FaPra WS 2009/2010
Aufgaben der Phase 2

Ralf Hartmut Güting, Thomas Behr, Christian Düntgen, Simone Jandt

26. November 2009

Lehrgebiet Datenbanksystem für neue Anwendungen
Fakultät für Mathematik und Informatik
Fernuniversität Hagen

1 Datenimport in Records

Viele Forschungsarbeiten in unserem Lehrgebiet beschäftigen sich mit der Verarbeitung und Darstellung von raum-zeitlichen Daten in Datenbanken, speziell in SECONDO. Für den Austausch von Daten aus Geographischen Informationssystemen gibt es unter anderem das `E00`-Datenformat und für die Übertragung von Daten durch GPS-Empfänger das `NMEA0183`-Protokoll. SECONDO kann diese beiden Datenformate bisher nicht einlesen und in SECONDO Datentypen umwandeln. Dies soll dank Ihrer Mithilfe zukünftig anders sein. Ihre Aufgabe ist es, die existierende `ImExAlgebra`, die den Datenaustausch von SECONDO mit verschiedenen Dateiformaten ermöglicht, so zu erweitern, dass auch `ASCII`-Dateien im `E00` oder `NMEA0183` Format in SECONDO importiert werden können. Die von Ihnen eingelesenen Daten sollen in dem ebenfalls von Ihnen neu zu implementierenden Attributdatentyp `record` gespeichert werden. Der neue Attributdatentyp `record` soll dabei ein aus anderen Attributdatentypen von SECONDO zusammengesetzter strukturierter Datentyp sein, wie Sie ihn im Zusammenhang mit Programmiersprachen im Verlauf Ihres Studiums bereits kennengelernt haben.

Im Zusammenhang mit den `E00` Geodaten könnte ein `record` beispielsweise alle Angaben zu einer Straße aufnehmen. Attribute des `record`s wären dann beispielsweise der Straßenname (`string`), der Verlauf der Straße (`line`), die Länge der Straße (`real`), der Straßentyp(`string`) und die zulässige Höchstgeschwindigkeit(`real`). Bewegungsdaten von GPS-Empfängern, wie sie über das `NMEA0183`-Protokoll übermittelt werden, würden unter anderem ein `moving(point)` Element in einem `record` erzeugen.

Der Datenimport in Records zerfällt also in zwei Teilaufgaben. Die eine Teilaufgabe ist die Implementierung des neuen Datentyps `record` und seiner Operationen in einer neu zu erstellenden `RecordAlgebra` (siehe 1.1.1). Die andere Teilaufgabe ist die Erweiterung der `ImExAlgebra` um zwei Schnittstellen zum Import von `ASCII`-Dateien im `E00` bzw. `NMEA0183` Format. Die Importschnittstelle soll die Daten in SECONDO(siehe 1.1.2) in dem neuen Datentyp `record` ablegen.

1.1 Die Teilaufgaben

1.1.1 RecordAlgebra

Aus Programmiersprachen kennen Sie strukturierte Datentypen, die es Ihnen ermöglichen, mehrere vorhandene Datentypen zu einem einzigen neuen Datentyp zusammenzufassen. So wird zum Beispiel aus mehreren `string` (für Straße, Ort und Land) und `integer` (für Hausnummer und Postleitzahl) Werten der strukturierte Datentyp `Anschrift` erzeugt, und im restlichen Programm als eigener Datentyp verwendet. Analog dazu soll nun in SECONDO ein neuer Datentyp `record` implementiert werden. Als Elemente eines `record` sollen dabei alle in SECONDO vorhandenen Attributdatentypen (Elemente der Kind `DATA`) benutzt werden können. Der neue Datentyp `record` soll dabei selbst auch wieder in Relationen als Attribut verwendet werden können, also auch Element der Kind `DATA` sein. **Achtung!** Der Datentyp `record` ist auf Grund der Allgemeinheit der darin aufzunehmenden Elemente nicht trivial. Insbesondere müssen die in den Elementen evtl. enthaltenen FLOBs gesondert behandelt werden.

Natürlich soll der Datentyp `record` auch mehrere SECONDO Operationen zur Verfügung stellen. Und zwar:

1. **createRecord**: $((id_1 T_1)(id_2 T_2)...(id_n T_n)) \rightarrow$ `record`$((id_1 T_1)(id_2 T_2)...(id_n T_n))$, mit $T_i \in DATA$ erzeugt ein einzelnes `record` mit den Elementen der übergebenen Liste.

2. **transformT2Rstream**: `stream(tuple(`T`))` $\rightarrow$ `stream(record(`T`))` erzeugt aus einem Tupelstrom einen Strom von `record`s, bei dem jeder `record` die Attribute eines Tupels aufnimmt.

3. **transformR2Tstream**: `stream(record(`T`))` $\rightarrow$ `stream(tuple(`T`))` erzeugt aus einem Strom von `record`s einen Tupelstrom, bei dem jedes Tupel die Komponenten eines `record`s als Attribute aufnimmt.

4. **attr**: `record`$((id_1 T_1)(id_2 T_2)...(id_n T_n)) \times id_i \rightarrow T_i$ ermöglicht den gezielten Zugriff auf ein einzelnes Element eines `record`s.

1.1.2 Import von `E00` und `NMEA0183` Dateien

Die `ImExAlgebra` soll so erweitert werden, dass zukünftig auch `ASCII`-Dateien im `E00`- und `NMEA0183`-Format von SECONDO importiert werden können. Die beim Import ausgelesenen Informationen sollen in SECONDO jeweils in logisch sinnvollen `record`s abgelegt werden. Der `record` soll dabei nicht durch die Übergabe einer `NestedList` an die In-Funktion des `record`s erzeugt werden, sondern ist beim Datenimport explizit anzulegen.

E00 (ArcInfo Interchange File) ist ein Datenaustauschformat der Firma ESRI für Geometrien, Sachdaten und topologische Informationen. Es handelt sich um einfache `ASCII`-Dateien. Im Internet finden Sie unter `http://webhelp.esri.com/arcgisdesktop/9.3/index.cfm?TopicName=welcome` im Kapitel "`Data support in ArcGIS`", Unterabschnitt "`Coverages`" Informationen über den Aufbau von sogenannten "coverages". Diese werden in `E00`-Dateien gesichert. Informationen zum Aufbau der entsprechenden `E00`-Dateien finden im Internet unter `http://avce00.maptools.org/docs/v7_e00_cover.html`.

Sie werden feststellen, dass die in der `E00`-Datei auftauchenden Kürzel vor den Abschnitten im wesentlichen den Dateinamen der in einem `Coverage` enthaltenen Files entsprechen. Die durch die Kürzel benannten Abschnitte der Datei enthalten die Inhalte der entsprechenden `Coverage`-Dateien. Damit haben Sie alle nötigen Angaben für die Analyse einer `E00`-Datei vereint.

`E00`-Dateien enthalten also Beschreibungen von Punkten, Linienzügen und Polygonen (Regionen). Jedes dieser räumlichen Elemente kann über zusätzliche Informationen wie Namen, Länge, Flächeninhalt etc. verfügen. Beim Einlesen einer `E00`-Datei gilt es nun die über die Datei verteilten Informationen wieder zu sinnvollen Objekteinheiten zusammenzuführen und diese Objekte in einem entsprechend aufgebauten `record` aus SECONDO Datentypen abzubilden, um sie so in SECONDO verfügbar zu machen.

Die für die Abbildung der einzelnen `record`-Elemente erforderlichen Datentypen stehen in SECONDO schon zur Verfügung. So gibt es neben den Standarddatentypen für Zahlen (`int` und `real`) und Text (`string`) auch räumliche Datentypen für Punkte (`point`), Linien (`line`) und Regionen (`region`), die Sie als Elemente Ihrer `record`s verwenden können. Die Standarddatentypen befinden sich in der `StandardAlgebra`. Die räumlichen Datentypen finden Sie in der `SpatialAlgebra`. Für längere Texte gibt es in der `FTextAlgebra` den Datentyp `text`.

NMEA0183 ist ein ursprünglich von der Marine entwickeltes Protokoll zum Austausch von GPS-Daten über serielle Schnittstellen. Die Übertragung der Daten erfolgt im `ASCII`-Format, so dass die Datenströme auch als einfache Textdateien abgespeichert werden können. Aus solchen Datenströmen entstandene Textdateien im `NMEA0183`-Format sollen nun nach SECONDO importiert werden können.

Die einzelnen GPS-Signale eines zu verfolgenden Objektes werden blockweise übertragen. Jeder Datenblock besteht aus mehreren Datensätzen. Jede Zeile im Textblock entspricht dabei einem Datensatz. Jeder Datensatz beginnt mit dem Zeichen $ und endet mit CR/LF (Zeilenende und Vorschub) und ist inklusive der Begrenzer maximal 82 Zeichen lang. Jeder Datensatz enthält die Information, um welche Geräteart es sich handelt (Stelle 1-2) und welches Datensatzformat (Stelle 3-5) in der folgenden kommaseparierten Liste übertragen wird. Eine Übersicht der möglichen Datensätze des Protokolls findet sich im Internet unter `http://www.tronico.fi/OH6NT/docs/NMEA0183.pdf`.

Die Datenblöcke können abhängig vom übertragenden Gerät sehr viele Datensätze mit unterschiedlichen Informationen beinhalten. So können neben der Positionsinformation z.B. auch Geschwindigkeit, Bewegungsrichtung, Höhe, Wegpunkte, Windgeschwindigkeiten usw. übertragen werden. Für uns sind im wesentlichen die Daten interessant, die sich auf die räumliche Bewegung des Objektes im 3-dimensionalen Raum und seine Umgebung beziehen.

Alle Datenblöcke einer Übertragung geben zusammen die Bewegung eines via GPS-Empfängers beobachteten Objektes wieder. Deshalb müssen alle übertragenen Datenblöcke in SECONDO wieder zu einer Einheit zusammengefasst werden. Der Strom von GPS-Daten enthält dabei vor allem auch Daten, die sich im Verlauf der Zeit ändern (etwa Positionsdaten, Geschwindigkeiten, Höhe). In SECONDO gibt es für solche sich im Verlauf der Zeit verändernde Datenobjekte das Konzept der `moving`-Datentypen. `moving` kann sowohl auf die Standarddaten-

typen wie `int`, `real` und `bool` angewandt werden, als auch auf die räumlichen Datentypen `point`, `line` und `region`. Für jeden der Datentypen α wird der entsprechende `moving`(α)-Datentyp in SECONDO mit `m`α bezeichnet. Eine detaillierte Erklärung des `moving` Konzeptes in SECONDO finden Sie im Internet unter der URL `http://dna.fernuni-hagen.de/papers/AlgMovObj.pdf` (Abschnitte 1—3).

Eine wechselnde Geschwindigkeit kann also als `mreal` in SECONDO gespeichert werden. Die von einem sich im Gelände bewegenden Objekt eingenommenen Positionen werden in SECONDO durch einen `mpoint` repräsentiert.

Sehen Sie in der Importfunktion für `NMEA0183`-Dateien Parameter vor, die es ermöglichen Datensätze von einer bestimmten Qualitätsstufe zu wählen. Datensätze mit geringerer Qualität als der angegebenen sind dann beim Import zu überlesen. Die Qualität eines Datensatzes kann von den Parametern `GPS Quality Indicator` und `Number of Satellites in view` abgeleitet werden.

Hinweis: Bitte beachten Sie beim Import der Daten, dass das `NMEA0183`-Protokoll ursprünglich aus der Seefahrt kommt. Je nach Datensatz werden deshalb beispielsweise Geschwindigkeitsangaben in Knoten über Grund, Abstände in nautischen Meilen und Höhenwerte in Fuß übertragen. Diese Maße müssen Sie natürlich beim Import in das metrische System umwandeln: Geschwindigkeit in Meter pro Sekunde, Abstände und Höhen in Meter.

1.2 Aufgabenstellung

1.2.1 Algebra-Implementierung

Implementieren Sie die oben genannten Erweiterungen in SECONDO. Kommentieren Sie dabei Ihren Quellcode ausreichend und verständlich. Die Dateien müssen durch das PD-System bearbeitet werden können.

1.2.2 Display-Klassen

Erweitern Sie die textbasierten Schnittstellen von SECONDO um Display-Klassen für die `RecordAlgebra`. Alle Kommentare müssen javadoc-fähig sein.

1.2.3 Erweiterung der GUI

Erweitern Sie den HoeseViewer so, dass auch `record`s angezeigt werden können. Beachten Sie dabei, dass Sie nicht wissen, welche Datentypen die Elemente der `record` enthalten. Verwenden Sie daher einen generischen Ansatz. Achten Sie auf die vernünftige Kommentierung Ihres Quellcodes. Das `javadoc` Tool soll ordentliche Schnittstellenbeschreibungen aus Ihrem Code generieren können. Denken Sie daran, dass möglichst viel die Funktionalität der Subtypen verwendet werden soll. Daher muss Ihre eigene Display-Klasse möglichst viele Interfaces, die für die Realisierung von Funktionen notwendig sind, implementieren. Da es sich um einen Attributdatentyp handelt, darf eine Menge nur einen einzigen Eintrag im `QueryResult` erzeugen. Implementieren Sie daher das `ExternDisplay` Interface. Zeigen Sie die einzelnen Elemente dann in einem separaten Fenster an. Falls Ihre Unterklasse ebenfalls das `ExternDisplay` Interface implementiert, reagieren Sie entsprechend auf einen (Doppel-)Klick auf das jeweilige Element.

In dieser Aufgabe arbeiten Sie mit einer neueren Version der Javagui. In dieser haben einige Methoden des Hoeseviewers einen zusätzlichen Parameter

für die Einrücktiefe der textuellen Darstellung von Objekten erhalten. Dieser ist insbesondere bei der Darstellung geschachtelter Objekte hilfreich.

1.2.4 Tests

Erstellen Sie für jede Signatur der Operatoren einen entsprechenden Eintrag in ihrer `example`-Datei. Schreiben Sie `TestRunner` Dateien, die sicherstellen, dass jeder Programmzweig Ihrer Implementierung durchlaufen wird. Testen Sie Ihre Implementierung auch mit großen Datenmengen. `E00`-Dateien können Sie im Internet z.B. unter `http://data.geocomm.com/catalog/` finden. `NMEA0183`-Dateien können von uns zur Verfügung gestellt werden. Versuchen Sie dabei einerseits, innerhalb eines `record`s viele Werte abzulegen. Andererseits sollten Sie auch große Relationen mit vielen `record` Attributen erzeugen und mit diesen Tests ausführen. Achten Sie bei solchen Tests auf den Speicher, der von SECONDO benötigt wird (Kommando `top` in der `bash`).

2 Unterstützung von Experimenten mit Indexstrukturen

2.1 Überblick

Das erweiterbare Datenbanksystem SECONDO dient unter anderem der Evaluation von bekannten und neuartigen Datenstrukturen und Algorithmen. Neben mathematisch-analytischen Modellen spielen insbesondere experimentelle Verfahren eine wichtige Rolle zur Bestimmung der praktischen Leistungsfähigkeit von neuen Verfahren.

Solche Experimente sollen einerseits nachvollziehbare und reproduzierbare Ergebnisse liefern. Dazu bedarf es genau zu spezifizierender Randbedingungen für die Experimente. Ferner möchte man häufig experimentelle Befunde analysieren, etwa um bestimmte kritische Eigenschaften genauer zu untersuchen.

Indexstrukturen sind wichtige Komponenten in Datenbanksystemen. Sie dienen insbesondere dem effizienten Zugriff auf gespeicherte Daten. Daher kommt ihnen eine besondere Bedeutung zu — auch im Rahmen der experimentellen Evaluation.

Im Rahmen dieser Aufgabe sollen Sie dazu beitragen, die Möglichkeiten für Experimente innerhalb von SECONDO zu verbessern. Dazu sollen Sie einerseits die Unterstützung für Analysen von bestehenden Datenstrukturen verbessern. Andererseits sollen wichtige, grundlegende Indexstrukturen zur Verfügung gestellt werden.

2.2 Teilaufgabe I: R-Baum-Introspektion

Mit der `RTreeAlgebra` liegt eine Implementierung des R*-Baums vor. (Durch Änderung fest kodierter Parameter lassen sich auch R-Bäume und R^+-Bäume darstellen.)

Ziel dieser Teilaufgabe ist es, die vorhandenen Möglichkeiten zur Analyse von R-Bäumen in SECONDO zu verbessern. Dazu soll einerseits die `RTreeAlgebra` erweitert werden, andererseits soll die JavaGUI um Möglichkeiten zur komfortablen Exploration von R-Bäumen erweitert werden.

2.2.1 Bereits implementierte Funktionalität

RTreeAlgebra Folgende Operatoren der `RTreeAlgebra` ermöglichen es bereits, Informationen aus einem R-Baum der Dimension D zu extrahieren:

- **nodes**: (rtree< D > (tuple ((x_1 t_1)...(x_n t_n))) t_i false) → `stream(tuple((level int) (nodeId int) (MBR rect<D>) (fatherId int) (isLeaf bool) (minEntries int) (maxEntries int) (countEntries int)))`
  Dieser Operator durchläuft den gesamten R-Baum und liefert einen Strom von Tupeln. Jedes Tupel beschreibt einen inneren Knoten, ein Blatt oder einen Schlüsseleintrag der Indexstruktur.

- **treeheight**: `rtree`< D > → `int`
  Gibt die Höhe des R-Baums zurück.

- **no_nodes**: `rtree`< D > → `int`
  Gibt die Anzahl benutzter Knoten zurück.

- **no_entries**: `rtree`< D > → `int`
  Informiert über Anzahl gespeicherter Schlüsseleinträge (ohne innere Knoten oder Blattknoten).

- **bbox**: `rtree`< D > → `rect`< D >
  Liefert das minimale achsenparalle Rechteck (auch *MBR* oder *bounding box* genannt) der gesamten Indexstruktur.

- **entries**: `rtree`< D > → `stream(tuple((nodeid int) (bbox key) (tid tupleid))`
  Erzeugt einen Strom, der alle Schlüssel- und Dateneinträge enthält. Der implementierte Datentyp rtree speichert mit dem Schlüssel ausschließlich Tupelidentifikatoren. Somit stellen R-Bäume in SECONDO immer Sekundärindexe dar, da sie Tupel in einem anderen (Relations-) Objekt referenzieren.

- **getFileInfo**: `rtee`< D > → `text`
  Dieser Operator liefert einige grundlegende Angaben zu der in der Implementierung des R-Baum-Objekts verwendeten BerkeleyDB-Datei.

Leider ist es mit den bestehenden Operatoren derzeit nicht möglich, Informationen zu einzelnen Knoten oder Einträgen gezielt und effizient abzurufen. Stattdessen müssen immer alle Knoten verarbeitet werden — was bei R-Bäumen mit vielen Einträgen (z.B. N>1.000.000) beträchtlichen Aufwand verursacht.

JavaGUI In der JavaGUI können R-Bäume derzeit nicht direkt dargestellt werden. Allerdings können im HoeseViewer Relationen, die Rechtecke enthalten, dargestellt werden. Mittels der in Abschnitt 2.2.1 vorgestellten Operatoren und den Operatoren der `RectangleAlgebra` können somit die MBRs/Schlüssel der R-Baum-Knoten und deren Schlüsseleinträge visualisiert werden. Dabei müssen n-dimensionale MBRs auf den 2-dimensionalen Raum projiziert werden.

2.2.2 Aufgabenstellung

Erweiterung der RTreeAlgebra Zu implementieren sind drei zusätzliche Operatoren der `RTreeAlgebra`. Der Quellcode ist ausreichend und verständlich zu kommentieren. Die Dateien müssen durch das PD-System bearbeitet werden können.

1. **getRootNode**: `rtree`$< D >\rightarrow$ `int`
   Dieser Operator liefert die Recordnummer des Wurzelknotes des R-Baums als Integer-Wert

2. **getNodeInfo**: `rtree`$< D > \times$ `int` $\rightarrow$ `stream(tuple((NodeId int) (MBR rect`$< D >$`) (NoOfSons int) (IsLeafNode bool) (IsRootNode bool) (MBRsize real) (MBRdead real) (MBRoverlapSize real) (MBRoverlapsNo real) (MBRdensity real)))`
   Dieser Operator liefert Informationen über einen bestimmten Knoten des R-Baums. Der Knoten wird über die zum Knoten gehörende Recordnummer (als `int`) selektiert. Gehört die Recordnummer nicht zum R-Baum, so liefert der Operator einen leeren Tupelstrom. Ansonsten enthält der Ergebnisstrom genau ein Tupel. Dieses enthält Informationen zum besagten R-Baumknoten:

   - Das Attribut *NodeId* enthält die Recordnummer des Knotens als Integer.
   - Das Attribut *MBR* enthält das MBR des Knotens als $< D >$-dimensionales Rechteck.
   - Das Attribut *NoOfSons* enthält die Anzahl der Söhne des Knotens als Integer.
   - Das Attribut *IsLeafNode* ist genau dann TRUE, wenn der Knoten ein Blattknoten des Baumes ist.
   - Das Attribut *IsRootNode* ist genau dann TRUE, wenn der Knoten die Wurzel des Baumes darstellt.
   - Das Attribut *MBRsize* ist das Mass des Volumens/der Fläche der MBRs des Knotens.
   - Das Attribut *MBRdead* ist das Mass des Volumens/der Fläche des MBR, das nicht von mindestens einem im Knoten enthaltenen Sohn/ Key-MBR überdeckt wird. Wie man das Massproblem für Rechteckmengen lösen kann, wird etwa im Kurs Datenstrukturen beschrieben.
   - Das Attribut *MBRoverlapSize* ist das Mass des Volumens/der Fläche, auf dem sich die MBRs von mindestens zwei Söhnen/Keys überdecken.
   - Das Attribut *MBRoverlapsNo* ist die Anzahl der sich paarweise überlappenden enthaltenen Sohn/Key-MBRs.
   - Das Attribut *MBRdensity* ist die durchschnittliche Sohn-Dichte des Knotens, also die durchschnittliche Anzahl von Söhnen/Keys, die eine Volumen-/Flächeneinheit des Knoten-MBRs überdecken.

3. **getNodeSons**: `rtree`$< D >$ $\times$ `int` $\rightarrow$ `stream(tuple((NodeId int) (SonId int) (SonMBR rect`$< D >$`)))`
   Dieser Operator liefert die Recordnummern und MBRs aller Söhne eines durch seine Recordnummer spezifizierten Knotens als Tupelstrom. Ist die übergebene Recordnummer ungültig, wird eine leerer Strom zurückgegeben. Die Recordnummer des untersuchten Knotens wird in NodeId weitergereicht.

Erweiterungen der JavaGUI Es soll ein Viewer zur Verfügung gestellt werden, der eine komfortable, interaktive Introspektion von R-Bäumen ermöglicht:

1. Die Struktur des Baumes soll hierarchisch dargestellt werden.
2. Es sollen globale Informationen (Höhe des Baums, Anzahl Knoten, Anzahl Schlüsssel) angezeigt werden.
3. Die Information, die man mittels der neuen Operatoren **getNodeInfo** und **getNodeSons** über einen Knoten erhalten kann, soll textuell und graphisch dargestellt werden. Die grafische Darstellung der Knoten-Gesamt-MBRs und der enthaltenen MBRs soll mindestens 2-dimensional erfolgen. Bei 2-dimensionaler Darstellung muss die verwendete Projektion wählbar sein (d.h. der Benutzer muss interaktiv wählen können, welche beiden Dimensionen dargestellt werden).
4. Für Blattknoten sollen (nach Auswahl einer Relation) die referenzierten Tupel mit ihren Daten dargestellt werden können. Dabei sollen die geometrischen Daten gemeinsam mit den MBRs des referenzierenden R-Baums graphisch dargestellt werden können.
5. Nach Auswahl eines R-Baumes soll zunächst der Wurzelknoten dargestellt werden.
6. Benutzer sollen interaktiv durch den Baum navigieren können, ohne lange Wartezeiten in Kauf nehmen zu müssen. Dies muss insbesondere bei großen R-Baumstrukturen (viele Millionen Schlüssel) möglich sein.
7. Kommentieren Sie Ihren Java-Quellcode ausreichend und verständlich. Die Dateien müssen durch das javadoc-System bearbeitet werden können.

Hinweise zur Visualisierung

- Sie könnten etwa eine vollständig graphische Repräsentation wählen oder eine geteilte Ansicht mit Baumansicht (ähnlich wie im Windows Explorer) für die Knotenhierarchie und eine graphische Anzeige für die MBRs.
- Es wäre sinnvoll, den Gesamt-MBR des Baumes bzw. aller übergeordneten Knoten anzuzeigen, um dem Benutzer die Orientierung zu erleichtern.
- Gegebenenfalls sollte man beliebige Knoten selektieren und gemeinsam darstellen können. Aktuell markierte Knoten sollten hervorgehoben werden.
- Die notwendigen Informationen können aus der GUI heraus durch automatisch generierte Anfragen abgefragt werden. Der Benutzer muss dazu keine expliziten Anfragen stellen (vgl. UpdateViewer).

2.3 Teilaufgabe II: Parametrisierbare B-Bäume

Die `BTreeAlgebra` stellt bereits B-Bäume in SECONDO zur Verfügung. Allerdings werden diese direkt über die von BerkeleyDB angebotenen B-Bäume realisiert. Dies ist zwar effizient, bedeutet jedoch leider auch, dass man wenig Einfluss auf wichtige Parameter, wie die Knotengrößen/-kapazitäten, Knotencachegröße etc. nehmen kann. Auch erhält man kaum Informationen zur Anzahl der z.B. während einer Datenbankanfrage besuchten Knoten. Daher sollen Sie parametrisierbare B-Bäume zur Verfügung stellen. Der Datentyp und die Operatoren sind in einer neuen Algebra, der `BTree2Algebra`, zu realisieren.

2.3.1 Datentypen der BTree2Algebra

Sie sollen einen Datentyp für umfangreich parametrisierbare B-Bäume zur Verfügung stellen. Implementieren Sie dazu eine Algebra `BTree2Algebra` unter Verwendung der Klasse `SmiRecordFile` (include/SecondoSMI.h). Alle Daten des B-Baumes sollen ausschließlich in den Blättern gespeichert werden (es handelt sich also um B^+-Bäume). Die Blattknoten sind durch Zeiger zu einer linearen Liste zu verbinden, so dass Range-Queries besonders effizient durchgeführt werden können. Der Datentyp werde wie folgt durch eine NestedList dargestellt:

```
(btree2 <filldegree> <nodesize> <keytype> <loadtype> <unique>)
```

Dabei gelte:

- `<filldegree>` bestimmt den Mindestfüllgrad f für Nichtwurzelknoten im Baum. Für einen B^+-Baum gilt etwa ein Mindestfüllgrad von 1/2, für einen B^*-Baum einer von 2/3. Es muss gelten: $0.0 < f < 1.0$. Andernfalls wird $f = 0.5$ gesetzt.
- `<nodesize>` bestimmt die Größe n eines Knotens in Byte. Falls die Knotengröße $n < n_min$ ist, so wird sie auf n_min gesetzt. n_min ist dabei eine von Ihnen zu bestimmende systembedingte minimale Knotengröße, vielleicht 128, 256 oder 512 Byte.
- `<keytype>` bestimmt den Datentyp T_k, der als Schlüssel dient. Dies muss ein Typ der Kind INDEXABLE sein.
- `<loadtype>` bestimmt den Datentyp T_d, der mit dem Schlüssel als Nutzdaten in den Blättern gespeichert wird. Es darf sich um einen beliebigen SECONDO-Attributdatentypen handeln. Ein spezieller "Typ" `none` besagt, dass der Baum nur die Schlüssel selbst speichert.
- `<unique>` ist ein SymbolAtom u und bestimmt, ob Schlüssel bzw. Schlüssel/Nutzdaten-Paare mehrdeutig oder eindeutig sind. Fehleingaben sind im Typemapping abzufangen. Folgende Optionen sind vorzusehen:
  - `multiple`: Schlüssel/Nutzdatenkombinationen dürfen beliebig oft gespeichert werden.
  - `uniqueKeyMultiData`: Jede Kombination von Schlüssel/Nutzdaten darf nur einmal eingetragen sein.
  - `uniqueKey`: Jeder Schlüssel darf höchstens einmal im Baum enthalten sein.

Hinweise

- Die Typen der `StandardAlgebra` (`int`, `real`, `bool`, `string`) sollen direkt als Schlüssel unterstützt werden, d.h. effizient gespeichert werden. Außerdem soll die natürliche Ordnung auf diesen Datentypen verwendet werden. Die `BTreeAlgebra` hat dies ebenso implementiert.

- Andere Datentypen sollen als Schlüssel unterstützt werden, wenn Sie der Kind INDEXABLE (include/StandardAttribute.h) angehören. Diese stellt sicher, dass ihre Miglieder ihren Wert in einen String konvertieren können. Die solchermaßen transformierten "externen" Schlüsselattributwerte lassen sich dann als "interne" Schlüssel im B-Baum verwenden (vgl. ebenfalls `BTreeAlgebra`).

- Innere Knoten und Blattknoten sollen dieselbe fixe Recordgröße (=Nodegröße) verwenden. Allgemein haben die Einträge in inneren Knoten eine andere Größe als die Blattknoten: Während innere Knoten Schlüssel und Recordnummern als Zeiger auf die Söhne enthalten, enthalten Blattknoten zwei Recordnummern für das linke und rechte Nachbarblatt sowie die Schlüssel bzw. Schlüssel/Datenpaare der Einträge. Daher werden innere Knoten und Blattknoten unterschiedliche Eintragskapazitäten aufweisen.

- In der `RelationAlgebra` können Sie lernen, wie Attributdatentypen persistent gemacht werden können. Neben dem `SmiRecordFile` für die Baumknoten sollten Sie ein weiteres, separates für die etwaigen FLOB-Daten der Nutzdatentypen verwenden.

- Um Relationen zu indizieren, können Tupelidentifikatoren (Datentyp `tid`) verwendet werden, die das jeweils entsprechende Tupel in einer Relation referenzieren. Dann kann man den Operator *gettuples* verwenden, um direkt auf referenzierte Tupel zuzugreifen.

- Zur Unterstützung der Statistikoperatoren und des Schätzoperators (s. 2.3.2) sollten Sie einige statistische Informationen (z.B. Anzahl der Knoten, Anzahl der Einträge, ggf. Anzahl eindeutiger Einträge) summarisch festhalten.

- Jeder B-Baum soll einen Cache für Knotenzugriffe vorhalten. Die Größe des Caches kann vom Benutzer zur Laufzeit verändert werden. Ferner kann der Benutzer gezielt Knoten im Cache fixieren. (Vgl. Abschnitt 2.3.2.)

- Bei der Dimensionierung der Standard-Cachegröße dürfen Sie maximal soviel Speicher verwenden, wie die Konstante `MaxMemPerOperator` vorgibt. Der Benutzer darf eine Cachegröße, die diesen Wert übersteigt, explizit anfordern. Die Cachegröße muss im B-Baum gespeichert werden.

- Der B-Baum soll Zähler für die Anzahl der Knotenzugriffe und die Anzahl der entsprechenden Cache-Hits bereit stellen. Operatoren erlauben das Löschen und Abfragen der Zählerstände. (Vgl. Abschnitt 2.3.2.)

2.3.2 Operatoren der BTree2Algebra

Im Folgenden werden die zu implementierenden Operatoren aufgeführt. Falls nichts weiteres angegeben wird, soll die Semantik der Operatoren derjenigen der namensgleichen Operatoren in der `BTreeAlgebra`, bzw. der nach Abschnitt 2.2.2 erweiterten `RTreeAlgebra`, entsprechen.

Da die `BTree2Algebra` die `BTreeAlgebra` ersetzen können soll, müssen neben den neuen, allgemeinen Signaturen auch noch einige spezialisierte Operatoren implementiert werden, die die alte Schnittstelle implementieren. Häufig handelt es sich dabei jedoch um "Wrapper"-Operatoren, welche die allgemeinen Operatoren verwenden.

Konstruktionsoperatoren Der erste Operator erzeugt einen neuen parametrisierten B-Baum. Der erste Parameter stellt den Strom von Tupeln dar, die jeweils mindestens das Schlüsselattribut a_k (4. Argument) vom Type T_k und das Datenattribut a_d (5. Argument) vom Typ T_d enthalten. Ist $a_d =$ `none`, so gelte $T_k =$ `none` — es werden im B-Baum nur die Schlüssel gespeichert. Das zweite Argument ist der Mindestfüllgrad, das dritte die Recordgröße in Byte, das letzte Argument ist das Symbol, das besagt, ob Schlüssel/Daten eindeutig oder mehrdeutig sein sollen (`multiple`, `uniqueKeyMultiData` oder `uniqueKey`).

Eine zweite und dritte Variante des Operators **createbtree** sollen die entsprechenden Operatoren der `BTreeAlgebra` ersetzen und sich genauso verhalten. Sie dienen der Erzeugung sekundärer Indizes. Diese Operatoren erhalten im Vergleich zum "Original" zwei optionale Parameter, um die Füllrate und die Knotengröße bestimmen zu können. Werden diese nicht explizit angegeben, so sei $f = 0.5$ und $n = 4096$.

1. **createbtree2**: `stream(tuple(T))` $\times$ `real` $\times$ `int` $\times a_k \times a_d \times u \rightarrow$ (btree2 f n T_k T_d u)
   createbtree: `rel(tuple((`x_1 t_1`)...(`x_n t_n`)))` $\times a_k$ $[\times f \times n] \rightarrow$ (btree2 f n T_k `tid multiple`)
   createbtree: `stream(tuple((`x_1 t_1`)...(`a_k T_k`)...(`x_n t_n`) (id tid)))` $\times a_k$ $[\times f \times n] \rightarrow$ (btree2 f n T_k `tid multiple`)

Manipulationsoperatoren Diese Operatoren verändern den persistenten B-Baum, indem Sie Einträge einfügen, löschen oder ändern. Sie reichen fernerhin die Eingabe weiter, damit ggf. auch andere Indizes verändert werden können. Schlüssel- und Datenattribut sind explizit anzugeben. Jedes Tupel der Eingabe soll höchstens einen passenden Eintrag im Baum manipulieren. a_k und a_d bezeichnen das zu verwendende Schlüssel- bzw. Datenattribut, die dem jeweiligen Typ des B-Baums entsprechen müssen.

Wiederum gibt es zu den Manipulationsoperatoren Wrapper, die das Verhalten der Operatoren aus der `BTreeAlgebra` genau nachbilden.

1. **insertbtree2**: `stream(tuple(T))` $\times$ (btree2 f n T_k T_d u) $\times a_k \times a_d \rightarrow$ stream(tuple(T))
   insertbtree: `stream(tuple(X@[TID tid]))` $\times$ (btree2 F n T_k `tid multiple`) $\times a_k \rightarrow$ stream(tuple(X@[TID tid]))]

2. **deletebtree2**: `stream(tuple(T))` $\times$ (btree2 f n T_k T_d u) $\times a_k \times a_d \rightarrow$ `stream(tuple(T))`

deletebtree: `stream(tuple(X@[TID tid]))` × (btree2 f n T_k `tid multiple`) $\times a_k \rightarrow$ stream(tuple(X@[TID tid]))]

3. **updatebtree2**: `stream(tuple(T))` × (btree2 f n T_k T_d u) $\times a_k \times a_d \rightarrow$ `stream(tuple(T))`
   updatebtree: `stream(tuple(X@[(`a_1 x_1`)...(`a_k T_k`)...(`a_n x_n`) (TID tid)]))` × (btree2 f n T_k `tid multiple`) $\times a_k \rightarrow$ `stream(tuple(X@[(`a_1 x_1`)...(`a_k T_k`)...(`a_n x_n`) (TID tid)]))`

Anfrageoperatoren Diese Operatoren stellen Anfragen gegen den Baum dar. Sie erzeugen einen geordneten Tupelstrom aller passenden Schlüssel, bzw. Schlüssel/Daten-Paare, die im Baum aufgefunden werden.

Auch zu den Anfrageoperatoren gibt es Pendants zu den Operatoren aus der `BTreeAlgebra`.

1. **exactmatch2**: (btree2 f n T_k `none` u) $\times T_k \rightarrow$ `stream(tuple((Key` T_k`)))`
   exactmatch2: (btree2 f n T_k T_d u) $\times T_k \rightarrow$ `stream(tuple((Key` T_k`) (Data` T_d`)))`
   exactmatchS: (btree2 F n T_k `tid multiple`) $\times T_k \rightarrow$ `stream(tuple((id tid)))`
   exactmatch: (btree2 F n T_k `tid multiple`) × `rel(tuple(T))` $\times T_k \rightarrow$ `stream(tuple(T))`

2. **range2**: (btree2 f n T_k `none` u) $\times T_k \times T_k \rightarrow$ `stream(tuple((Key` T_k`)))`
   range2: (btree2 f n T_k T_d u) $\times T_k \times T_k \rightarrow$ `stream(tuple((Key` T_k`) (Data` T_d`)))`
   rangeS: (btree2 F n T_k `tid multiple`) $\times T_k \times T_k \rightarrow$ `stream(tuple((id tid)))`
   range: (btree2 F n T_k `tid multiple`) × `rel(tuple(T))` $\times T_k \times T_k \rightarrow$ `stream(tuple(T))`

3. **leftrange2**: (btree2 f n T_k `none` u) $\times T_k \rightarrow$ `stream(tuple((Key` T_k`)))`
   leftrange2: (btree2 f n T_k T_d u) $\times T_k \rightarrow$ `stream(tuple((Key` T_k`) (Data` T_d`)))`
   leftrangeS: (btree2 F n T_k `tid multiple`) $\times T_k \rightarrow$ `stream(tuple((id tid)))`
   leftrange: (btree2 F n T_k `tid multiple`) × `rel(tuple(T))` $\times T_k \rightarrow$ `stream( tuple(T))`

4. **rightrange2**: (btree2 f n T_k `none` u) $\times T_k \rightarrow$ `stream(tuple((Key` T_k`)))`
   rightrange2: (btree2 f n T_k T_d u) $\times T_k \rightarrow$ `stream(tuple((Key` T_k`) (Data` T_d`)))`
   rightrangeS: (btree2 F n T_k `tid multiple`) $\times T_k \rightarrow$ `stream(tuple((id tid)))`
   rightrange: (btree2 F n T_k `tid multiple`) × `rel(tuple(T))` $\times T_k \rightarrow$ `stream(tuple(T))`

Beschreibungsoperatoren Diese Operatoren dienen dazu, Strukturinformationen über den B-Baum anzufordern. Der **keyrange** operator erhält ein Gegenstück zu seiner Implementierung in der `BTreeAlgebra`

1. **keyrange2**: (btree2 f n T_k T_d u) $\times T_k$ → `stream(tuple( (Less real) (Equal real) (Greater real) (NumOfKeys int)))`
   keyrange: (btree2 F n T_k `tid multiple`) × `rel(tuple(T))` $\times T_k$ → `stream(tuple( (Less real) (Equal real) (Greater real) (NumOfKeys int)))`
2. **getFileInfo**: (btree2 f n T_k T_d u) → `text`
3. **treeheight**: (btree2 f n T_k T_d u) → `int`
4. **no_nodes**: (btree2 f n T_k T_d u) → `int`
5. **no_entries**: (btree2 f n T_k T_d u) → `int`
6. **getRootNode**: (btree2 f n T_k T_d u) → `int`
7. **getNodeInfo**: (btree2 f n T_k T_d u) × `int` → `stream(tuple( (NodeId int) (NoOfSons int) (IsLeafNode bool) (IsRootNode bool) (MinKey` T_k`)))`
8. **getNodeSons**: (btree2 f n T_k T_d u) × `int` → `stream(tuple( (NodeId int)(SonId int) (Lower` T_k`) (Upper` T_k`)))`
9. **internal_node_capacity**: (btree2 f n T_k T_d u) → `int`
10. **leaf_node_capacity**: (btree2 f n T_k T_d u) → `int`
11. **getMinFillDegree**: (btree2 f n T_k T_d u) → `real`

keyrange dient dazu, die Anzahl der Tupel zu schätzen, die Anfrageoperatoren wahrscheinlich liefern werden. Es wird geschätzt, wieviele Einträge im B-Baum kleiner, gleich und größer einem übergebenen Schlüssel sind. Diese Information kann etwa zur Selektivitätsschätzung im Optimierer verwendet werden. Anstatt eine vollständige Suche durchzuführen, soll dieser spezielle Operator jedoch schnelle Ergebnisse liefern. Daher sollen nur globale statistische Angaben (etwa: Anzahl der Einträge im Baum) sowie eine systematische, schnelle Inspektion einiger weniger Knoten im Baum verwendet werden.

getMinFillDegree liefert den für den Baum geltenden minimalen Knotenfüllgrad.

Spezialoperatoren Diese Operatoren dienen speziell der Durchführung von Experimenten mit den Indexstrukturen und deren Anpassung an konkrete Datenbankanforderungen.

1. **reset_counters**: (btree2 f n T_k T_d u) → `bool`
2. **set_cache_size**: (btree2 f n T_k T_d u) × `int` → `bool`
3. **get_cache_size**: (btree2 f n T_k T_d u) → `int`
4. **pin_nodes**: stream(int) × (btree2 f n T_k T_d u) × `int` → `stream(tuple( (Node int) (Ok bool)))`
5. **unpin_nodes**: `stream(int)` × (btree2 f n T_k T_d u) → `stream(tuple( (Node int) (Ok bool)))`

6. **get_pinned_nodes**: (btree2 f n T_k T_d u) → `stream(int)`

7. **get_no_nodes_visited**: (btree2 f n T_k T_d u) → `int`

8. **get_no_cachehits**: (btree2 f n T_k T_d u) → `int`

Der B-Baum kann immer drei Knoten im Hauptspeicher halten (um Merge, Balance and Split Operationen duchführen zu können). Zusätzlich kann er einen Cache verwenden, auf dessen Größe und Verwendung der Benutzer Einfluss nehmen kann. Der Operator **set_cache_size** setzt die Größe des vom B-Baum verwendeten Knoten-Caches neu fest, der Operator **get_cache_size** fragt diese für den B-Baum ab. **pin_node** erlaubt es dem Benutzer, bestimmte Knoten (spezifiziert durch einen Strom von Recordnummern) im Cache festzusetzen. Diese Knoten werden dann zukünftig dauerhaft im Cache festgehalten. Beispielsweise kann man so bestimmte, häufig verwendete Knoten (meistens die obersten Ebenen des Baumes) im Cache fixieren, um die Zugriffe zu beschleunigen. Scheitert eine Fixierung, etwa weil bereits der gesamte Cache fixiert wurde, so wird im entsprechenden Tupel FALSE zurückgegeben. Mittels **unpin_node** kann man die Fixierung von Knoten aufheben. Auch hier wird in der Ausgabe für jeden Knoten signalisiert, ob die Operation erfolgreich war. Der Operator **get_pinned_nodes** liefert einen Strom mit den Recordnummern aller fixierten Knoten.

Der B-Baum stellt Zähler zur Verfügung, die die Anzahl der Knotenzugriffe und die Anzahl der entsprechenden Cache-Hits zählen. Diese Zähler kann man mittels **reset_counters** zurücksetzen und mittels **get_no_nodes_visited** und **get_no_cachehits** abfragen.

2.3.3 Aufgabenstellung

Implementieren und testen Sie die Datentypen nach Abschnitt 2.3.1 und die Operatoren nach Abschnitt 2.3.2 in SECONDO. Kommentieren Sie dabei Ihren Quellcode ausreichend und verständlich. Die Dateien müssen durch das PD-System bearbeitet werden können.

3 3D-Visualisierung von Objekten

3.1 Motivation

In SECONDO sind eine ganze Reihe räumlicher und raum-zeitlicher Objekte implementiert. In der `SpatialAlgebra` findet man die Typen `point` (ein einzelner Punkt), `points` (eine Menge von Punkten), `line` (eine komplexe Linie), `region` (ein Gebiet). Zusätzlich sind in der `RectangleAlgebra` Rechtecke in verschiedenen Dimensionen implementiert.

Raum-zeitliche Objekte sind solche Objekte, die ihre Geometrie (Position, Form) im Laufe der Zeit ändern. Konzeptionell handelt es sich bei solchen Objekten um eine Funktion, die einen Zeitpunkt auf eine Teilmenge des Euklidischen Raums R^2 abbildet.

$$moving(SPATIAL) : instant \rightarrow P(R^2)$$

mit $P(R^2) = \{r \subseteq R^2 | r \text{ ist begrenzt}\}$.

Beispielsweise kann man so einen sich bewegenden Punkt beschreiben. Die in SECONDO verwendete Implementierung verwendet Sequenzen sogenannter *Units* für die Darstellung der Funktion. Eine Unit besteht aus einem Zeitintervall sowie den Parametern einer einfachen parametrisierbaren Funktion, die die Werteveränderung innerhalb dieses Zeitintervalls beschreibt. Die Parameter der Darstellung eines sich bewegenden Punktes sind die Start- und die Endposition des Punktes in diesem Zeitintervall. Bewegt sich der Punkt im angegebenen Zeitraum nicht, sind beide Punkte gleich. Die Funktion stellt die direkte Bewegung von Start- zum Endpunkt mit konstanter Geschwindigkeit dar. Ebenso kann man auch sich bewegende Linien, Gebiete oder Punktmengen darstellen.

Datentypen für einzelne Units finden sich in der `TemporalUnitAlgebra`. Komplette Bewegungen sind in der `TemporalAlgebra` zu finden.

Momentan besteht in der Javagui die Möglichkeit, solche Objekte im Hoese-Viewer als Animation darzustellen. Die animierte Darstellung ist recht anschaulich, erlaubt jedoch kaum eine visuelle Analyse der Beziehungen zwischen den vollständigen Objekten. Um dies zu ermöglichen, kann man die zeitliche Komponente solcher Objekte als 3. Dimension auffassen und somit solche Objekte im 3D-Raum als Ganzes darstellen.

So kann ein sich bewegender Punkt als 3-dimensionale Linienzug dargestellt werden. Ein sich zeitlich veränderndes Gebiet bildet einen Körper in der 3D-Visualisierung.

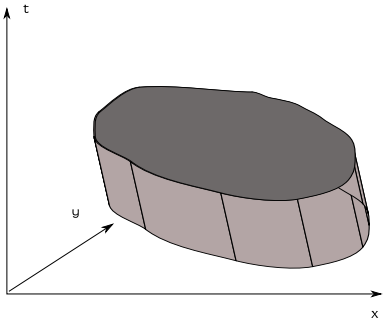

Abbildung 1: Darstellung eines sich bewegenden Gebiets im R^3

In dieser Aufgabe ist ein neuer Viewer zu erstellen, der eine 3-dimensionale Darstellung räumlicher und raum-zeitlicher Objekte ermöglicht.

3.2 Technik

Zur Darstellung 3-dimensionaler Objekte sollen Sie **keine** eigene 3D-Engine entwickeln, sondern das für viele Platformen zur Verfügung stehende Java3D verwenden. Dieses können Sie unter:
`http://java.sun.com/javase/technologies/desktop/java3d/`
kostenlos herunterladen. Machen Sie sich zunächst mit den Funktionen des Pakets vertraut – führen Sie ggf. erste Schritte außerhalb der Javagui durch, damit Sie sicher sein können, dass die Installation ordnungsgemäß erfolgt ist.

3.3 Aufgabenbeschreibung

In dieser Aufgabe ist ein Viewer für SECONDOs Javagui zu entwickeln, der räumliche und raum-zeitliche Objekte 3-dimensional darstellt. Die Menge der darzustellenden Objekte soll dabei nicht begrenzt sein, sondern (ähnlich wie beim HoeseViewer) durch Implementieren neuer Display- Klassen erweitert werden können. Neben der graphischen Aufbereitung sind Objekte auch durch einen kurzen Text in einem separaten Bereich darzustellen. Dabei ist es möglich, auch solche Objekte textuell zu beschreiben, die keine graphische Entsprechung haben.

Zusätzlich zur Implementierung des Viewers sollen in dieser Aufgabe Display-Klassen für die folgenden Typen bereitgestellt werden:

- Typen der StandardAlgebra
- Typen der DateTimeAlgebra
- Typen der RelationAlgebra
- Typen der SpatialAlgebra
- Typen der TemporalAlgebra
- Typen der TemporalUnitAlgebra
- Typen der MovingRegionAlgebra

3.3.1 Basisfunktionalität

Im 3D-Bereich des Viewers ist es möglich, die Ansicht beliebig zu manipulieren. Dazu soll man seinen Standpunkt, die Blickrichtung sowie den Zoom einstellen können. Falls man sich mal „verlaufen“ haben sollte, kann man über einen Knopfdruck zu einer Standardansicht wechseln.

Wird im Textbereich des Viewers ein Objekt ausgewählt, so wird dieses Objekt auch in der 3D-Ansicht hervorgehoben. Falls möglich, soll ebenfalls die andere Richtung implementiert werden.

Selektierte Objekte können ausgeblendet werden, damit die Sicht auf andere Objekte freigegeben wird. Natürlich lassen sich ausgeblendete Objekte auch wieder in die graphische Darstellung aufnehmen. Das ist für einzelne Objekte und für alle ausgeblendeten Objekte möglich. Weiterhin ist ein Aus- und Einblenden der Objekte eines einzelnen Anfrageergebnisses (z.B. Relation) möglich.

Die zeitliche Dimension ist skalierbar, d.h. eine Angabe, welches räumliche Ausmaß eine Zeiteinheit haben soll, ist an der Oberfläche einstellbar. Das Verändern dieses Parameters zieht die Aktualisierung der graphischen Darstellung nach sich.

3.3.2 Aussehen von Objekten

Die Darstellung von Objekten wird durch Kategorien gesteuert. Für graphische Objekte sind (mindestens) die folgenden Parameter einstellbar:

- Linienstärke
- Linienfarbe

- Farbe/Textur der Oberfläche
- Gitter- oder solide Darstellung (einstellbare Transparenz wäre wünschenswert).

Bei einfachen räumlichen Objekten fehlt die Information für die zeitliche Dimension. Hier ist zusätzlich anzugeben, wo sich das Objekt befinden soll. Mögliche Werte sind:

- ein fester, einzugebener Zeitpunkt
- erstes Auftreten irgendeines raum-zeitlichen Objekts[1] (+/- Zeitdauer)
- letztes Auftreten irgendeines raum-zeitlichen Objekts (+/- Zeitdauer)
- komplettes Zeitintervall, d.h. das Objekt wird als unverändertes raum-zeitliches Objekt dargestellt. Die Definitionszeit entspricht der Gesamt-Definitionszeit aller enthaltenen raum-zeitlichen Objekte

Eine spezielle Art einer Kategorie ermöglicht es, räumliche und raum-zeitliche Objekte, die in einer Relation gespeichert sind, in Abhängigkeit eines Attributwerts des gleichen Tupels darzustellen. Hier sind mindestens die Farbe und die Linienstärke als Kategorieparameter wählbar. Dabei soll nicht eine feste Typmenge unterstützt werden, sondern die jeweiligen Displayklassen geben Auskunft über die Möglichkeit, die Darstellung graphischer Objekte zu manipulieren (siehe auch Interface RenderAttribute des HoeseViewers).

3.3.3 Label

In der 3D-Darstellung ist es möglich, die enthaltenen Objekte mit einer Beschriftung zu versehen. Diese kann durch den Benutzer manuell zugewiesen werden. Weiterhin ist es möglich, das Label automatisch aus dem Abfrageergebnis zu generieren. So lässt sich beispielsweise ein Attribut *Name* einer Relation als Beschriftung auswählen. Die Typen, die als Label dienen können, sind nicht fest im Viewer eingebaut. Vielmehr berichtet die Displayklasse, ob sie in der Lage ist, als Label zu dienen (Realisierung über *Interface*).

3.3.4 Dokumentation

Neben der eigentlichen Implementierung ist eine Dokumentation des Viewers zu erstellen. Diese umfasst:

1. Eine Anleitung zur Installation von Java3D,
2. Eine Anleitung zur Einbindung des Viewers in eine bestehende Javagui,
3. Ein Benutzungshandbuch,
4. Die verwendeten Konzepte.

Weiterhin ist der gesamte Quellcode mit passenden `javadoc` Kommentaren zu versehen.

[1] Solange kein raum-zeitliches Objekt vorhanden ist, wird ein Default-Zeitpunkt verwendet.

4 Richtlinien für die Teamarbeit der Phase 2

1. Die Systementwicklung sollte in mehrere Etappen aufgeteilt werden. Am Ende jeder Etappe (auch schon der ersten) sollte ein lauffähiges Teilsystem vorliegen. In Anbetracht der Gesamtzeit von etwa zwei Monaten sollten die Etappen eine Länge von 2-3 Wochen haben.

2. In der 2.Präsenzphase legen die Gruppen die Etappen und die Aufgabenverteilung fest. Aufgaben sollten möglichst so verteilt werden, dass jeder in jeder Etappe etwas beitragen kann. Die Etappen sind entsprechend zu planen.

3. Jedes Team sollte einen Gruppensprecher bzw. eine Gruppensprecherin auswählen, der jeweils zu den Etappenendterminen dem Betreuer über die Erreichung der Ziele berichtet. Das Teilsystem der Etappe n liegt jeweils auf dem CVS-Server vor und kann vom Betreuer getestet werden.

4. Das Team ist gemeinsam für die Lösung der gestellten Aufgabe verantwortlich. Sollten Mitglieder des Teams nicht angemessen mitarbeiten bzw. an der Kommunikation teilnehmen, sollte zunächst versucht werden, das Problem innerhalb des Teams zu lösen. Falls das nicht gelingt, ist der Betreuer zu informieren.

5. Die letzte Etappe endet spätestens am 5.2.2010. Zu diesem Zeitpunkt sollte die Softwareentwicklung abgeschlossen sein. In der Woche zwischen dem 5.2.2010 und dem 13.2.2010 werden die Betreuer das entwickelte System ausprobieren. Ggf. können noch von den Betreuern entdeckte Fehler korrigiert werden.

6. Die letzte Woche sollte von den Teilnehmern dazu genutzt werden, die Abschlußpräsentation vorzubereiten. In der Abschlußpräsentation sollten einerseits die Konzepte der Implementierung erklärt werden, andererseits natürlich das System vorgeführt werden. Für jede Präsentation steht etwa eine Stunde zur Verfügung. Bitte achten Sie darauf, dass genügend Zeit bleibt, das System tatsächlich vorzuführen.