Fachpraktikum
Erweiterbare Datenbanksysteme
Wintersemester 2015/16
Aufgabe 2 der Phase 2

3D Algebra - Version 2

Ralf Hartmut Güting, Thomas Behr, Fabio Valdés, Holger Helmut Hennings

19.11.2015

Lehrgebiet Datenbanksysteme für neue Anwendungen
Fakultät für Mathematik und Informatik

FernUniversität in Hagen

1 Motivation

SECONDO unterstützt neben vielen anderen Datentypen auch geometrische Datentypen oder raum-zeitliche Datentypen. Dabei befanden sich die Geometrien bis zum letzten Jahr stets im zweidimensionalen Raum. Beim letzten Durchlauf des Fachpraktikums wurde eine Algebra für 3-dimensionale Objekte sowie ein passender Viewer implementiert. Die Algebra stellt eine Reihe von Typen sowie einige Operationen auf diesen Typen bereit. Leider arbeitet die Algebra nicht sonderlich robust. Daher soll in diesem Semester eine robuste Variante dieser Algebra entstehen. Da der Viewer der vorherigen Version gut funktioniert, sollten die Nested-List-Darstellungen der `3DAlgebra` für die Typen verwendet werden. Die Strukturen auf der C++-Seite sind durch neue Strukturen zu ersetzen. Um Rundungsfehler zu vermeiden, sollen Berechnungen mit Hilfe der Multi-Precision-Bibliothek durchgeführt werden. Diese stellt Zahlen beliebiger Genauigkeit sowie Operationen auf solchen Zahlendarstellungen zur Verfügung. Nach Ende der Berechnungen sollen die so entstandenen Ergebnisse in eine Standard-Darstellung, die `double` zur Darstellung einer Zahl verwendet, gebracht werden.

2 Datentypen

In diesem Abschnitt beschreiben wir die zu implementierenden Datentypen detailliert.

2.1 Geometrische Datentypen

Alle geometrischen Datentypen sind von der Klasse `StandardSpatialAttribute<3>` abgeleitet, die in der Datei `RectangleAlgebra.h` im Verzeichnis `Algebras/Rectangle` definiert ist. Damit sind diese Klassen automatisch von der Klasse `Attribute` abgeleitet und müssen deren Anforderungen genügen. Neben den für die Klasse `Attribute` notwendigen Funktionen sind noch weitere Funktionen zu implementieren, die in der Klassendeklaration von `StandardSpatialAttribute` zu finden sind. Im Einzelnen sind dies:

```
virtual const Rectangle<dim> BoundingBox(const Geoid* geoid = 0) const;
virtual double Distance(const Rectangle<dim>& rect,
                        const Geoid* geoid=0) const;

virtual bool Intersects(const Rectangle<dim>& rect,
                        const Geoid* geoid=0 ) const;

virtual bool IsEmpty() const;
```

Die Funktion `BoundingBox` berechnet den kleinsten achsenparallelen Quader, in den das komplette Objekt hineinpasst. Für mengenwertige Typen ist es sinnvoll, diese Box als Member vorzuhalten. Der `Geoid`-Parameter kann ignoriert werden. Die Funktion `Distance` berechnet die Euklidische Distanz der Geometrie zu einem gegebenen Quader. Wieder darf der `Geoid`-Parameter ignoriert werden. Ist das geometrische Objekt leer oder undefiniert, so wird -1 zurückgeliefert. Die Funktion `Intersects` prüft, ob das Objekt einen gegebenen Quader schneidet. Die Funktionen `Distance` und `Intersects` werden zur Vereinfachung lediglich auf den Bounding Boxen der Objekte durchgeführt. Die Funktion `IsEmpty` liefert `true`, wenn das Objekt undefiniert ist oder die leere Menge darstellt.

Die geometrischen Datentypen werden den Kinds `DATA` und `SPATIAL3D` zugeordnet. Durch diese Vorgehensweise können z.B. *Spatial Joins* auf Mengen solcher Objekte effizient durchgeführt werden.

2.1.1 *point3d*

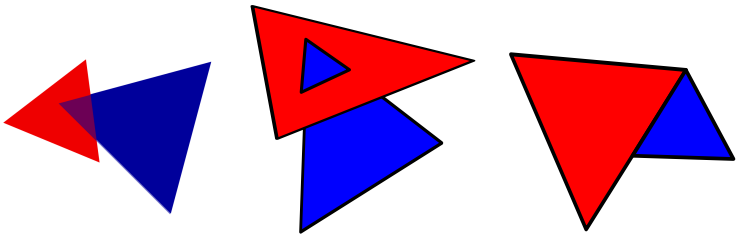
Dieser Typ repräsentiert einen Punkt im $\mathbb{R}^3$.

2.1.2 *surface3d*

Dieser Typ dient der Darstellung von 3D-Oberflächen, die durch Mengen von Dreiecken dargestellt sind. Oberflächen müssen nicht zusammenhängend sein, d.h. sie können aus verschiedenen Komponenten be-

stehen. An die Dreiecksmenge werden bestimmte Anforderungen gestellt. So dürfen zwei Dreiecke keine gemeinsame Fläche besitzen. Falls zwei Dreiecke ein gemeinsames Segment besitzen, so ist dieses Segment zwingend eine vollständige Außenkante beider Dreiecke. Ggf. müssen Dreiecke geteilt werden, um diese Anforderungen zu erfüllen.

Somit sind z.B. folgende Konfigurationen nicht erlaubt:



(a) (b) (c)

Konfiguration (a) ist nicht erlaubt, da sich die Dreiecke überlappen. In Konfiguration (b) gibt es ein gemeinsames Segment, welches keine Kante beider Dreiecke darstellt. In Konfiguration (c) überdeckt das gemeinsame Segment nicht die komplette Kante des einen Dreiecks.

2.1.3 *volume3d*

Mit diesem Typ werden Körper im $\mathbb{R}^3$ beschrieben. Wie schon Oberflächen, werden die Objekte durch Dreiecksmengen beschrieben. Für die Unterstützung einiger Operatoren muss hierbei für ein einzelnes Dreieck erkennbar sein auf welcher Seite das Äußere und auf welcher Seite das Innere des dargestellten Objekts liegt. Dies kann durch einen zusätzlichen Richtungsvektor am Dreieck oder (speichersparender) durch eine vorgegebene Reihenfolge der Punkte im Dreieck realisiert werden. Neben den Anforderungen an die Dreiecksmenge für Oberflächen muss die Dreiecksmenge eines Körpers geschlossen sein, d.h. jedes Dreieck hat an jeder Kante ein benachbartes Dreieck, das die dargestellte Oberfläche erweitert. Durch ein solches Objekt können auch mehrere Körper dargestellt werden. Dann besteht dieses Objekt aus mehreren „wasserdichten“ Oberflächen. Es ist jedoch nicht erlaubt, dass sich Teile einer Oberfläche innerhalb anderer Objektteile befinden.

2.2 Hilfsdatentypen

Die folgenden Datentypen werden in Operationen verwendet. Diese müssen weder in Relationen gespeichert werden können noch muss es eine besondere Unterstützung für Joins geben.

2.2.1 *vector3d*

Wie dreidimensionale Punkte, besteht dieser Datentyp aus drei Koordinaten. Diese werden hier jedoch nicht als Punkt, sondern als Richtungsvektor aufgefasst.

2.2.2 *plane3d*

Dieser Typ stellt eine Ebene im $\mathbb{R}^3$ dar. Diese kann durch einen Punkt und zwei Richtungsvektoren, also neun reellwertige Koordinaten, beschrieben werden. Alternativ ist auch eine Darstellung durch einen Punkt und einen Normalenvektor möglich.

3 Operatoren

3.1 Info

3.1.1 size

Dieser Operator nimmt eine 3D-Geometrie und liefert die Anzahl der enthaltenen Elemente (Punkte oder Dreiecke) zurück.

3.1.2 bbox

Mit Hilfe des **bbox**-Operators wird der kleinste achsenparallele Quader ermittelt, der das Objekt vollständig enthält. Die Rückgabe ist vom Typ *rect3*, dessen dazugehörige C++-Klasse `Rectangle` in der Datei `RectangleAlgebra.h` definiert ist.

3.2 Import

Um größere Mengen von Objekten in einer Datenbank zu speichern, muss es möglich sein, Standardformate dieser Objekte einzulesen. Es gibt eine Vielzahl an Formaten, die der Darstellung dreidimensionaler Objekte dienen. Ein sehr einfaches Format ist das sogenannte STL (Surface Tesselation Language) Format, in dem 3D-Objekte praktischerweise durch Mengen von Dreiecken dargestellt werden. Der Operator **importSTL** erhält als Eingabe einen Text, der den Dateinamen einer STL-Datei darstellt und liefert ein Objekt vom Typ *volume3d*. Existiert die Datei nicht oder stellt der Inhalt der Datei kein 3D-Objekt dar, ist das Ergebnis ein undefiniertes *volume3d*. Die in den Dateien vorhandenen Dreiecksmengen können fehlerbehaftet sein. Dies schließt überlappende Dreiecke, Dreiecke im Inneren des Körpers, eine fehlerhafte Orientierung der Dreiecke (Außenseite) und nicht geschlossene Körper ein. Der Import-Operator versucht, solche Fehler zu korrigieren. Gelingt eine Korrektur nicht, ist das Ergebnis ein undefinierter Körper. Es werden folgende Schritte beim Import durchgeführt:

- Einlesen der Dreiecksmenge
- Prüfung und ggf. Korrektur der eingelesenen Dreiecksmenge

Der Korrekturprozess umfasst folgende Schritte:

- Teilung überlappender oder sich schneidender Dreiecke
- Entfernen von Duplikaten
- Schließen von Löchern
- Korrektur der Orientierung
- Entfernen von Dreiecken aus dem Objektinneren

STL-Dateien gibt es in zwei verschiedenen Formaten, die beide durch den Operator unterstützt werden. Das Format wird automatisch erkannt.

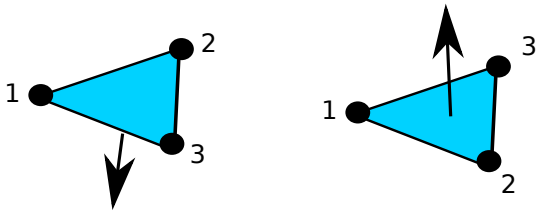
3.2.1 STL-ASCII-Format

STL-Dateien, die im ASCII-Format vorliegen, können in einem einfachen Texteditor angesehen und manipuliert werden. Solche Dateien haben den folgenden Aufbau:

```
solid <name>
  facet
    normal n1 n2 n3
    outer loop
      vertex x1 y1 z1
```

```
      vertex x2 y2 z2
      vertex x3 y3 z3
    endloop
  endfacet
  ...
endsolid
```

Hierbei stellen `solid`, `facet`, `normal`, `outer`, `loop`, `vertex`, `endloop`, `endfacet` und `endsolid` Schlüsselwörter dar. Leerzeichen und Zeilenumbrüche können an beliebigen Stellen zwischen den einzelnen Elementen auftreten. <name> ist der Name des Objekts, der beim Import ignoriert werden kann. Ein Objekt besteht aus einer Menge von `facet` Definitionen, die in der Praxis stets ein Dreieck darstellen. Hinter dem Schlüsselwort `normal` folgen 3 reelle Zahlen, die den Normalenvektor des Dreiecks angeben. Dieser steht senkrecht auf dem Dreieck, hat die Länge 1 und zeigt nach außen. Die Knotenreihenfolge ist gegen den Uhrzeigersinn, wenn man von außen auf das Objekt sieht. Die Information, wo sich außen befindet, ist also redundant gespeichert.



3.2.2 STL-Binärformat

Das Binärformat hat den Vorteil wesentlich kleiner als das ASCII-Format zu sein, jedoch ist das Format nicht menschenlesbar. Koordinaten werden als 32-Bit-Real-Zahl dargestellt (little endian). Eine solche Datei beginnt mit einem 80 Byte großen Header, dessen Inhalt ignoriert werden kann. Anschließend folgt eine vorzeichenlose Integer-Zahl (32 Bit), die die Anzahl der Dreiecke in der Datei angibt. Danach folgen die einzelnen Dreiecke. Jedes Dreieck besteht aus zwölf reellen Zahlen (je 32 Bit) gefolgt von einer vorzeichenlosen 16-Bit Integerzahl, die die Größe eines am Dreieck hängenden Attributs in Byte angibt. Im Normalfall ist diese Zahl einfach 0. Die ersten drei reellen Zahlen stellen den Normalenvektor des Dreiecks dar. Die restlichen neun Koordinaten sind jeweils die Koordinaten der drei Punkte.

3.3 Export

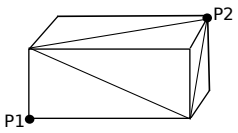
Möchte man in SECONDO generierte oder manipulierte Objekte in anderen Programmen weiterverarbeiten, ist es notwendig, die Objekte wieder zu exportieren. Die Algebra3D bietet hierzu den Operator **exportSTL** an, der einen Körper exportiert. Neben dem zu exportierenden Objekt erhält der **exportSTL**-Operator noch den Namen der Datei, in die exportiert wird (*text*), den Objektnamen (*string*), sowie ein boolesches Argument, das angibt, ob das binäre Format oder ASCII-Text für den Export verwendet wird. Der Operator liefert einen booleschen Wert zurück, der angibt, ob der Export erfolgreich war. Dieser kann z.B. fehlschlagen, wenn das Objekt undefiniert ist oder Dateioperationen nicht erfolgreich sind.

3.4 Erzeugen

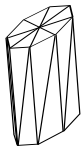
Die hier beschriebenen Operatoren dienen der Erzeugung einfacher Körper.

3.4.1 createCube

Mit diesem Operator wird ein achsenparaller Quader erzeugt. Als Eingaben dienen zwei Punkte, die zu einer der Raumdiagonalen gehören. Kann aus den Punkten kein Quader erzeugt werden (Gleichheit in mind. einer Dimension, oder mind. einer der Punkte ist undefiniert), ergibt sich ein undefinierter Körper.



3.4.2 createCylinder

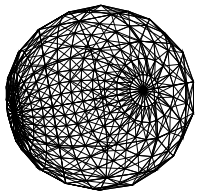


Der Operator **createCylinder** erzeugt einen Zylinder, dessen Hauptachse (Drehachse) parallel zur z-Achse liegt. Das erste Argument des Operators ist ein Punkt, der den Mittelpunkt der Grundplatte des Zylinders darstellt. Das zweite Argument (Typ *real*) gibt den Radius des Zylinders an. Das dritte Argument, das ebenfalls vom Typ *real* ist, steht für die Höhe des Zylinders. Das vierte und letzte Argument ist vom Typ *int* und gibt an, wieviele Eckpunkte zur Approximation des Kreises verwendet werden.

3.4.3 createCone

Mit Hilfe dieses Operators lässt sich ein spitzer Kegel konstruieren, der wie auch der Zylinder die gleiche Orientierung wie die z-Achse aufweist. Die Spitze des Kegels zeigt nach oben. Die Argumente entsprechen denen des Operators **createCylinder**.

3.4.4 createSphere



Der Operator **createSphere** erzeugt eine Kugel und erhält als Argumente den Mittelpunkt der Kugel (*point3d*), ihren Radius (*real*) sowie ein Argument, mit dem die Approximationsgüte bestimmt wird (*int*). Die Konstruktion kann folgendermaßen geschehen. Zunächst wird der Approximationsparameter verwendet, um eine Einheitskugel zu konstruieren. Diese wird dann in alle Richtungen mit dem Radius skaliert und anschließend in den gewünschten Mittelpunkt verschoben.

Die Konstruktion der Einheitskugel kann mit dem Approximationsparameter n folgendermaßen ablaufen:

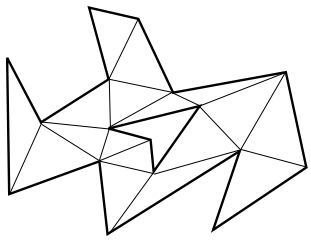
- Falls n keine gerade Zahl ist, wird n um 1 erhöht.
- Es wird an der Stelle (1,0,0) ein Punkt generiert.
- Dieser Punkt wird nun in $n/2$ Schritten um insgesamt 180 Grad um die z-Achse gedreht. Alle Punkte zusammen bilden die Approximation eines Halbkreises, dessen Endpunkte auf der x-Achse liegen.
- Diese Punkte werden nun in n Schritten insgesamt 360 Grad um die x-Achse gedreht. Jeweils vier Punkte (Nachbarn auf dem Halbkreis, letzte Drehung und aktuelle Drehung) werden durch zwei Dreiecke verbunden. (Achtung bei den Punkten auf der x-Achse gibt es nur drei Punkte und somit pro Drehung nur ein Dreieck.)
- Die dazugehörenden Normalenvektoren erhält man, indem man die Anfangspunkte jedes Rechtecks in die Mitte dieses Rechtecks dreht.

3.5 Konvertieren

Mit den folgenden Operationen werden 3D-Objekte aus anderen Objekttypen erzeugt.

3.5.1 region2surface

Der Operator **region2surface** erstellt aus einer Region eine Oberfläche, die auf der xy-Ebene liegt. Dazu ist es notwendig, die Region zu triangulieren. Hierbei ist zu beachten, dass Regionen in SECONDO aus verschiedenen Komponenten bestehen und Löcher besitzen können.



3.5.2 region2volume

Dieser Operator erstellt aus einer Region einen dreidimensionalen Körper. Als Eingaben dienen die Region selbst sowie eine reelle Zahl, die die Dicke des zu erzeugenden Körpers darstellt. Der Operator arbeitet wie folgt: Zunächst wird die Region trianguliert wie beim Operator **region2surface**. Dann werden Klone der Dreiecke um die vorgegebene Dicke in z-Richtung verschoben und die Dreiecke, die die Ränder der Region darstellen, durch weitere Dreiecke miteinander verbunden.

3.5.3 mregion2volume

Eine *moving region*, kurz *mregion*, ist eine Region, die im Laufe der Zeit ihre Position und/oder Form ändert. Fasst man die Zeit nun als dritte Dimension auf, ergibt sich ein dreidimensionaler Körper. Jede *mregion* besteht aus einer Menge von sogenannten *uregion*-Elementen, die eine einfache Veränderung der Region darstellen. Ein solches Element ist für ein bestimmtes Zeitintervall definiert und gibt den Zustand der Region am Anfang und am Ende dieses Zeitintervals an. Um hieraus einen Körper zu erzeugen, trianguliert man den Anfangs- und den Endzustand und verbindet die so erhaltenen Dreiecksmengen durch weitere Dreiecke. Das Gesamtobjekt ergibt sich aus der Vereinigung aller durch die *uregion* Werte erhaltenen Körper. Neben der *mregion* selbst erhält der Operator noch ein Argument vom Typ *real*, das die „Dicke“ eines Tages darstellt, d.h. das Objekt, das aus einer Unit, die über einen Zeitraum von zwei Tagen definiert ist, mit einem Wert von 0.25 erzeugt wird, hat eine Höhe von 0.5.

3.6 Transformieren

Die nun folgenden Operatoren wenden affine Transformationen auf geometrische Objekte an. Solche Transformationen erhalten die Lagebeziehungen zwischen den einzelnen Elementen. Alle Operationen funktionieren auf die gleiche Art und Weise. Jeder Punkt des Objekts wird mit einer Matrix multipiliert und durch das Ergebnis ersetzt. Die verschiedenen Operatoren unterscheiden sich nur durch das Zusammensetzen der Matrix. Ggf. müssen die Komponenten des Ergebnisses neu sortiert werden oder die Punkte innerhalb eines Dreiecks umsortiert werden.

Aus technischen Gründen verwendet man einen vierdimensionalen Punkt und eine 4×4-Matrix. Die vierte Koordinate eines Punkts erhält den Wert 1. Die Matrizen haben je nach gewünschter Operation verschiedene Inhalte. Sollen mehrere Operationen auf einem Punkt hintereinander ausgeführt werden, so multipiziert man nicht den Punkt mit jeder Matrix, sondern bildet zunächst das Matrizenprodukt aller Abbildungen und multipliziert den Punkt mit diesem Ergebnis. Die Matrizen der Grundoperationen haben folgende Inhalte:

Verschiebung um T_x, T_y, T_z

$$\begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Skalierung mit S_x, S_y, S_z

$$\begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

x-Rotation um ϕ

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

y-Rotation um ϕ

$$\begin{pmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

z-Rotation um ϕ

$$\begin{pmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation um eine Ursprungsgerade (geg. durch Einheitsvektor (n_1, n_2, n_3)) um ϕ

$$\begin{pmatrix} n_1^2(1-\cos\phi)+\cos\phi & n_1n_2(1-\cos\phi)-n_3\sin\phi & n_1n_3(1-\cos\phi)+n_2\sin\phi & 0 \\ n_2n_1(1-\cos\phi)+n_3\sin\phi & n_2^2(1-\cos\phi)+\cos\phi & n_2n_3(1-\cos\phi)-n_1\sin\phi & 0 \\ n_3n_1(1-\cos\phi)-n_2\sin\phi & n_3n_2(1-\cos\phi)+n_1\sin\phi & n_3^2(1-\cos\phi)+\cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3.6.1 rotate

Dieser Operator rotiert ein gegebenes Objekt um einen gegebenen Punkt um eine gegebene Achse und einen gegebenen Winkel. Als Eingabe dienen also:

- das zu drehende Objekt (*point3d*, *surface3d*, *volume3d*)
- der Punkt, um den gedreht wird (*point3d*)
- die Achse, um die rotiert wird (*vector3d*)
- der Drehwinkel (*real*)

Die folgenden Matrixoperationen führen zur gewünschten Rotationsmatrix:

- Verschiebung des Drehpunkts in den Ursprung
- Skalierung des Vektors auf Länge 1
- Rotation um die durch den skalierten Vektor gegebene Ursprungsachse
- Verschiebung in den Drehpunkt

3.6.2 mirror

Dieser Operator spiegelt ein geometrisches Objekt an einer beliebigen Ebene des Raums. Das Objekt und die Ebene bilden die Eingabe des Operators.

Die Matrix kann wie folgt zusammengesetzt werden:

- Verschiebung des Ebenenpunkts in den Ursprung
- Drehung der Ebene auf eine Hauptebene
- Spiegelung an dieser Achse (Entspricht einer Skalierung mit Wert -1 für diese und 1 für die anderen Achsen)
- Rückdrehung der Ebene
- Rückverschiebung in den Ebenenpunkt

3.6.3 translate

Der Operator **translate** verschiebt ein Objekt um einen gegebenen Vektor. Hier kann direkt die Translationsmatrix verwendet werden.

3.6.4 scaleDir

Diese Operation skaliert ein Object aus einem gegebenen Punkt heraus in eine bestimmte Richtung um einen Faktor. Folgende Matrixoperationen führen zur gewünschten Matrix:

- Verschiebung des Punkts in den Ursprung
- Skalierung entsprechend des Vektors, dessen Länge zunächst auf den Skalierungsfaktor angepasst wurde
- Verschiebung in den Punkt

3.6.5 scale

Diese Operation skaliert in alle drei Richtungen der Koordinatenachsen von einem gegebenen Punkt aus. Als Eingabe dienen die Geometrie, der Punkt sowie der Skalierungsfaktor. Die Matrix wird gebildet durch:

- Verschiebung des Punkts in den Ursprung
- Skalierung
- Rückverschiebung

3.7 Mengenoperationen

Mittels Mengenoperatoren lassen sich zwei 3D-Geometrien gleichen Typs kombinieren. Es sind die üblichen Mengenoperationen, d.h. Vereinigung, Schnitt und Differenz, verfügbar. Alle Operationen erzeugen gültige Objekte. Dazu ist es bei Punkten notwendig, Duplikate zu eliminieren, Überlappende oder sich schneidende Dreiecke von Oberflächen oder Körpern müssen geteilt werden und es dürfen nur diejenigen (ggf. geteilten) Dreiecke im Ergebnis vorhanden sein, die zur Repräsentation des Objekts notwendig sind.

3.7.1 Mengenoperationen auf Punktmengen

Da die Punkte intern geordnet sind, genügt für die Mengenoperationen ein paralleler Durchlauf durch beide Punktmengen. Somit wird eine lineare Laufzeit erreicht.

3.7.2 Mengenoperationen auf Oberflächen

Hier kann zunächst geprüft werden, ob sich die Bounding Boxen der beiden Objekte überhaupt überlappen. Bei disjunkten Boxen lassen sich die Operationen sehr leicht implementieren. Komplizierter wird es, wenn sich die Boxen beider Objekte überlappen, da dann die Möglichkeit besteht, dass sich auch die enthaltenen Dreiecke schneiden oder überlappen. Solche Dreiecke müssen zerlegt werden, um die Eigenschaften von Oberflächen zu gewährleisten. Da Oberflächen aus sehr vielen Dreiecken bestehen, ist es ungünstig, die Dreiecke paarweise auf mögliche Schnitte zu prüfen. Um dies effizient zu bewerkstelligen, kann man wie folgt vorgehen: Man fügt die Dreiecke eines Objekts in einen R-Baum ein. Da davon ausgegangen werden kann, dass ein einzelnes Objekt im Hauptspeicher gehalten werden kann, kann hier eine Hauptspeicherimplementierung eines R-Baums verwendet werden. Eine solche Implementierung findet sich in der Datei `MMRTree.h` im `include`-Verzeichnis von SECONDO. Für jedes Dreieck des zweiten beteiligten Objekts wird auf dem erstellten R-Baum eine Suche durchgeführt. Dafür bietet sich die Operation `find` der R-Baum-Klasse an, die einen Iterator liefert. Damit werden alle Dreiecke des ersten Objekts gefunden, deren Boxen Überlappungen mit der Box des Suchdreiecks besitzen. Diese stellen Kandidaten für echte Überlappungen dar. Im Falle von Schnitten oder Überlappungen, sind die beteiligten Dreiecke entsprechend zu teilen. Treten Überlappungen von Dreiecken auf, so ist darauf zu achten, dass der überlappende Teil nur einmal oder, im Falle der Differenz, gar nicht im Ergebnis auftaucht.

3.7.3 Mengenoperationen auf Körpern

Zunächst können die Dreiecke wieder wie bei den Oberflächen geteilt werden. Zusätzlich ergibt sich hier die Schwierigkeit der Auswahl der Dreiecke, die zum Endergebnis gehören. So können z.B. bei einer Vereinigung nicht einfach alle Dreiecke im Ergebnis auftauchen, da diese sich auch innerhalb des Körpers befinden können. Weiterhin müssen ggf. auch die Informationen über die Außenseite von Dreiecken angepasst werden.

3.8 Zerlegen

Oberflächen und Volumina können aus mehreren, nicht verbundenen Teilen bestehen, auf die ggf. einzeln zugegriffen werden soll. Der Operator **components** zerlegt ein Objekt in seine zusammenhängenden Komponenten. Er erhält eine 3D-Geometrie und liefert einen Strom solcher Geometrien, der die einzelnen Komponenten des Arguments enthält. Zwei Dreiecke gellten hierbei als verbunden, falls sich sich eine Kante teilen. Besteht beispielsweise ein *volume3d* aus zwei spitzen Kegeln, die sich nur an ihren Spitzen berühren, so wird dieses Objekt in zwei Teile zerlegt. Eine Möglichkeit der Realisierung ist es, jedem Dreieck eine Komponentennummer zuzuordnen. Um zusammenhängende Dreiecke zu finden, kann wiederum ein R-Baum verwendet werden.

3.9 Robustheit

In der Version 2 dieser Algebra soll besonderer Wert auf die Robustheit der einzelnen Operatoren gelegt werden. Speziell betroffen sind hier solche Operationen, die neue Dreiecke aus vorhandenen Dreiecken berechnen. Hier stellen vor allem Rundungsfehler ein großes Problem dar. Daher sollen solche Berechnungen auf dreidimensionalen Dreiecken unter der Verwendung der GNU Multiple Precision Library realisiert werden. Daher gibt es zwei Darstellungen der Dreiecke. Die Darstellung innerhalb der zu implementierenden Datentypen verwendet zur Darstellung von Zahlen den Typ `double`. Für Berechnungen wird eine Repräsentation von Dreiecken verwendet, die zur Darstellung einer Koordinate eine rationale Zahl beliebiger Genauigkeit verwendet. Diese Zahlendarstellung wird durch die Klasse `mpq_class` bereitgestellt. Neben der exakten Darstellung von Zahlen, sind hier auch einige mathematische Operatoren definiert, die verlustfreie Berechnungen durchführen. Eine Beschreibung der `gmp`-Bibliothek findet sich in `https://gmplib.org/gmp-man-6.0.0a.pdf`.

Nach der Berechnung des Ergebnisses, muss das im Hauptspeicher vorliegende Objekt wieder in seine Standarddarstellung gebracht werden. Hierbei kann das Problem entstehen, dass nicht alle Koordinaten als `double` darstellbar sind. Dadurch können durch die Konvertierung der Ecken ungültige Dreiecke

entstehen, d.h. Dreiecke können zu einem Punkt oder einer Linie degenerieren. Wird ein solches Dreieck erkannt, so werden die Koordinaten der angrenzenden Dreiecke korrigiert und das Dreieck selbst gelöscht.

4 Zusammenfassung

In diesem Abschnitt sind noch einmal die zu implementierenden Datentypen und Operatoren aufgelistet.

4.1 Zu implementierende Datentypen

- *point3d*
- *vector3d*
- *plane3d*
- *surface3d*
- *volume3d*

4.2 Zu implementierende Operatoren

$geo3d \in \{point3d, surface3d, volume3d\}$,
$sgeo3d \in \{surface3d, volume3d\}$

Name	Signatur		Bedeutung
size	$geo3d$	$\rightarrow$ *int*	Ermittelt die Anzahl der Elemente
bbox	$geo3d$	$\rightarrow$ *rect3*	Ermittelt die Bounding Box des Objekts
importSTL	*text*	$\rightarrow$ *volume3d*	Importiert eine STL-Datei
exportSTL	*volume3d* $\times$ *string* $\times$ *text* $\times$ *bool*	$\rightarrow$ *bool*	Export in eine STL-Datei
createCube	*point3d* $\times$ *point3d*	$\rightarrow$ *volume3d*	Erzeugen eines Quaders
createCylinder	*point3d* $\times$ *real* $\times$ *real* $\times$ *int*	$\rightarrow$ *volume3d*	Erzeugen eines Zylinders
createCone	*point3d* $\times$ *real* $\times$ *real* $\times$ *int*	$\rightarrow$ *volume3d*	Erzeugen eines Kegels
createSphere	*point3d* $\times$ *real* $\times$ *int*	$\rightarrow$ *volume3d*	Erzeugen einer Kugel
region2surface	*region*	$\rightarrow$ *surface3d*	Umwandeln einer Region in eine Oberfläche
region2volume	*region* $\times$ *real*	$\rightarrow$ *volume3d*	Umwandeln einer Region in einen Körper
mregion2volume	*mregion* $\times$ *real*	$\rightarrow$ *volume3d*	Umwandeln einer MRegion in einen Körper
rotate	$geo3d \times$ *point3d* $\times$ *vector3d* $\times$ *real*	$\rightarrow$ $geo3d$	Drehung eines Objekts
mirror	$geo3d \times$ *plane3d*	$\rightarrow$ $geo3d$	Spiegeln eines Objekts
translate	$geo3d \times$ *vector3d*	$\rightarrow$ $geo3d$	Verschieben eines Objekts
scaleDir	$geo3d \times$ *point3d* $\times$ *vector3d*	$\rightarrow$ $geo3d$	Skalierung in eine Richtung
scale	$geo3d \times$ *point3d* $\times$ *real*	$\rightarrow$ $geo3d$	Skalierung in alle Richtungen
union	$geo3d \times geo3d$	$\rightarrow$ $geo3d$	Vereinigung zweier Objekte
intersection	$geo3d \times geo3d$	$\rightarrow$ $geo3d$	Schnitt zweier Objekte
minus	$geo3d \times geo3d$	$\rightarrow$ $geo3d$	Differenz zweier Objekte
components	$sgeo3d$	$\rightarrow$ stream($sgeo3d$)	Zerlegung in Komponenten

Die Operationen **importSTL**, **union**, **intersection**, **minus** sowie die **IN**-Funktionen von Körpern und Flächen verwenden intern die Multi-Precision-Library, um Rundungsfehler zu vermeiden.