

/* -----
   Objekt-Typ für Dreieckreferenz erzeugen
   ----- */
CREATE OR REPLACE TYPE dreieckref_objtyp
AS OBJECT(
  dreieck_ref  REF dreieck_objtyp
)NOT FINAL
/

/* -----
   NestedTable-Typ für TIN erzeugen
   ----- */
CREATE OR REPLACE TYPE dreieckaggregat_ntabtyp
AS TABLE OF dreieckref_objtyp
/

/* -----
   Objekt-Typ für TIN erzeugen
   ----- */
CREATE OR REPLACE TYPE tin_objtyp

```

```

AS OBJECT (
  tin_Id          NUMBER(10,0),
  dreieckaggreat_ntab  dreieckaggreat_ntabtyp
)
/

/*-----
   Objekt-Tabelle für TIN erzeugen
----- */

CREATE TABLE tin_objtab
OF tin_objtyp(
  CONSTRAINT tin_idx
    PRIMARY KEY (tin_ID)
)
OBJECT IDENTIFIER IS PRIMARY KEY
NESTED TABLE dreieckaggreat_ntab STORE AS dreieckaggreat_storetab
/

/*-----
   Ändern der REF SCOPE Anweisung für NestedTable
----- */

ALTER TABLE dreieckaggreat_storetab
ADD (SCOPE FOR (dreieck_ref) IS dreieck_objtab)
/

/*#####
   ##          ##
   ##  Flächeometrie  ##
   ##          ##
   #####*/

/*-----
   Objekt-Typ für Flächeometrie erzeugen
----- */

CREATE OR REPLACE TYPE flaecheometrie_objtyp
NOT FINAL
/

/*-----
   Objekt-Typ für Flächenaggreat erzeugen
----- */

CREATE OR REPLACE TYPE flaecheaggreat_objtyp
AS OBJECT(
  flaechegeom_ref  REF flaecheometrie_objtyp
)NOT FINAL
/

/*-----

```

```

    NestedTable-Typ für Flächenaggregat erzeugen
    _____ */
CREATE OR REPLACE TYPE flaechenaggregat_ntabtyp
AS TABLE OF flaechenaggregat_objtyp
/

/* _____
    Objekt-Typ für Flächengeometrie erzeugen
    _____ */
CREATE OR REPLACE TYPE flaechengeometrie_objtyp
AS OBJECT (
    id_Flaeche           NUMBER(10,0),
    flaechenaggregat_ntab flaechenaggregat_ntabtyp ,
    masche_ref          REF masche_objtyp ,
    tin_ref             REF tin_objtyp ,
    materialoben_Ref    REF material_objtyp ,
    materialunten_Ref   REF material_objtyp ,
    is_aggr             NUMBER
)NOT FINAL
/

/* _____
    Objekt-Tabelle für Flächengeometrie erzeugen
    _____ */
CREATE TABLE flaechengeometrie_objtab
OF flaechengeometrie_objtyp(
    CONSTRAINT flaechengeometrie_idx
        PRIMARY KEY (id_Flaeche),
    FOREIGN KEY (masche_Ref)
        REFERENCES masche_objtab
        ON DELETE CASCADE,
    FOREIGN KEY (tin_Ref)
        REFERENCES tin_objtab
        ON DELETE CASCADE,
    FOREIGN KEY (materialoben_Ref)
        REFERENCES material_objtab
        ON DELETE CASCADE,
    FOREIGN KEY (materialunten_Ref)
        REFERENCES material_objtab
        ON DELETE CASCADE
)
OBJECT IDENTIFIER IS PRIMARY KEY
NESTED TABLE flaechenaggregat_ntab STORE AS flaechenaggregat_storetab
/

/* _____
    Ändern der REF SCOPE Anweisung für NestedTable
    _____ */

```

```

ALTER TABLE flaechenaggregat_storetab
ADD (SCOPE FOR (flaechegeom_ref) IS flaechegeometrie_objtab)
/

/*#####*/
##
##   Volumen- Punkt- und Liniengeometrie   ##
##                                           ##
#####*/

/*-----*/
      Objekt-Typ für Volumengeometrie erzeugen
      ----- */
CREATE OR REPLACE TYPE volumengeometrie_objtyp
AS OBJECT(
  volumengeometrie_id NUMBER(10,0)
)NOT FINAL
/

/*-----*/
      Objekt-Tabelle für Volumengeometrie erzeugen
      ----- */
CREATE TABLE volumengeometrie_objtab
OF volumengeometrie_objtyp
/

/*-----*/
      Objekt-Typ für Liniengeometrie erzeugen
      ----- */
CREATE OR REPLACE TYPE liniengeometrie_objtyp
AS OBJECT(
  liniengeometrie_id NUMBER(10,0)
)NOT FINAL
/

/*-----*/
      Objekt-Tabelle für Liniengeometrie erzeugen
      ----- */
CREATE TABLE liniengeometrie_objtab
OF liniengeometrie_objtyp
/

/*-----*/
      Objekt-Typ für Punktgeometrie erzeugen
      ----- */
CREATE OR REPLACE TYPE punktgeometrie_objtyp
AS OBJECT(
  punktgeometrie_id NUMBER(10,0)

```

```

)NOT FINAL
/

/*-----
  Objekt-Tabelle für Punktgeometrie erzeugen
-----*/

CREATE TABLE punktgeometrie_objtab
OF punktgeometrie_objtyp
/

/*#####
  ##          ##
  ## Spatial  ##
  ##          ##
  #####*/

/*-----
  relationale Tabelle für Spatial Objekte
-----*/

CREATE TABLE geometrie_reltab(
  geometrie_Id      NUMBER(10,0),
  geometrie_Shape   mdsys.sdo_geometry,
  material_Ref      REF material_objtyp,
  volumen_Ref       REF volumengeometrie_objtyp,
  flaeche_Ref       REF flaechegeometrie_objtyp,
  dreieck_Ref       REF dreieck_objtyp,
  linie_Ref         REF liniengeometrie_objtyp,
  punkt_Ref         REF punktgeometrie_objtyp,
  CONSTRAINT geometrie_idx
    PRIMARY KEY (geometrie_Id),
  FOREIGN KEY (material_Ref)
    REFERENCES material_objtab
    ON DELETE CASCADE,
  FOREIGN KEY (volumen_Ref)
    REFERENCES volumengeometrie_objtab
    ON DELETE CASCADE,
  FOREIGN KEY (flaeche_Ref)
    REFERENCES flaechegeometrie_objtab
    ON DELETE CASCADE,
  FOREIGN KEY (dreieck_Ref)
    REFERENCES dreieck_objtab
    ON DELETE CASCADE,
  FOREIGN KEY (linie_Ref)
    REFERENCES liniengeometrie_objtab
    ON DELETE CASCADE,
  FOREIGN KEY (punkt_Ref)
    REFERENCES punktgeometrie_objtab
    ON DELETE CASCADE

```

```

)
/

/*-----
   Metadata für Spatial-Spalte der Dreiecke definieren
-----*/
INSERT INTO user_sdo_geom_metadata
VALUES(
  'geometrie_reltab',
  'geometrie_Shape',
  mdsys.sdo_dim_array(
    mdsys.sdo_dim_element('X', 0, 10000000, 0.001),
    mdsys.sdo_dim_element('Y', 0, 10000000, 0.001),
    mdsys.sdo_dim_element('Z', 0, 10000, 0.001)
  ),
  NULL
)
/

/*-----
   einen Pseudo-Datensatz anlegen,
   um Spatial-Index erzeugen zu können
-----*/
INSERT INTO geometrie_reltab
VALUES(
  -1,
  mdsys.sdo_geometry(
    3003,
    NULL,
    NULL,
    mdsys.sdo_elem_info_array(1,1003,1),
    mdsys.sdo_ordinate_array(
      -1,0,0,
      -3,0,0,
      -3,-1,0,
      -1,0,0
    )
  ),
  NULL,
  NULL,
  NULL,
  NULL,
  NULL,
  NULL
)
/

/*-----

```

Spatial Index für Geometrien erzeugen

```
_____/
CREATE INDEX geometrie_idx
ON geometrie_reltab(geometrie_Shape)
INDEXTYPE IS mdsys.spatial_index
PARAMETERS (
  'sdo_indx_dims = 3'
)
/

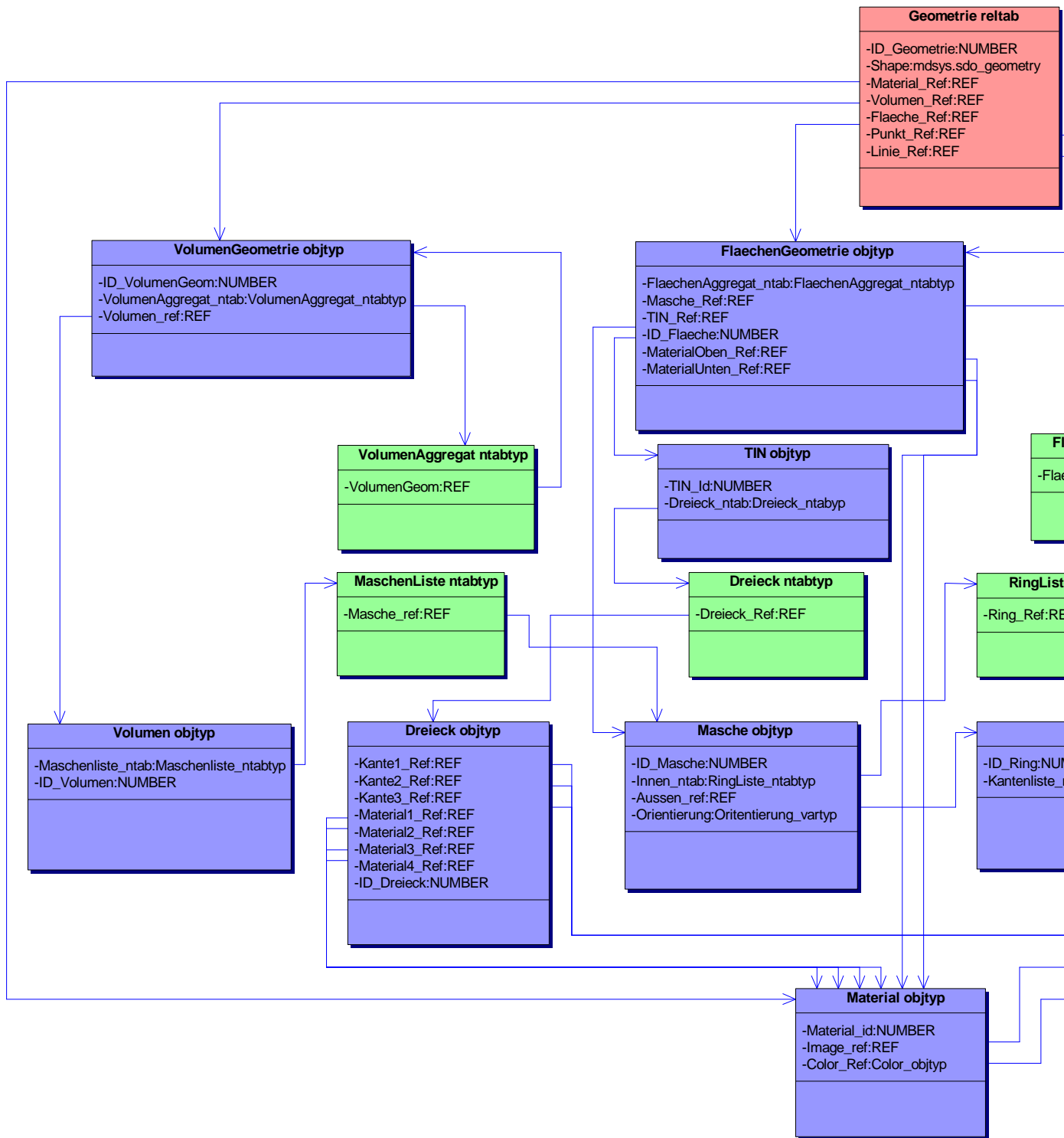
/*_____
  Pseudodatensatz löschen
_____*/
DELETE geometrie_reltab
WHERE geometrie_Id = -1
/
```

B.2 SQL-Befehle zum Löschen des DB-Schemas

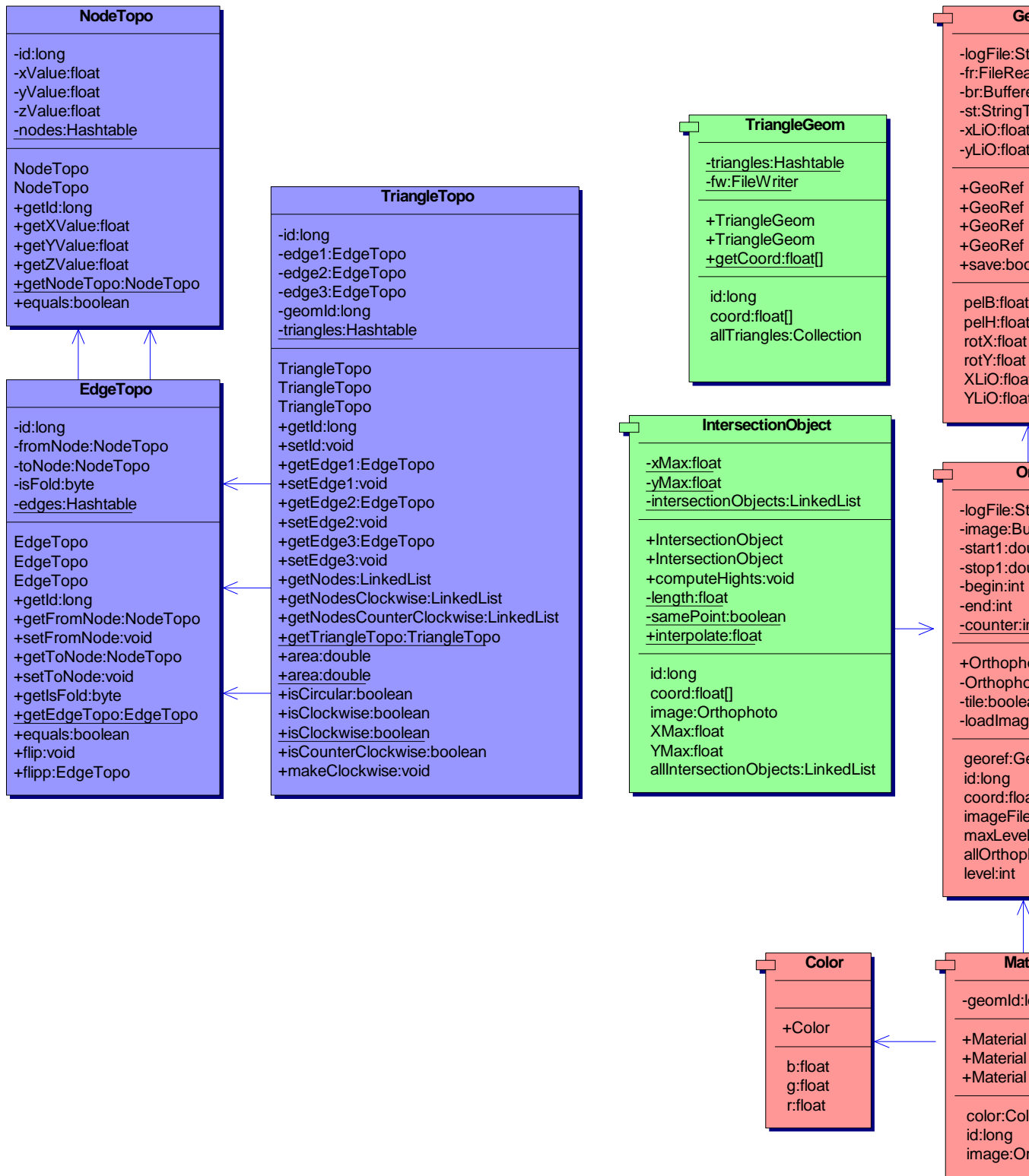
Wird das DB-Schema nur zu Testzwecken erzeugt, so können alle Objekttypen und -tabellen, Tabellentypen und Varray mit diesen Befehlen wieder gelöscht werden. Dazu muss der gleiche Benutzer an der DB angemeldet sein, der das Schema erzeugt hat!

```
DROP TABLE geometrie_reltab ;
DROP TYPE flaechenaggregat_ntabtyp FORCE;
DROP TYPE flaechenaggregat_objtyp FORCE;
DROP TABLE volumengeometrie_objtab;
DROP TYPE volumengeometrie_objtyp FORCE ;
DROP TABLE flaechegeometrie_objtab;
DROP TYPE flaechegeometrie_objtyp FORCE ;
DROP TABLE liniengeometrie_objtab;
DROP TYPE liniengeometrie_objtyp FORCE ;
DROP TABLE punktgeometrie_objtab;
DROP TYPE punktgeometrie_objtyp FORCE ;
DROP TABLE tin_objtab;
DROP TYPE tin_objtyp FORCE ;
DROP TYPE dreieckaggregat_ntabtyp FORCE;
DROP TYPE dreieckref_objtyp FORCE;
DROP TABLE dreieck_objtab;
DROP TYPE dreieck_objtyp FORCE;
DROP TABLE masche_objtab;
DROP TYPE masche_objtyp FORCE;
DROP TABLE kante_objtab ;
DROP TYPE kante_objtyp FORCE;
DROP TABLE knoten_objtab ;
DROP TYPE knoten_objtyp FORCE;
DROP TABLE material_objtab ;
DROP TYPE material_objtyp FORCE;
DROP TABLE image_objtab ;
DROP TYPE image_objtyp FORCE;
DROP TYPE georef_vartyp FORCE;
DROP TABLE color_objtab ;
DROP TYPE color_objtyp FORCE;
```

B.3 Grafische Übersicht des DB-Schemas



B.4 Klassendiagramm der Schnittstellen



Anhang C

PL/SQL

PL/SQL Skript zum Erzeugen der Referenzen zwischen Dreiecken und Orthophotos.

DECLARE

```
-- Werte, die aus dem Cursor gelesen werden:  
dreieck_id_tmp dreieck_objtab.dreieck_id%TYPE;  
mat_id_tmp material_objtab.material_id%TYPE;
```

```
-- Prüfvariablen:  
-- ID des aktuellen Dreiecks  
id_aktuell dreieck_objtab.dreieck_id%TYPE := 0;  
-- Anzahl der bereits bearbeiteten Mat. des Dreiecks  
anz_mat BINARY_INTEGER := -1;
```

CURSOR dreieck_id_cur IS

```
SELECT t.dreieck_id  
FROM  
  (  
    SELECT tr1.dreieck_Id, ge1.geometrie_shape  
    FROM dreieck_objtab tr1, geometrie_reltab ge1  
    WHERE tr1.dreieck_Id = ge1.dreieck_Id_ref  
  ) t,  
  (  
    SELECT ma2.material_Id, ge2.geometrie_shape  
    FROM material_objtab ma2, geometrie_reltab ge2  
    WHERE ma2.material_Id = ge2.material_Id_ref  
  ) m  
WHERE SDO_GEOM.RELATE(  
  t.geometrie_shape, 'ANYINTERACT', m.geometrie_shape, 0.001) = 'TRUE';
```

CURSOR mat_id_cur IS

```
SELECT m.material_Id  
FROM  
  (  
    
```

```

SELECT tr1.dreieck_Id , ge1.geometrie_shape
FROM dreieck_objtab tr1 , geometrie_reltab ge1
WHERE tr1.dreieck_Id = ge1.dreieck_Id_ref
) t,
(
SELECT ma2.material_Id , ge2.geometrie_shape
FROM material_objtab ma2, geometrie_reltab ge2
WHERE ma2.material_Id = ge2.material_Id_ref
) m
WHERE SDO_GEOM.RELATE(
t.geometrie_shape , 'ANYINTERACT' , m.geometrie_shape , 0.005) = 'TRUE';

```

BEGIN

```

OPEN dreieck_id_cur;
OPEN mat_id_cur;

```

LOOP

```

FETCH dreieck_id_cur INTO dreieck_id_tmp;
FETCH mat_id_cur INTO mat_id_tmp;

IF id_aktuell = -1 THEN --erster Datensatz
id_aktuell := dreieck_id_tmp; --Dreieck wird aktuelles Dreieck
anz_mat := 1; --erster Datensatz liegt vor
UPDATE dreieck_objtab
SET
material1_ref =
(SELECT REF(m1)
FROM material_objtab m1
WHERE material_id = mat_id_tmp)
WHERE dreieck_Id = dreieck_id_tmp;
dreieck_id_tmp := NULL; --falls im nächsten Schleifendurchlauf
mat_id_tmp := NULL; --nicht überschrieben
ELSIF id_aktuell = dreieck_id_tmp THEN --Dreieck wurde schon bearbeitet
IF anz_mat = 1 THEN --zweiter Datensatz liegt vor
anz_mat := anz_mat + 1;
UPDATE dreieck_objtab
SET
material2_ref =
(SELECT REF(m2)
FROM material_objtab m2
WHERE material_id = mat_id_tmp)
WHERE dreieck_Id = dreieck_id_tmp;
dreieck_id_tmp := NULL; --falls im nächsten Schleifendurchlauf
mat_id_tmp := NULL; --nicht überschrieben
ELSIF anz_mat = 2 THEN --dritter Datensatz liegt vor
anz_mat := anz_mat + 1;
UPDATE dreieck_objtab

```

```

SET
  material3_ref =
    (SELECT REF(m3)
     FROM material_objtab m3
     WHERE material_id = mat_id_tmp)
WHERE dreieck_Id = dreieck_id_tmp;
dreieck_id_tmp := NULL;  --falls im nächsten Schleifendurchlauf
mat_id_tmp := NULL;  --nicht überschrieben
ELSIF anz_mat = 3 THEN  --vierter Datendatz liegt vor
UPDATE dreieck_objtab
SET
  material4_ref =
    (SELECT REF(m4)
     FROM material_objtab m4
     WHERE material_id = mat_id_tmp)
WHERE dreieck_Id = dreieck_id_tmp;
dreieck_id_tmp := NULL;  --falls im nächsten Schleifendurchlauf
mat_id_tmp := NULL;  --nicht überschrieben
END IF;
ELSE  --neues Dreieck (id_aktuell != dreieck_id_tmp)
id_aktuell := dreieck_id_tmp;
anz_mat := 1;
UPDATE dreieck_objtab
SET
  material1_ref =
    (SELECT REF(m5)
     FROM material_objtab m5
     WHERE material_id = mat_id_tmp)
WHERE dreieck_Id = dreieck_id_tmp;
dreieck_id_tmp := NULL;  --falls im nächsten Schleifendurchlauf
mat_id_tmp := NULL;  --nicht überschrieben
END IF;

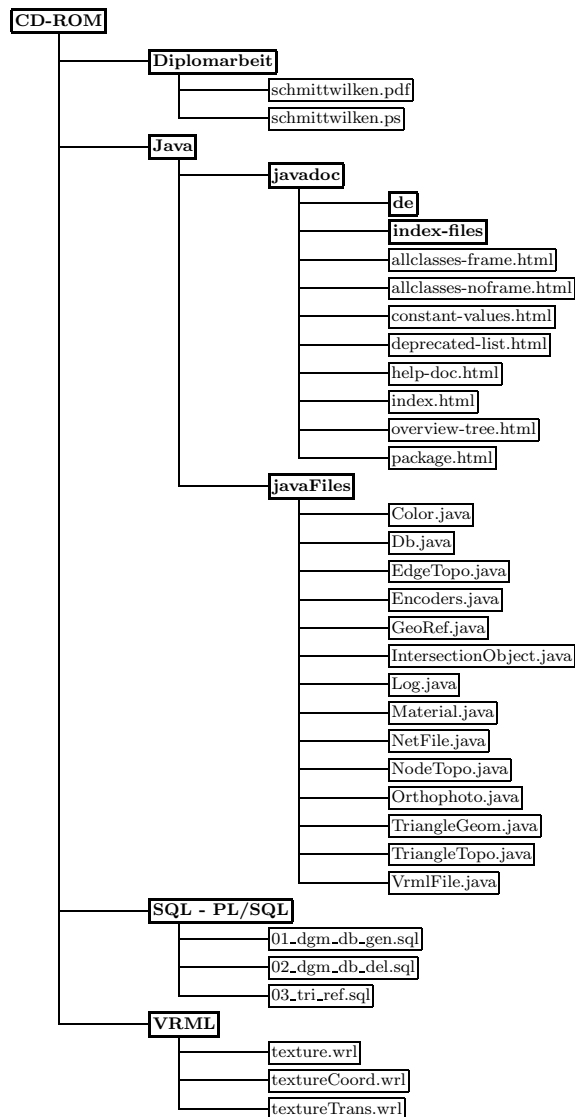
EXIT WHEN dreieck_id_cur%NOTFOUND OR mat_id_cur%NOTFOUND;
END LOOP;

CLOSE dreieck_id_cur;
CLOSE mat_id_cur;
END;
/

```


Anhang D

Inhaltsverzeichnis der CD-ROM



Literaturverzeichnis

- [ASB03] ARENS, CĂLIN, JANTIEN STOTER und ERNST VON BIRON: *Modelling 3D Spatial Objects in a Geo-DBMS Using a 3D Primitive*. In: *Proceedings of the 6th AGILE Dates – Lieu*, Apr. 2003. URL (zuletzt besucht: 24.03.2004): <http://www.gdmc.nl/stoter/publicaties/agile2003.zip>.
- [Cie03] CIESLIK, BERNHARD: *Hamburg in der dritten Dimension*. zfv - Zeitschrift für Geodäsie, Geoinformation und Landmanagement, 128(4):254–260, 2003.
- [EF91] EGENHOFER, M. und R.D. FRANZOSA: *Point-Set Topological Spatial Relations*. International Journal of Geographical Information Systems, 5(2):161–174, 1991. URL (zuletzt besucht: 24.03.2004): <http://www.spatial.maine.edu/~max/pointset.pdf>.
- [FDFH95] FOLEY, J., A. VAN DAM, S. FEINER und J. HUGHES: *Computer Graphics: Principles and Practice*. Addison Wesley, 1995.
- [Gut84] GUTTMAN, A.: *R-trees: a dynamic index structure for spatial searching*. In: *Proceedings of the 1984 ACM SIGMOD Conference*, Seiten 47–54, 1984.
- [JRH⁺04] JECKLE, MARIO, CHRIS RUPP, JÜRGEN HAHN, BARBARA ZENGLER und STEFAN QUEINS: *UML2 glasklar*. Hanser Verlag, 2004. URL (zuletzt besucht: 24.03.2004): <http://www.uml-glasklar.de>.
- [KG03] KOLBE, THOMAS H. und GERD GRÖGER: *Towards Unified 3D City Models*. In: *Proceedings of the ISPRS Comm. IV Joint Workshop on „Challenges in Geospatial Analysis, Integration and Visualization II“*, Sep. 2003. URL (zuletzt besucht: 12.03.2004): http://www.ikg.uni-bonn.de/Kolbe_home/publications/CGAIV2003_Kolbe_Groeger.pdf.
- [Len03] LENK, ULRICH: *2.5D-Diskontinuitäten in der Geländemodellierung und spezielle Aspekte bei Delaunay-Triangulationen*. zfv - Zeitschrift für Geodäsie, Geoinformation und Landmanagement, 128(6):357–365, 2003.
- [LH02] LENK, ULRICH und CHRISTIAN HEIPKE: *Ein 2.5D-GIS-Datenmodell durch Integration von DGM und DSM mittels Triangulationen - theoretischer und praktischer Vergleich von Algorithmen und ihrer Ergebnisse*. Photogrammetrie, Fernerkundung, Geoinformation, Seiten 205–216, Mrz. 2002. URL

- (zuletzt besucht: 12.03.2004): <http://www.ipi.uni-hannover.de/html/publikationen/2002/heipke/pfg01.htm>.
- [LN02] LOCKMAN, DAVID und DEBES NORBERT: *Oracle 9i Datenbankentwicklung in 21 Tagen*. Markt+Technik Verlag, 2002.
- [Mis91] MISGELD, W.: *Oracle für Profis*. Hanser Verlag, 1991.
- [Mol92] MOLENAAR, MARTIEN: *A Topology for 3D Vector Maps*. ITC Journal, 1:25–33, 1992.
- [Obj03] OBJECT MANAGEMENT GROUP: *OMG Unified Modeling Language Specification, Version 1.5*, 2003. URL (zuletzt besucht: 24.03.2004): <http://www.omg.org/docs/formal/03-03-01.pdf>.
- [OBSC99] OKABE, ATSUYUKI, BARRY BOOTS, KOKICHI SUGIHARA und SUNG NOK CHIU: *Spatial Tessellations*. Wiley, 1999. <http://okabe.t.u-tokyo.ac.jp/okabelab/Voronoi/index.html>.
- [Ope99] OPENGIS CONSORTIUM: *OpenGIS® Simple Features Specification For SQL, Revision 1.1*, 1999. URL (zuletzt besucht: 24.03.2004): <http://www.opengis.org/docs/99-049.pdf>.
- [Ora01a] ORACLE COOPERATION: *Oracle® interMedia - User's Guide and Reference, Release 9.0.1*, 2001. URL (zuletzt besucht: 24.03.2004): http://download-uk.oracle.com/docs/pdf/A88786_01.pdf.
- [Ora01b] ORACLE COOPERATION: *PL/SQL - User's Guide and Reference, Release 9.0.1*, 2001. URL (zuletzt besucht: 24.03.2004): http://download-uk.oracle.com/docs/cd/A91202_01/901_doc/appdev.901/a89856.pdf.
- [Ora02a] ORACLE COOPERATION: *Oracle® 9i - Application Developer's Guide - Object-Relational Features, Release 2(9.2)*, 2002. URL (zuletzt besucht: 24.03.2004): http://download-uk.oracle.com/docs/cd/B10501_01/appdev.920/a96594.pdf.
- [Ora02b] ORACLE COOPERATION: *Oracle® 9i - Javadoc for JDBC Drivers, Release 9.2.0.3*, 2002. URL (zuletzt besucht: 24.03.2004): http://download.oracle.com/otn/utilities_drivers/jdbc/9203/javadoc.tar.
- [Ora02c] ORACLE COOPERATION: *Oracle® 9i - JDBC Developer's Guide and Reference, Release 2(9.2)*, 2002. URL (zuletzt besucht: 24.03.2004): http://download-uk.oracle.com/docs/cd/B10501_01/java.920/a96654.pdf.
- [Ora02d] ORACLE COOPERATION: *Oracle® 9i - SQL Reference, Release 2(9.2)*, 2002. URL (zuletzt besucht: 24.03.2004): http://download-uk.oracle.com/docs/cd/B10501_01/server.920/a96540.pdf.

- [Ora02e] ORACLE COOPERATION: *Oracle©Spatial - User's Guide and Reference, Release 9.2*, 2002. URL (zuletzt besucht: 24.03.2004): http://download-uk.oracle.com/docs/cd/B10501_01/appdev.920/a96630.pdf.
- [OSQZ02] OOSTEROM, PETER VAN, JANTIEN STOTER, WILKO QUAK und SISI ZLATANOVA: *The Balance Between Geometry and Topology*. In: RICHARDSON, DIANNE und PETER VAN OOSTEROM (Herausgeber): *Advances in Spatial Data Handling, 10th International Symposium on Spatial Data Handling*, Seiten 209–224. Springer-Verlag, 2002.
- [PB96] POMBERGER, GUSTAV und GÜNTHER BLASCHEK: *Software Engineering - Prototyping und objektorientierte Software-Entwicklung*. Hanser Verlag, 1996.
- [PFK⁺02] PLÜMER, LUTZ, WOLFGANG FÖRSTNER, THOMAS H. KOLBE, ANSGAR BRUNN und GERD GRÖGER: *Konzeption und Dokumentation eines Verfahrens zur Ableitung von 3D-Stadtmodellen aus der Regengelderfassung (Versiegelungskartierung)*. unveröffentlicht, 2002.
- [QST03] QUAK, WILKO, JANTIEN STOTER und THEO TIJSSEN: *Topology in spatial DBMSs*. In: *Digital Earth, Brno, Czech Republic*, Sep. 2003.
- [Reu03] REUTER, M.: *Implementierung eines 3D-Stadtmodells in eine objektrelationale Datenbank am Beispiel von Oracle Spatial*. Diplomarbeit, Rheinische Friedrich-Wilhelms-Universität Bonn, Jul. 2003.
- [SZ03] STOTER, JANTIEN und SIYKA ZLATANOVA: *Visualisation and Editing of 3D Objects Organised in a DBMS*. In: *Proceedings of EUROSDR workshop: Rendering and visualisation*, Jan. 2003. URL (zuletzt besucht: 12.03.2004): http://www.gdmc.nl/stoter/publicaties/st_zl.zip.
- [WD97] WEINHAUS, F.M. und V. DEVARAJAN: *Texture Mapping 3D Models of Real-World Scenes*. ACM Computing Surveys, 29(4), Dez. 1997.
- [Web97] WEB3D CONSORTIUM: *The Virtual Reality Modeling Language*, 1997. URL (zuletzt besucht: 24.03.2004): <http://www.web3d.org/technicalinfo/specifications/vrml97/vrml97specification.pdf>.
- [ZRP02] ZLATANOVA, S., A.A. RAHMAN und M. PILOUK: *3D GIS: current status and perspectives*. In: *Proceedings of the Joint Conference on Geo-spatial theory*, Seite 6p, Jul. 2002. URL (zuletzt besucht: 12.03.2004): http://www.gdmc.nl/zlatanova/thesis/html/refer/ps/SZ_AR_MP02.pdf.
- [ZS02] ZIPF, A. und A. SCHILLING: *Dynamische Generierung von VR-Stadtmodellen aus 2D- und 3D-Geodaten für Tourenanimationen*. GIS - Geo-Informationssysteme. Zeitschrift für raumbezogene Information und Entscheidungen, Seiten 24–30, Jun. 2002. URL (zuletzt besucht: 12.03.2004): <http://dookie.geoinform.fh-mainz.de/~zipf/Zipf-Schilling-2D-3D-GIS-2002-last.pdf>.