Kapitel 2

Implementation

Ein Großteil des Arbeitsaufwandes für diese Diplomarbeit floß in die Implementierung des EM-Kontur-Algorithmus. Es wurde entschieden, den Algorithmus in ein Rahmenprogramm einzubauen und kein isoliertes Programm aufzubauen. Dies hat folgende Vorteile:

- Die vielen mit der Fahrzeugverfolgung verbundenen Teilalgorithmen müssen nur einmal implementiert werden.
- Das EM-Verfolgungsverfahren kann durch anderen Verfolgungsverfahren ersetzt werden, um Vergleiche anstellen zu können, deren Ergebnisse garantiert auf Algorithmusunterschieden und nicht etwa auf Implementierungsfehlern im Rahmenprogramm basieren.
- Bei kooperativer Programmierung sinkt der Gesamtaufwand.

2.1 Entwurfsrichtlinien

Das verwendete Rahmenprogramm heißt `Motris` (Model-Based-TRacking in Image Sequences) und wird zur Zeit am IAKS entwicke lt. Im Rahmen dieser Neuentwicklung hatte man die Gelegenheit, grundsätzliche Entwurfsentscheidungen zu treffen, welche im folgenden angeführt werden:

- Kapselung: Dieses wichtigste Ziel aus der Softwaretechnik (vgl. [Balzert 2000]) besagt, daß Programmcode und Variablen soweit wie irgend möglich lokal innerhalb eines Funktionalitätsblocks verwaltet werden und daß die Kommunikation mit anderen Funktionalitätsblöcken lediglich über spezifizierte Schnittstellen erfolgt. Nur so ist es möglich, daß der Implementationsaufwand eines komplexen Programms annähernd linear mit der Funktionszahl wächst.

- Modularität: Weiterhin definiert man für Funktionalitäten lediglich die Schnittstelle mit anderen Programmteilen, nicht die konkrete Implementation dieser Funktion. Auf diese Weise kann man Algorithmen des gleichen Typs auswechseln, ohne den Rest des Programmes bearbeiten zu müssen. Beispiele für solche auswechselbare Module sind:
  - Minimierungsverfahren: Gauß-Newton, Levenberg-Marquardt[1]
  - Minimierungsprobleme: EM-Kontur-Quadratminimierung (Formel 1.28), Kantenelement-Anpassung, OF-Anpassung.
  - Bewegungsmodell: Für Fahrzeuge [Pece & Worrall 2002], [Koller 1992], [Schwarz 1997; Leuck 2000] sowie für Menschen [Reuter 2003].
  - Modelle: Hierarchische Baumstruktur aus Volumenprimitiven,
  - Volumenprimitive: Verallgemeinerte Kegelstümpfe und Autos[2].
  - Bildimport: Es werden die Dateiformate `.pm, .pgm, .bmp, .jpg, .pict, .png, .psd, .tga, .xpm, .xbm` und `.pcx` unterstützt.
  - Bildexport: Es werden die Dateiformate `.ps, .eps, .pdf, .bmp, .jpg, .pict, .png, .psd, .tga, .xpm, .xbm` und `.pcx` unterstützt.

  Es ist also z.B. möglich, eine Bildsequenz aus .jpg-Dateien zu laden, dort einen Autozustand mit den Formeln aus [Pece & Worrall 2002] zu prädizieren, den Zustand mit einem kombinierten OF- und EM-Verfahren anzupassen und die Konvergenz mit einem Levenberg-Marquardt-Verfahren zu beschleunigen[3].
- Erweiterbarkeit: Dem Modulsystem können jederzeit weitere Module hinzugefügt werden, z.B. weitere Volumenprimitive wie eine Autobatterie oder ein Minimierungsproblem zur Stoßdämpferabdeckkappendeckellageschätzung.

Ausführungsgeschwindigkeit hingegen ist zunächst von untergeordneter Bedeutung, da es sich um ein akademisches System handelt, dessen Ziel nicht in erster Linie darin besteht, Fahrzeugverfolgung in Echtzeit zu ermöglichen, sondern verschiedene Algorithmen schnell zu implementieren, zu verändern und deren Einsatzbreite sowie Robustheit zu bewerten.

Programmiersprache von `Motris` ist Java, um das Programm einfach auf verschiedenen Platformen demonstrieren zu können.

[1] Levenberg-Marquardt ist eine Verallgemeinerung von Gauß-Newton, entspricht also diesem bei entsprechender Parameterwahl. Trotzdem wurden beide Verfahren separat implementiert, um eventuelle Implementierungsfehler schneller zu erkennen.

[2] In der Implementation ist dies ein Volumenprimitiv, da ein solches als Objekt mit Funktionen zur Bestimmung von Kanten, konvexen Oberflächenteilen und Schattenpunkten definiert wurde.

[3] Dies ist das Entwurfsziel des Programms – zur Zeit ist die Kombination verschiedener Aktualisierungsverfahren wie OF+EM noch nicht möglich, aber leicht nachrüstbar.

2.2 Entwicklerhandbuch

2.2.1 Grobaufteilung des Programms

Abbildung 2.1 zeigt ein UML-Diagramm der Grobstruktur von `Motris`. Oben in der Mitte sieht man die Klasse `Motris`, die die `main`-Prozedur enthält, welche beim Start des Programms ausgeführt wird. In dieser Klasse verzweigt sich die Struktur von `Motris` zum Einen in die Graphische Benutzungsschnittstelle (Graphical User Interface, GUI) und zum Anderen in die Daten- und Algorithmenstruktur.

Diese Dreiteilung ist ein typisches Entwicklungsmuster ("Pattern") der Softwaretechnik, genannt Modell-Anzeige-Kontrolle (Model-View-Controller, [Balzert 2000]):

- Das Modul *Modell* beinhaltet Daten sowie Funktionen, um diese Daten zu bearbeiten.
- Die *Anzeige* visualisiert die Daten ohne sie zu verändern. Es kann mehrere Visualisierungen für die selben Daten geben.
- Die *Kontrolle* gehört ebenfalls zur Graphischen Schnittstelle, enthält aber Funktionen um das Modell zu manipulieren. Zu einem Modell kann es ebenfalls mehrere Kontrollen geben, z.B. einen Parameterdialog, mit dessen Hilfe man die Position und Ausrichtung eines Fahrzeugs numerisch eingeben kann, sowie eine Werkzeugleisten-Funktion, mit der man diese Werte per Mausklick auswählen kann.

In beiden Zweigen gibt es zunächst eine Verwaltungsinstanz für eine Menge von Experimenten, *ExperimentSet* auf der Modellseite und *ExperimentSetViewController* auf der Anzeige-/Kontroll-Seite.

Ebenso ist jedes Experimente im *ExperimentSet* sowohl durch eine Modellklasse *Experiment* als auch durch eine Anzeige-/Kontrollklasse *ExperimentViewController* repräsentiert.

Die Klasse *Experiment* ist eine zentrale Modellklasse und besitzt Verweise auf weitere wichtige Programmteile, die in den folgenden Abschnitten im Detail erläutert werden.

- Die *AlgorithmFactory* verwaltet Algorithmen und Algorithmenparameter.
- Die *SceneDescription* verwaltet das Modell der Szene, inklusive Agenten (Autos, Menschen,...) und Geometrieberechnungen.
- Der *DataManager* verwaltet die Bilddaten und davon abgeleitete Werte, die als bildartige Felder gespeichert werden, z.B. Optische-Fluß-Bilder oder Kantenbilder. Weiterhin beinhaltet er eine Referenz auf das Kamera-Objekt, welches Kameraparameter und Projektionsmethoden enthält. Diese Projektionsmethoden werden vom Modul *SceneDescription* aufgerufen.

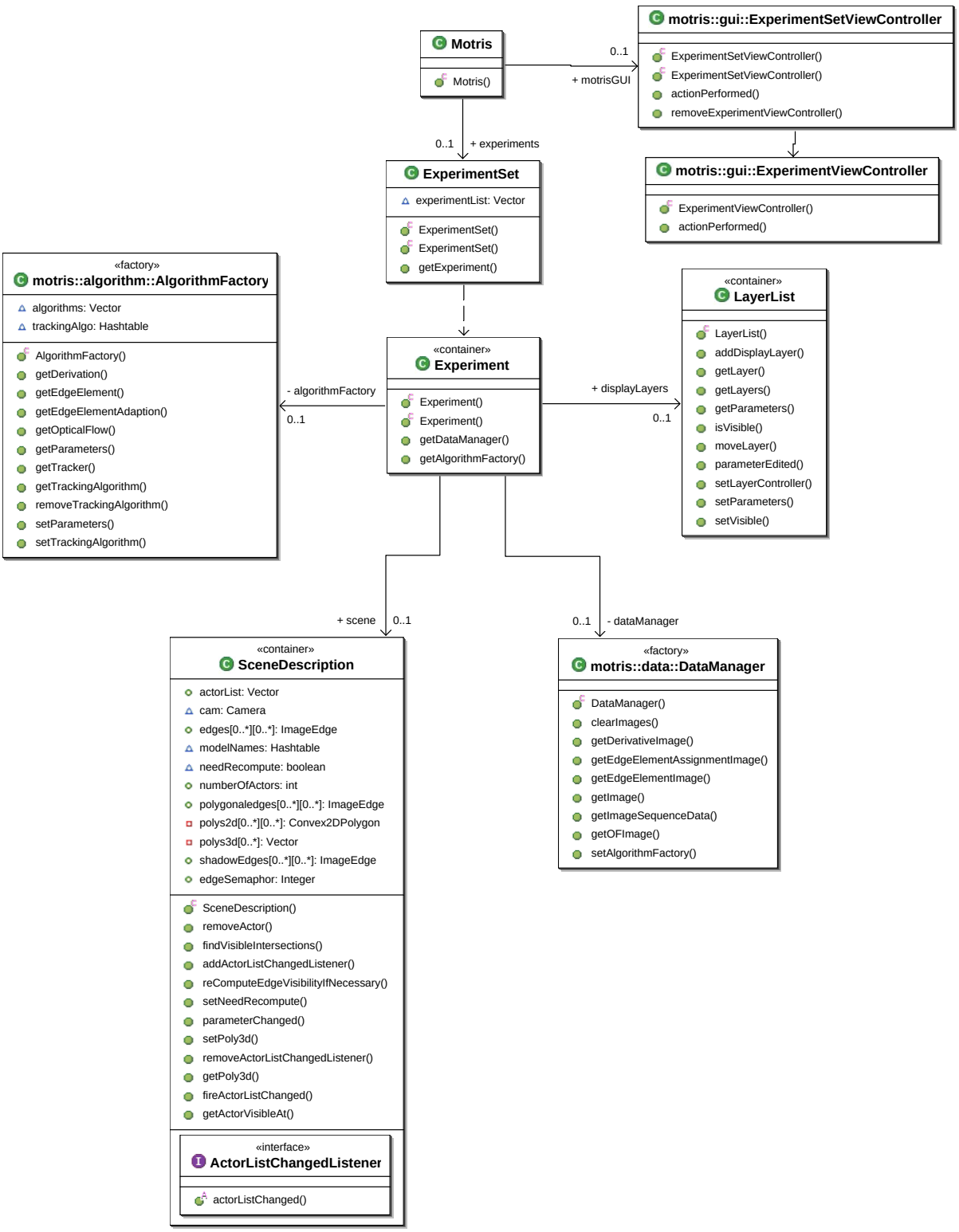

Abbildung 2.1: UML-Diagramm mit Grobübersicht über `Motris`

- *Layer* sind Anzeigeschichten, durch welche verschiedene Programmteile ihre Prüfinformationen (Debug-Informationen) visualisieren können.

2.2.2 Benachrichtigung bei Änderung

`Motris` verfügt über ein Benachrichtigungsverfahren, um eine Datenänderung an alle abhängigen Module zu propagieren. Wenn sich also z.B. der Fahrzeugzustand ändert, so wird diese Änderung direkt an die folgenden Module weitergeleitet:

- Die *SceneDescription* berechnet die Szenengeometrie neu (durch Aufruf von anderen Klassen), da sich ja ein Fahrzeug bewegt hat und somit die in *SceneDescription* gespeicherte Szene nicht mehr mit dem aktuellen geschätzten Modell übereinstimmt.
- Die *ParameterViewController*, bei denen die Auswahlbox "Update values in this window if handled Object changes state" angewählt ist, zeigen die neuen Parameter an (vgl. Abb. 2.2). Wie bereits erwähnt kann es mehrere Anzeigefenster für das gleiche Fahrzeug geben, um unterschiedliche Knoten des Parameterbaumes gleichzeitig anzuzeigen.
- Das *EM-Kontur-Verfahren* berechnet die Position der Abtast- und Meßpunkte neu.
- Der *LayerViewer* zeichnet das Bild (falls vom Benutzer gewünscht) neu und ruft hierfür die sichtbaren Anzeigeschichten auf.

Der Benachrichtigungsmechanismus ist so realisiert, daß sich die abhängigen Module beim Quellmodul mit der Funktion `addParameterChangedListener` registrieren. Sobald dieses dann relevante Daten ändert, ruft es bei jedem registrierten abhängigen Modul eine Benachrichtigungsmethode *parameterChanged* auf.

2.2.3 Parameterverwaltung

Sämtliche Parameter werden in `Motris` über eine einheitliche Schnittstelle verwaltet und abgespeichert. Diese Schnittstelle wurde mit den beiden Zielen entwickelt,

- die Parameterrepräsentation für den Programmierer einfach zu gestalten, damit dieser sie häufig verwendet und Parameter nicht "versteckt", sowie um
- die Parameterverwaltung (eingeben, ändern, laden, speichern) für den Benutzer zu vereinfachen.

Den ersten Punkt erreicht `Motris` durch eine Klasse *Parameter*, welche jeweils einen Parameter von verschiedenstem Typ (reelle Zahlen, ganze Zahlen, boolesche Werte,

Zeichenketten, Matrizen, Dateinamen, Verzeichnisnamen, Auswahllisten und Farben) enthalten kann.

Diese Parameter können außerdem hierarchisch in einem Baum angeordnet werden, den die Klasse *ParameterSet* kapselt. Dies hilft zum einen dabei, die Übersicht über die Parameter zu behalten, und ordnet zum anderen die Parameter verschiedenen Programmteilen zu. So enthalten die Parameter für einen Agenten z.B. zwei Äste für die Parameter des dynamischen Modells (Position, Geschwindigkeit etc) und für die Volumenparameter (Breite, Höhe...). Zusammen mit der graphischen Benutzungsschnittstelle *ParameterViewController* ergibt sich ein mächtiger Mechanismus, der in Abbildung 2.2 und 2.3 zu betrachten ist.

Weiterhin besitzt die *ParameterSet*-Klasse die Möglichkeit, ihre Informationen in einer XML-Datei zu speichern bzw. daraus zu laden. XML eignet sich als Dateiformat besonders, da es plattformunabhängig, einfach zu bearbeiten (mit einem Texteditor) und automatisch in anderen Programmen weiterverarbeitbar ist.

Jedes Modul, das Parameter verwendet, implementiert wahlweise die Schnittstelle *ParameterizedObject* oder erweitert die abstrakte Klasse *ParameterizedObjectAdaptor*. Diese Klassen besitzen die Funktionen `getParameters()` und `setParameters()`, mit denen man die Parameter liest und setzt.

2.2.4 Algorithmen

Die Klasse *Algorithmfactory* verwaltet die Liste der verfügbaren und aktiven Algorithmen. Hierzu erstellt sie auch eine hierarchische Liste der Parameter sämtlicher Algorithmen, sodass diese Parameter an einer Stelle zentral gesammelt werden. Dies vereinfacht die Abarbeitung umfangreicher Experimentserien, da sämtliche Parameter in einer XML-Datei abgelegt und mit einem Mausklick geladen werden können.

Abb. 2.2 zeigt diese Parameterliste, wie sie dem Benutzer präsentiert wird.

Sämtliche vorkommenden Algorithmen stammen von der abstrakten Basisklasse *Algorithm* ab, welche zum einen einen Zeiger zur Klasse *Experiment* für die notwendigen Datenverweise besitzt, zum anderen auch ein *ParameterizedObject* ist.

Eine Unterklasse von *Algorithm* ist *MinimizationProblem*, welche zusätzlich eine Funktion `calculateResidual()` beinhaltet. Diese Klasse wird von jedem Algorithmus erweitert, der als zu minimierende Funktion formulierbar ist und einen Gradient sowie eine Hessematrix zur Minimierung anbietet. Beispiele für solche Minimierungsprobleme sind sämtliche Anpassungsverfahren (EM-Kontur, Kantenelement, OF).

Um ein solches *MinimizationProblem* zu lösen existieren verschiedene Verfahren, z.B. das Levenberg-Marquardt-Verfahren sowie ein Spezialfall davon, das Gauß-Newton-Verfahren. Alle diese Algorithmen erweitern den *Minimizer*, welcher eine Referenz auf das Problem bekommt und darauf seine Minimierung anwendet.

Verwendet wird der *Minimizer* z.B. im *Tracker*, der einen bestimmten Agenten mit

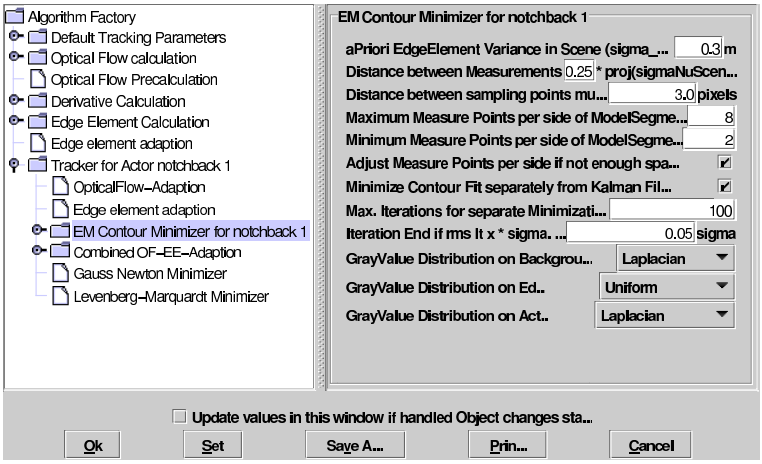

Abbildung 2.2: Die Klasse *ParameterViewController* stellt eine Oberfäche bereit, mit der sich ein hierarchisch angeordneter Satz von Parametern verschiedenster Typen bearbeiten läßt. Hier: Parameter der *AlgorithmFactory*

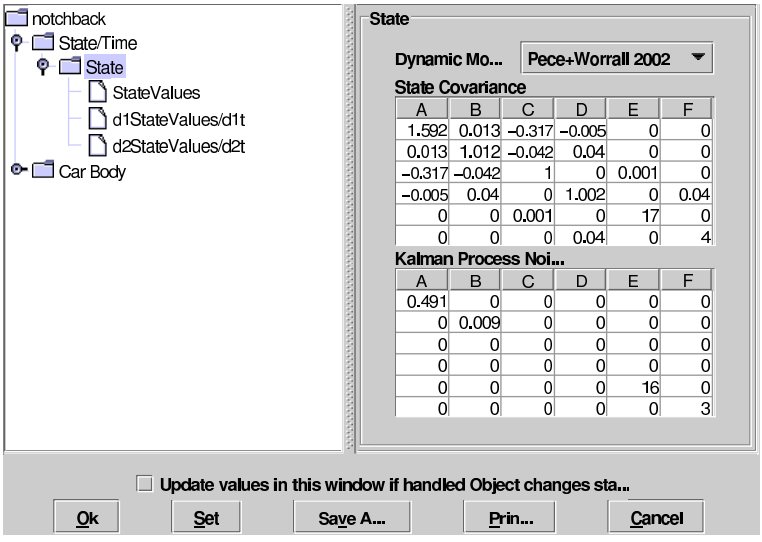

Abbildung 2.3: *ParameterViewController* zeigt Agentenparameter an. Der Name des angezeigten Agenten findet sich dabei in der Wurzel des Parameterbaumes (hier “notchback”)

einem bestimmten Verfahren verfolgt und hierzu die Verwaltungsmethoden enthält, um das Verfahren, den Agenten und die Parameter auszuwählen. Sämtliche für die Fahrzeugverfolgung relevanten Algorithmen sind jedoch im *Minimizer* und *MinimizationProblem* zu finden, sodass die Aufgaben des *Tracker* lediglich die angesprochene Verwaltung umfassen.

2.2.5 EM-Kontur-Algorithmus

Abbildung 2.4 zeigt ein UML-Diagramm des EM-Kontur-Algorithmus. Die dort sichtbare Hauptklasse *EMContourMinimizer* erbt von *MinimizationProblem* (vgl. Abschnitt 2.2.4) und besitzt Referenzen auf die Teilalgorithmen:

- Die *ModelSegmentPointList* kapselt die Liste der Abtastpunkte (*ModelSegmentPoint*s) und berechnet deren Positionen.
- Ein *ModelSegmentPoint* berechnet den Schätzwert für $\hat{\boldsymbol{\nu}}_k$ an seiner Stelle.
- Ein *MeasurePoint* stellt einen Meßpunkt dar und speichert im Wesentlichen dessen Grauwertdifferenz und Position.

Die *GrayvalueDistribution*s sind verschiedene Wahrscheinlichkeitsverteilungen, die beim EM-Kontur-Algorithmus verwendet werden:

- *GrayValueLaplaceDistribution* bestimmt die Wahrscheinlichkeiten für Grauwertdifferenzen an der gleichen Seite einer Grauwertdiskontinuität entsprechend Gleichung (1.12).
- *GrayValueUniformDistribution* und *GrayValueLinearDistribution* bestimmen die Wahrscheinlichkeiten für Grauwertdifferenzen auf der Stelle einer GD entsprechend Gleichung (1.13) und Abschnitt 4.1.2.
- *GrayValueGaussianDistribution* modelliert die A-Priori-Verteilung der GD-Positionen gemäß Formel (1.11).

2.2.6 Agenten

Die Agentenklassen wurden größtenteils von P. Reuter im Rahmen seiner Diplomarbeit [Reuter 2003] implementiert und sind in Abbildung 2.5 dargestellt.

Die Klasse *Actor* modelliert dabei den Agenten. Sie besitzt zum Einen eine Referenz auf den *State* und zum anderen auf das *RigidModel*.

Die Klasse *State* kapselt den Agentenzustand und ist u.a. zuständig für die Speicherung der Zustandsvektoren zu verschiedenen Zeitpunkten sowie für die Prädiktion des nächsten Zustands. Hierzu gehört auch die Fortschreibung der Zustandskovarianz. Die

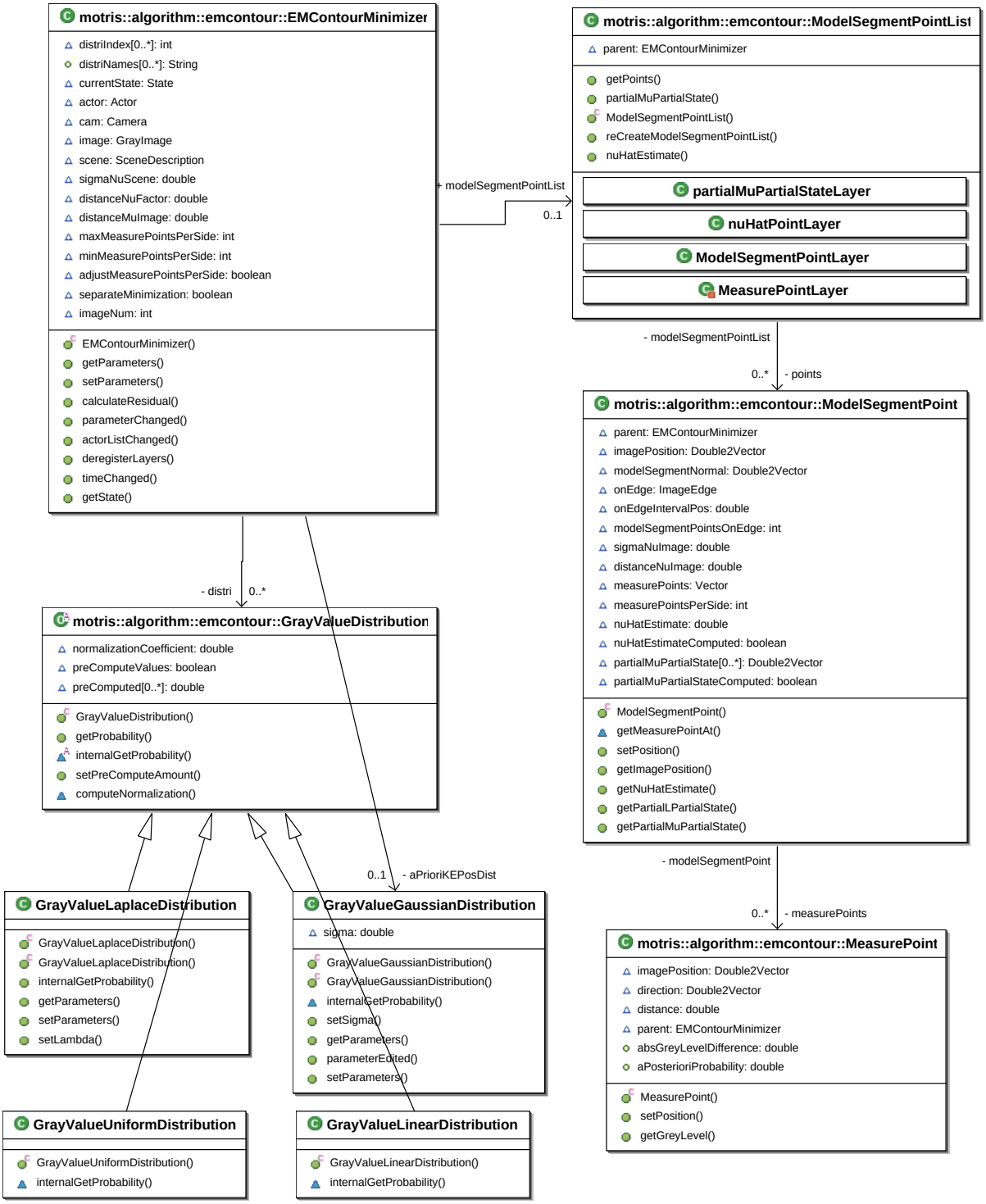

Abbildung 2.4: UML-Diagramm zum EM-Kontur-Algorithmus

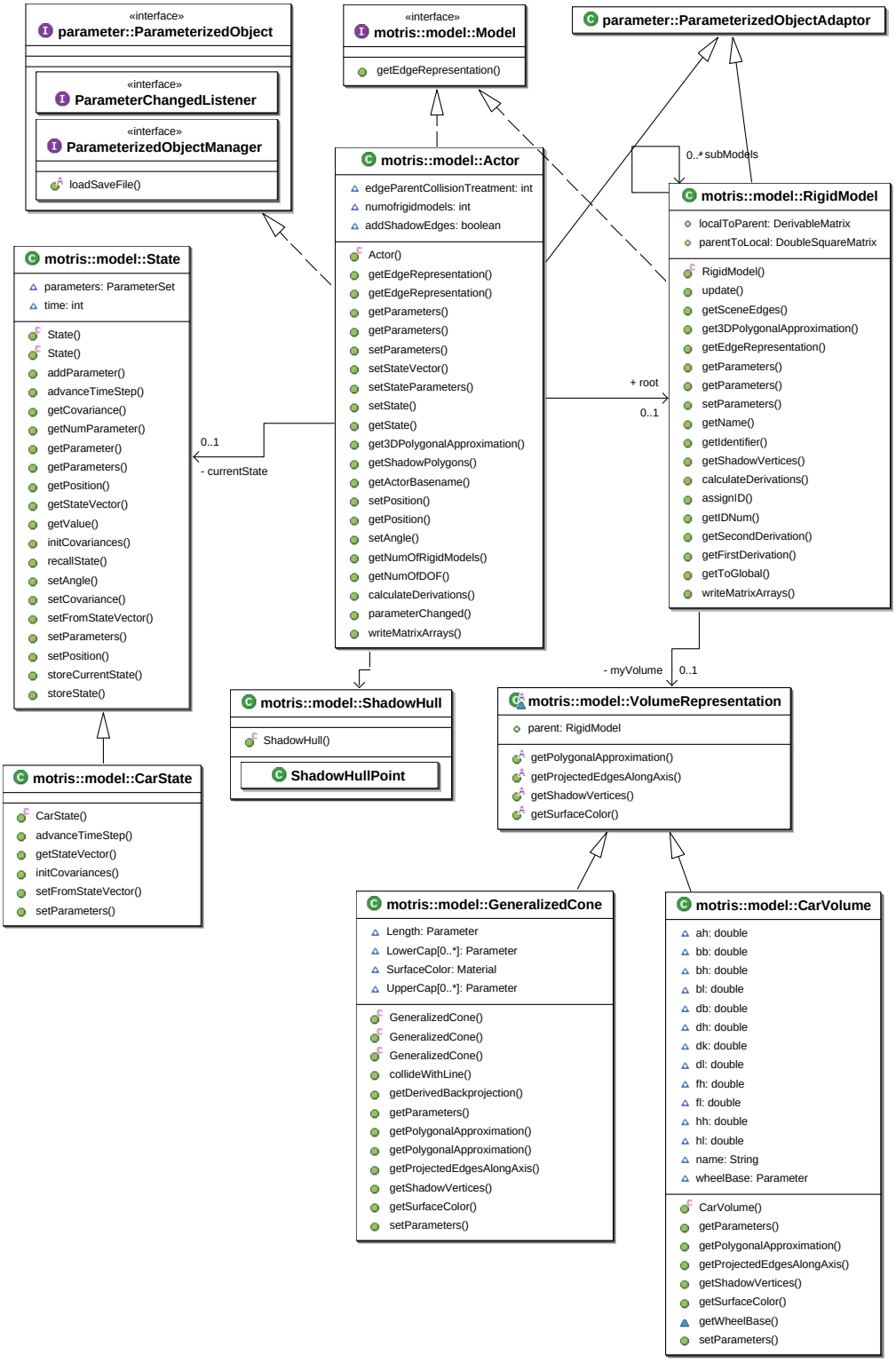

Abbildung 2.5: UML-Diagramm zur Modellierung eines Agenten

Funktionen des *State* sind hierbei allgemeingültig gehalten. Da beim Fahrzeug spezielle Berechnungen nötig sind (z.B. für komplizierte Parameter wie Lenkwinkel oder Geschwindigkeit in Fahrtrichtung), wurde eine Unterklasse *CarState* angelegt, welche in Abschnitt 1.3 beschrieben ist.

Das *RigidModel* repräsentiert eine hierarchische Struktur aus Volumenprimitiven (*VolumeRepresentation*). Für einen Menschen modelliert man hier den Torso, den Kopf sowie die einzelnen Glieder, zur Fahrzeugverfolgung besteht das RigidModel lediglich aus einem Volumenprimitiv, *CarVolume*. Diese Klasse wiederum implementiert das Fahrzeugmodell aus [Koller 1992] mit 16 Eckpunkten, 13 Flächen und 26 Kanten, welches in Abbildung 2.6 dargestellt ist.

2.2.7 Geometrieberechnung

In `Motris` gibt es mehrere Klassen, die jeweils ein geometrisches Objekt repräsentieren:

- *SceneEdge* entspricht einer Kante im 3-Dimensionalen, repräsentiert durch die beiden Endpunkte sowie eine Liste aus *EdgeIntervals*, welche eine Partition des Intervalls $[0, 1]$ aus Intervallen abspeichert und für jedes dieser Teilintervalle angibt, ob es sichtbar ist oder nicht. Wenn man also z.B. eine Szenenkante mit den Endpunkten $(0, 0, 0)$ und $(0, 3, 6)$ annimmt und hierzu das Intervall $[0, 1]$ in ein sichtbares Teilintervall $[0, 2/3]$ sowie ein unsichtbares Teilintervall $[2/3, 1]$ partitioniert, so entspricht dies einer Kante, die von $(0, 0, 0)$ bis $(0, 2, 4)$ sichtbar ist und von $(0, 2, 4)$ bis $(0, 3, 6)$ unsichtbar.

  Zusätzlich sind für den Start- und Endpunkt $n \times 4 - Matrizen$ abgespeichert, die die Ableitung des Punktes nach den n Freiheitsgraden des Agenten enthalten (jeweils 4-dimensional aufgrund der Verwendung homogener Koordinaten).

- *ImageEdge* funktioniert analog zu *SceneEdge*, jedoch werden hier 2-dimensionale Koordinaten im Bild verwendet.

- *Convex3DPolygon* entspricht einem Polygon in der Szene, repräsentiert durch eine Menge von Eckpunkten und einen Normalenvektor.

- *Convex2DPolygon* leistet das gleiche im Bild. Die Methode `collide3D` bestimmt die Verdeckung einer übergebenen *SceneEdge*-Kante bezüglich einer ebenfalls übergebenen Kameraposition und markiert gegebenenfalls nicht sichtbare Kantenteile auf diesem Kantensegment als unsichtbar. Hierzu ruft `collide3D` zunächst `collide2D` auf, um die Intervalle des 2-dimensionalen Schnittes der Linie mit dem Polygon zu bestimmen. Anschließend bestimmt `collide3D` für jedes dieser Intervalle die Verdeckung.

- *Volumerepresentation* modelliert ein Volumenprimitiv. Zur Erweiterung des Programms um ein neues Volumenprimitiv muß man hier Methoden zum Zurückliefern der Kanten und Flächen implementieren. Wenn das Objekt zudem noch

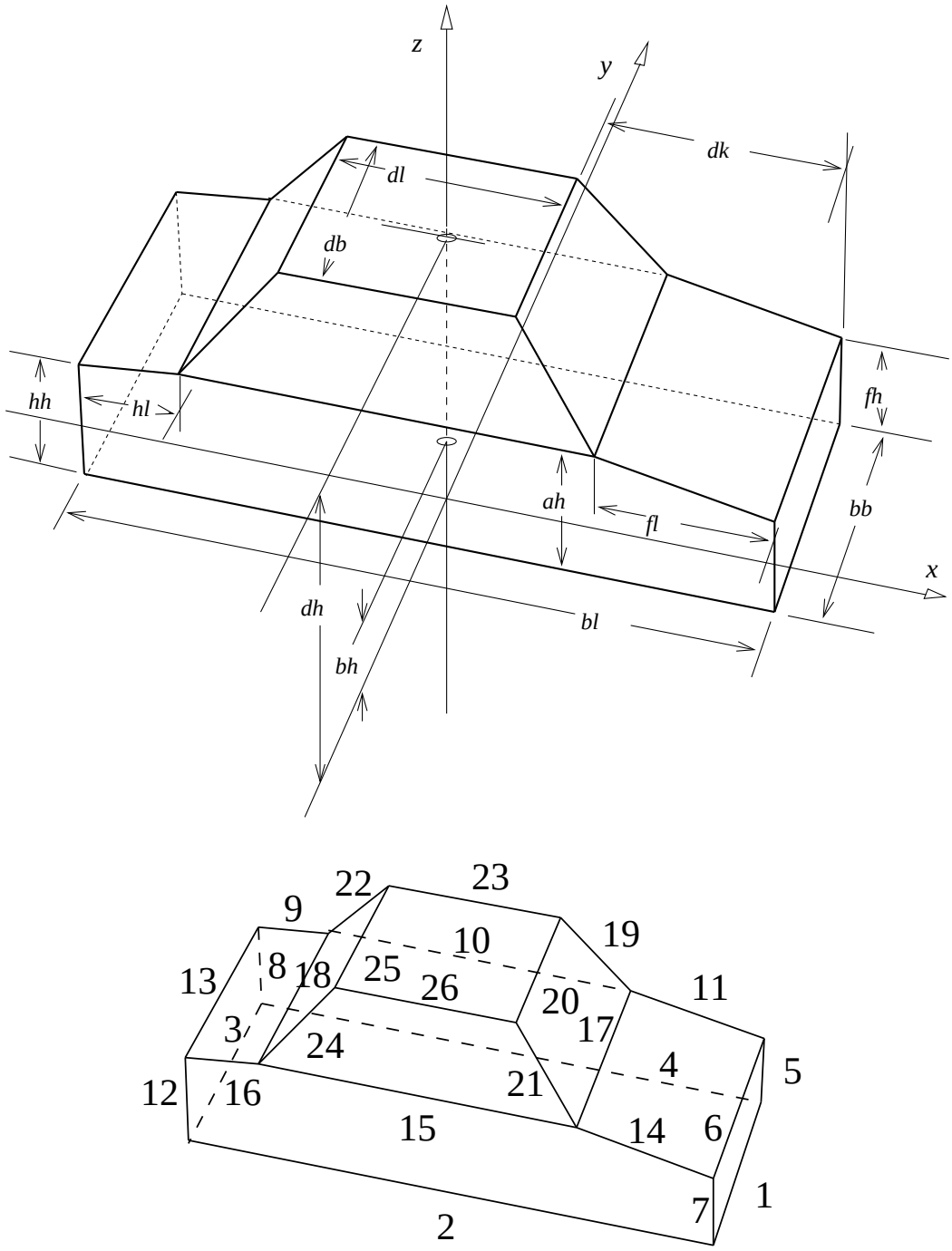

Abbildung 2.6: Fahrzeugmodell aus [Koller 1992] mit Nummerierung der Kanten

Schatten werfen soll, so muß diese Implementation zudem noch die Eckpunkte des Modells zurückgeben.

Eine Klasse zur Punktrepräsentation fehlt und sollte nachgerüstet werden, sobald Programmiererkapazität frei ist, da sich hierdurch manche Methoden eleganter programmieren lassen.

SceneDescription enthält wie bereits angesprochen eine Liste der Agenten. Wenn sich die Szenengeometrie ändert (z.B. wenn Agenten ihre Position ändern), wird *SceneDescription* benachrichtigt und aktualisiert ihre interne Repräsentation der Szene, bestehend aus Listen der Polygone (`poly3d` und `poly2d`) sowie Listen der Kanten (`edges`, `polygonaledges` berechnet aus den Polygonen und `shadow-edges` berechnet aus dem Agentenschatten).

Ein UML-Diagramm mit allen zur Geometrieberechnung relevanten Klassen findet man in Abbildung 2.7.

2.2.8 Anzeigeschichten

Visualisierung von (Teil-)Ergebnissen ist einer der wichtigsten Punkte der Bildauswertung, da der Mensch einen reinen Zahlenwust nur sehr schwer interpretieren kann, während er geometrische Zusammenhänge schnell visuell aufnimmt. Aus diesem Grund enthält `Motris` eine Schichtenarchitektur, die eine Vielzahl von Informationen darstellen. Sie erlaubt dem Benutzer, sich die gewünschten Angaben durch Mausklick auszusuchen, sodass er experimentieren sowie Lösungen nachverfolgen kann ohne den Programmquelltext zu verändern.

Hierzu gibt es eine Liste aus Anzeigeschichten, die einzeln parametrisiert, ein/ausgeschaltet und bezüglich der Anzeigereihenfolge geordnet werden können.

Abbildung 2.8 zeigt ein UML-Diagramm aus allen mithelfenden Klassen sowie allen verfügbaren Schichten.

Am wichtigsten sind dabei folgende Klassen:

- *Layer* ist die Schnittstelle, die von jeder Anzeigeschicht implementiert werden muß. Seine wichtigste Funktion ist dabei `draw`, welche aufgerufen wird, wenn in den übergebenen Graphikkontext *Graphics* gezeichnet werden soll.

  Diese Art der Implementation hat den Vorteil, daß der Graphikkontext nicht nur ein Fensterinhalt sein muß, sondern auch anders aussehen kann: So gibt es im Java-API z.B. einen Graphikkontext, der in Postscript-Dateien schreibt, welche dann z.B. in Diplomarbeiten wie diese eingebunden werden können, um eine vektorisierte, nicht-pixelbasierte Abbildung zu generieren.

- Die abstrakte Klasse *ColorizedLayer* stellt ein paar Bequemlichkeitsmethoden für *Layer* bereit, z.B. die Möglichkeit, Zeichenfarbe und -dicke vom Benutzer einstell-

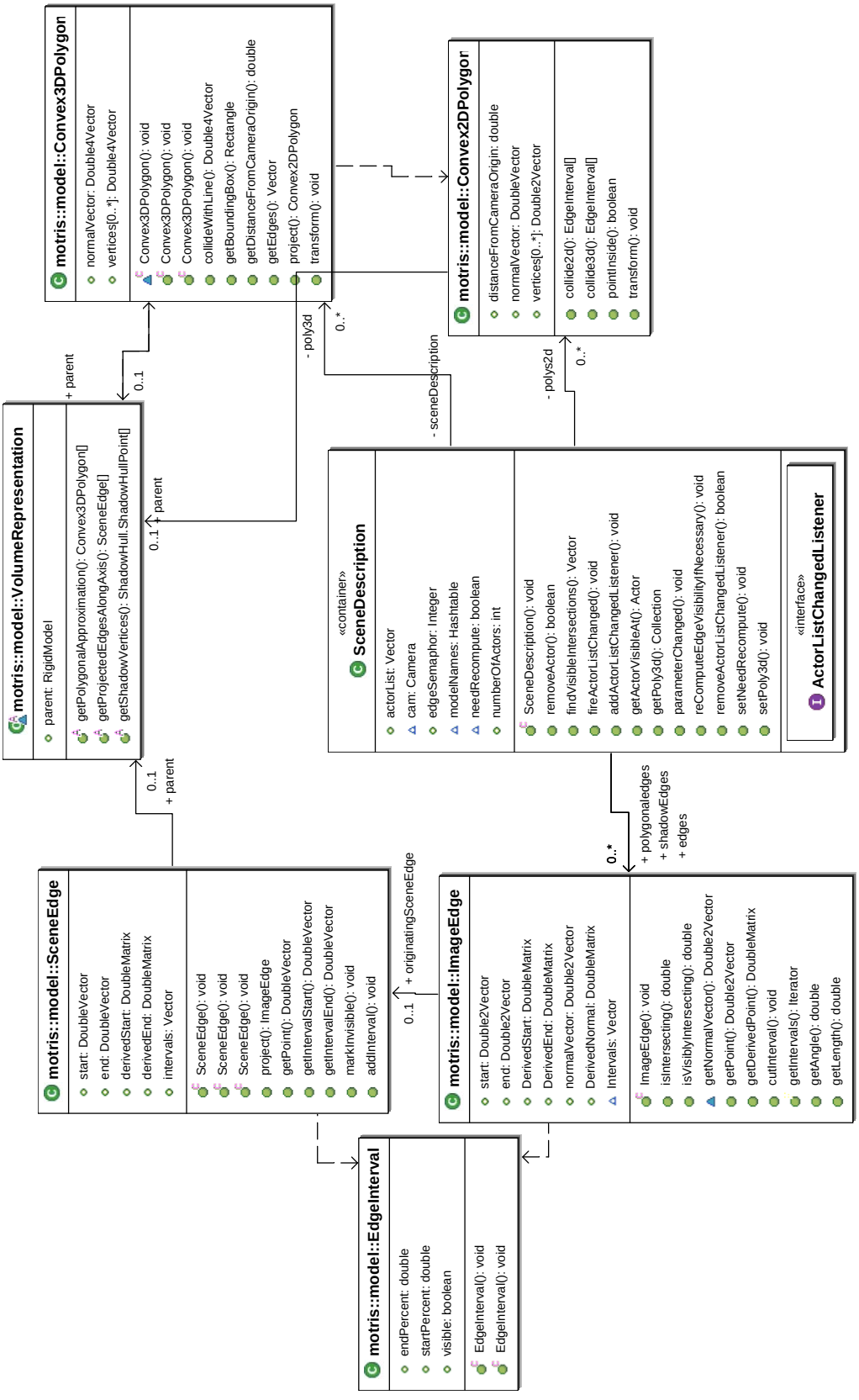

Abbildung 2.7: UML-Diagramm zur Geometrieberechnung

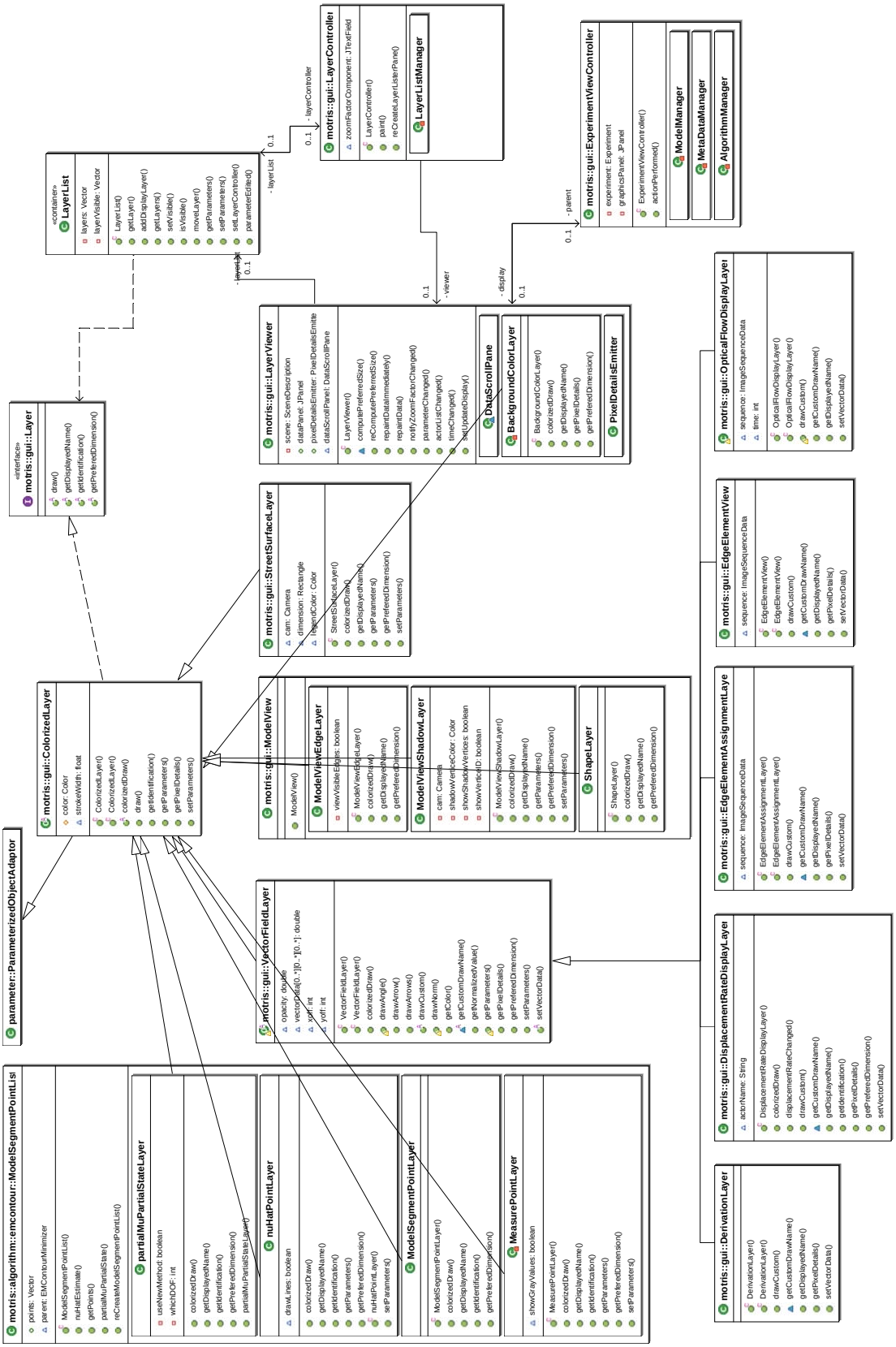

Abbildung 2.8: UML-Diagramm zu den Anzeigeschichten

bar zu halten. Sämtliche zur Zeit verfügbaren Anzeigeschichten erben daher von ColorizedLayer und nicht von Layer.

- Die *LayerList* verwaltet die Liste aller Anzeigeschichten und stellt Methoden bereit, um Anzeigeschichten zur Laufzeit zu registrieren oder zu löschen. Weiterhin speichert die Klasse Informationen über Sichtbarkeit und Zeichenreihenfolge der Schichten.

- Der *LayerViewer* ist ein Java AWT-Panel und kümmert sich um die Anzeige der Schichten, indem er die einzelnen Zeichenmethoden aufruft. Weiterhin verwaltet er den Vergrößerungsfaktor (Zoom) sowie die Bildlaufleisten der Anzeigefläche und reagiert auf Mausereignisse.

- Der *LayerController* ist eine Werkzeugleiste, mit der der Benutzer Reihenfolge, Parameter und Sichtbarkeit der Schichten einstellen kann.

Die restlichen Klassen sind einzelne Schichten, die jeweils Teilinformationen visualisieren.

2.2.9 Graphische Benutzungsoberfläche

Für die graphische Benutzungsoberfläche wurde die Java Swing Bibliothek verwendet, da diese neben der Plattformunabhängigkeit auch einfache Programmierbarkeit und Flexibilität bietet. Die Bildschirmphotos in Abbildung 2.9 und 2.10 zeigen dies eindrucksvoll, indem sie zwei völlig unterschiedliche Ansichten des gleichen Programms zeigen, die ohne Umprogrammierung nur durch Mausklicks individualisiert wurden.

Das UML-Diagramm mit den wichtigsten an der Graphischen Benutzungsoberfläche (GBO) beteiligten Klassen findet sich in Abb. 2.11.

Dort sollte man zunächst die Klasse *ExperimentViewController* beachten, die das Haupt-Experimentfenster beinhaltet. Diese Klasse besitzt neben dem *LayerViewer* eine Reihe von *Controller*n (*Actor-*, *Display-*, *Tracking-* und *ImageSequenceController*). Jeder dieser Controller ist für einen Aspekt des Programms zuständig und besitzt hierzu eine Reihe von *DropDownAction*s als innere Klassen, die jeweils einen Benutzerbefehl kapseln. Die Klasse *ActorController.FindActorAction* z.B. findet den zur Zeit ausgewählen Agent auf dem Bildschirm, zentriert ihn und passt den Bildschirmausschnitt entsprechend an.

Diese Art der Implementation von Befehlen als Action-Klassen führt zu einer höheren Kapselung und damit Wartbarkeit als die klassische Methode, die Befehle via Fallunterscheidung in einer Ereignisbehandlungsmethode auszuführen. Zudem enthält die neu entwickelte Klasse *DropDownAction* nicht nur den Programmcode für die Aktion selber, sondern zudem noch ein ausklappbares Menü, einen Befehlsnamen, ein Icon und ein Tastaturkürzel. Hierdurch kann man eine DropDownAction sowohl in ein Menü als

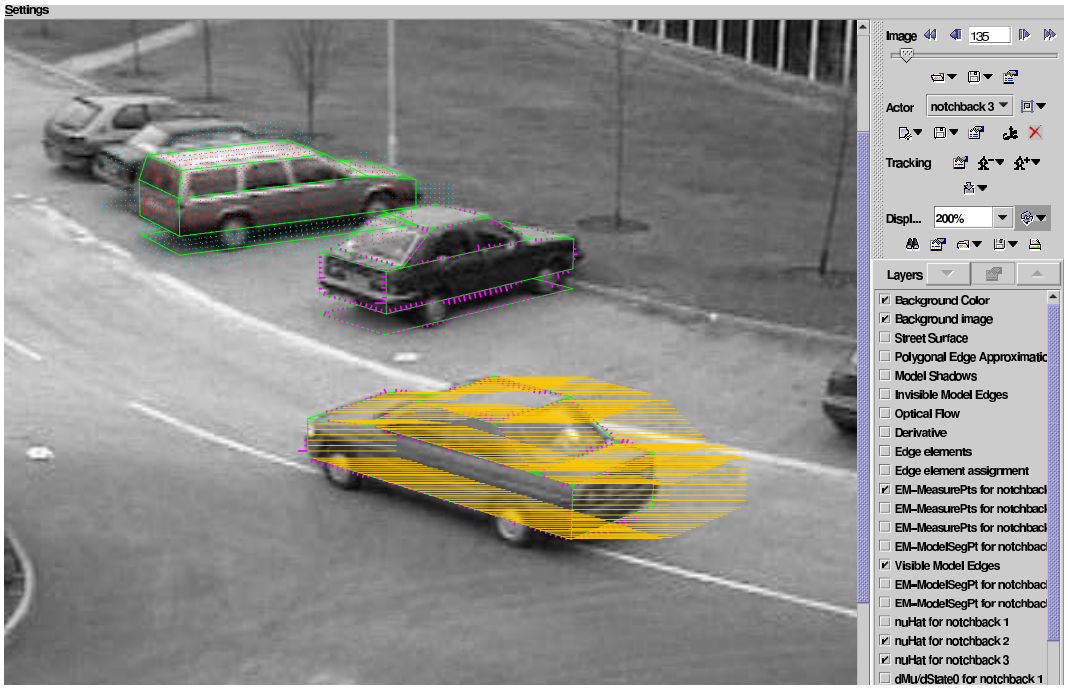

Abbildung 2.9: Das `Motris`-Hauptfenster

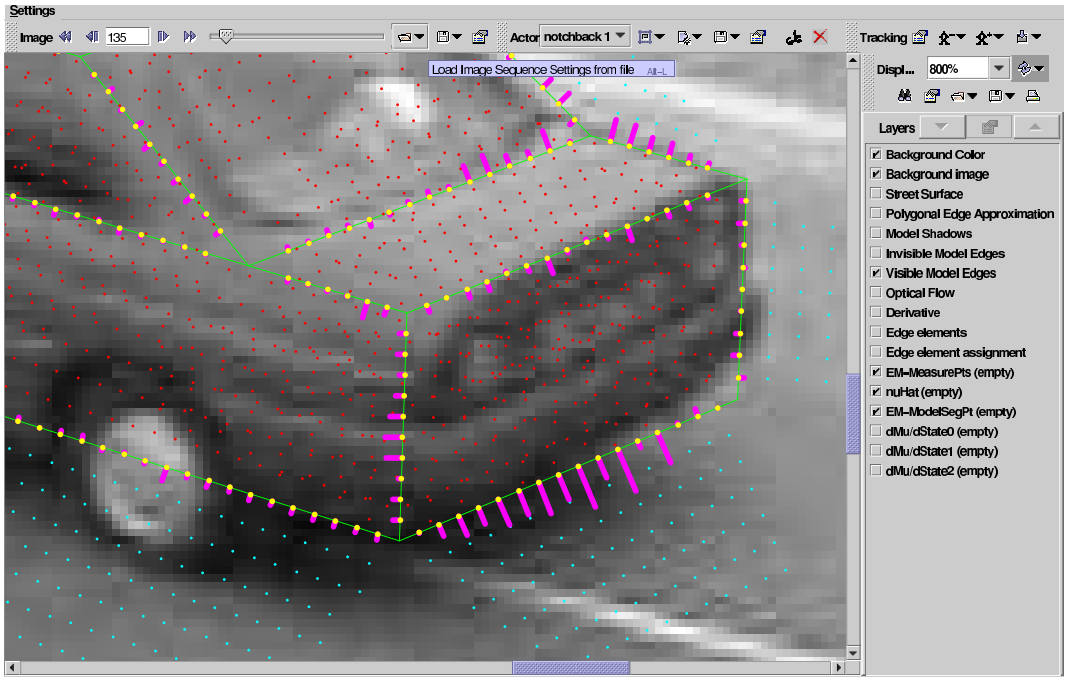

Abbildung 2.10: Weitere Aufnahme des `Motris`-Hauptfensters mit anderen Daten

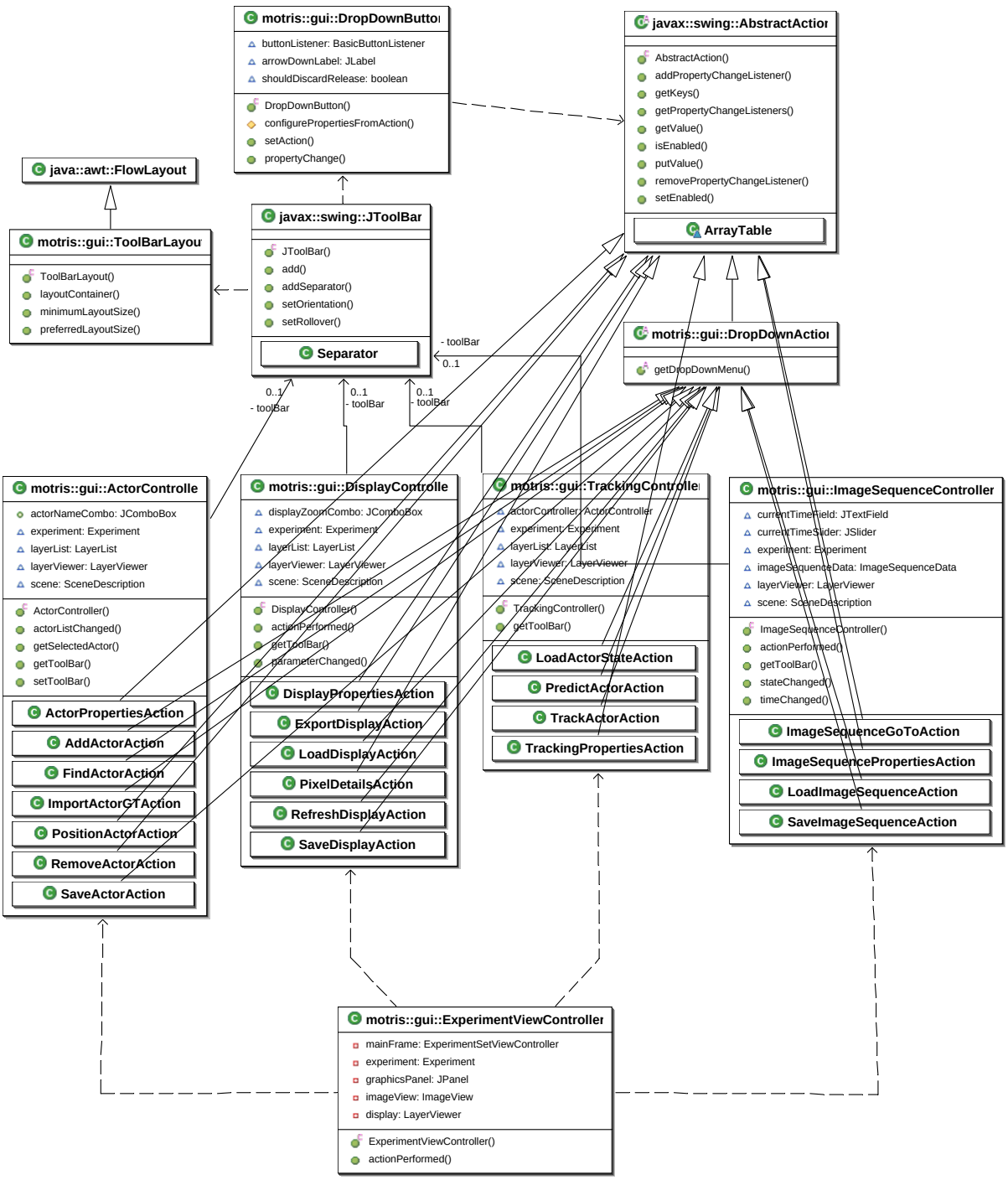

Abbildung 2.11: UML-Diagramm zur graphischen Benutzungsoberfläche

auch in die Werkzeugleiste einfügen ohne Namen etc. an verschiedenen Stellen setzen zu müssen.

Das Einfügen in ein Menü funktioniert dabei ohne Probleme, zum Einfügen in eine Werkzeugleiste musste ein separater *DropDownButton* als Erweiterung des *Button*s entwickelt werden, der das ausklappbare Menü darstellen kann. Schließlich wurde noch das *ToolBarLayout* als Erweiterung des Java-*FlowLayout* entwickelt, welches den Einsatz von mehr als einer Werkzeugleiste pro Seite des Hauptfensters ermöglicht.

2.2.10 Experimentverwaltung

Wie man bei der Arbeit mit dem Programm feststellt, kann die weitgehende Fehlerfreiheit einzelner Funktionen rasch durch interaktive Arbeit mit dem Programm getestet werden, da man sämtliche Programmteile einfach und schnell aufruft sowie visualisiert.

Um jedoch einen optimalen *Parameterwert* herauszubekommen oder die Auswirkung einer geringfügigen Algorithmusänderung einzuschätzen benötigt man Experimente, die eine oder gar mehrere komplette Bildfolgen berechnen und auswerten. Dies kann nicht mehr interaktiv bewerkstelligt werden, da die zur Zeit vorhandene Rechenleistung hierzu nicht ausreicht. Aus diesem Grund wurde eine Experimentverwaltung implementiert, die es ermöglicht, ein komplettes Experiment mit Hilfe einer Datei zu beschreiben. Um dann einen Versuch zu berechnen, liest man diese Datei einfach ein und `Motris` erledigt die Berechnungen, den Bild/Videoexport und die Speicherung der Fahrzeugtrajektorien zur späteren Auswertung.

In der Praxis benötigen Experimente nicht nur unterschiedliche Parameter, sondern auch der Ablauf kann unterschiedlich aussehen: manchmal möchte man die Fahrzeuge von Hand statt automatisch initialisieren, in anderen Fällen möchte man spezielle Ansichten abspeichern und als Video zusammenführen und eventuell will man zwischendurch die Einstellungen modifizieren, wenn sich z.B. Lichtverhältnisse ändern.

.

Um dies alles ohne Umprogrammierung von `Motris` zu erlauben wurde eine einfache Skriptsprache entworfen, in der sich Experimente beschreiben lassen. Diese Sprache setzt auf dem in Abschnitt 2.2.3 beschriebenen Parametermodell und damit auf XML auf. Ein Programm besteht dabei aus vier Blöcken:

- Das `InitScript` führt `Motris` bei Programmstart aus es dient dazu, Parameter zu laden und Fahrzeuge zu initialisieren.
- Das `FinishScript` wird am Ende ausgeführt und kann z.B. Ergebnisse speichern oder visualisieren.
- Das `ActionScript` spezifiziert, in welchen Bildern `Motris` verfolgt, und gibt an, welche Schritte `Motris` für jedes Bild ausführt.

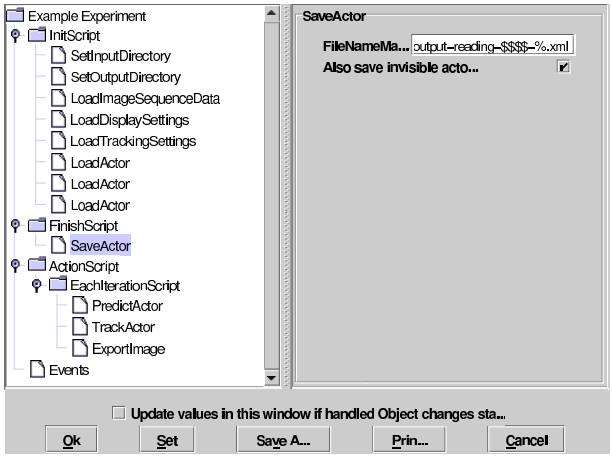

Abbildung 2.12: Ein Beispiel-Skript visualisiert im *ParameterViewController*

- Die `Events` bestehen aus eine Menge von Teilskripten, die `Motris` jeweils bei Erreichen einer bestimmten Halbbildnummer ausführt.

Abbildung 2.12 zeigt ein solches Skript, wie es vom *ParameterViewController* angezeigt wird. Die Programmierung selber erfolgt mit einem Texteditor.

Da der Algorithmus seine Schätzung iterativ erarbeitet, benennt man den Zustandsvektor $\mathbf{x}$ zum Zeitpunkt des Iterationsschritts t mit $\mathbf{x}^{(t)}$ und erhält wegen der strengen Monotonie des Logarithmus:

$$\underset{\mathbf{x}}{\operatorname{argmax}} \frac{P(\Delta\mathbf{i}|\hat{\boldsymbol{\mu}}(\mathbf{x}^{(t)}))}{P(\Delta\mathbf{i})} = \underset{\mathbf{x}}{\operatorname{argmax}} \ln \frac{P(\Delta\mathbf{i}|\hat{\boldsymbol{\mu}}(\mathbf{x}^{(t)}))}{P(\Delta\mathbf{i})} = \underset{\mathbf{x}}{\operatorname{argmax}}\, l(t). \tag{1.41}$$

Zunächst wird dazu ein initiales $\mathbf{x}^{(0)}$ geschätzt. Der Schätzwert wird dann dadurch optimiert, dass der Term

$$l(t) - l(t-1) = ln \frac{P(\Delta\mathbf{i}|\hat{\boldsymbol{\mu}}(\mathbf{x}^{(t)}))}{P(\Delta\mathbf{i}|\hat{\boldsymbol{\mu}}(\mathbf{x}^{(t-1)}))} \tag{1.42}$$

so lange maximiert wird, bis $\sum_{t=1}^{k} l(t) - l(t-1) = l(k) - l(0)$ konvergiert[3].

Die Maximierung eines einzelnen Terms $l(t) - l(t-1)$ geschieht in 2 Schritten:

1. Im Erwartungswert-Schritt (**E-Schritt**) wird $l(t) - l(t-1)$ als Funktion von $\mathbf{x}^{(t)}$ aufgestellt und umgeformt, wobei $\mathbf{x}^{(t-1)}$ als numerische Größe einfließt. Der Name *Erwartungswert*-Schritt kommt daher, dass $l(t) - l(t-1)$ als Erwartungswert ausgedrückt werden kann (vgl. Formel B.30 in der detaillierten Behandlung im Anhang).

2. Im Maximierungs-Schritt (**M-Schritt**) wird $\mathbf{x}^{(t)}$ auf den Wert gesetzt, der $l(t) - l(t-1)$ maximiert.

1.3 Bewegungsmodelle

Ein Bewegungsmodell für Fahrzeuge umfaßt im Rahmen dieser Arbeit fünf Komponenten:

- Der Zustandsvektor $\mathbf{x} \in \mathbb{R}^{\text{xdim}}$, xdim $\in \mathbb{N}$ enthält sämtliche für den Fahrzeugzustand relevanten freien Parameter.

- Der zusätzliche Fahrzeugdatenvektor $\mathbf{y} \in \mathbb{R}^{\text{ydim}}$, ydim $\in \mathbb{N}$ enthält weitere nichtfreie, d.h. zeitlich unveränderte Fahrzeugparameter.

- Die Prädiktionsfunktion $\mathbf{f} : \mathbb{R}^{\text{xdim}} \times \mathbb{R}^{\text{ydim}} \rightarrow \mathbb{R}^{\text{xdim}}$ prädiziert aus einem Fahrzeugzustand zum Zeitpunkt k sowie aus den unveränderlichen Parametern $\mathbf{y}$ und der Realzeitdifferenz Δt einen Fahrzeugzustand zum Zeitpunkt k+1, d.h. $\mathbf{x}_{k+1} = \mathbf{f}(\mathbf{x}_k, \mathbf{y}, \Delta t)$.

[3] Wie in [McLachlan & Krishnan 1997] bewiesen, konvergiert das Verfahren zu einem festen Wert $\mathbf{x}$, der jedoch auch ein nur lokales Optimum sein kann, kein globales sein muß.

- Die Jacobimatrix der Prädiktionsfunktion $\mathbf{f}$ an der Stelle $(\mathbf{x}_k, \mathbf{y})$ wird mit der Matrix F bezeichnet, also $F = \frac{\partial \mathbf{f}}{\partial \mathbf{x}^T}\big|_{(\mathbf{x}_k, \mathbf{y})}$ *und* $F \in \mathbb{R}^{\mathrm{xdim} \times \mathrm{xdim}}$.

- Eine Kovarianzmatrix $Q \in \mathbb{R}^{\mathrm{xdim} \times \mathrm{xdim}}$, die das Zustandsrauschen modelliert, das man bei jeder Prädiktion auf die Zustandskovarianz addiert.

Im Prädiktionsschritt dient die Funktion $\mathbf{f}$ der Prädiktion des Folgezustandes und die Matrix F der Berechnung der nächsten a-priori-Zustandskovarianzmatrix gemäß der Formel

$$\hat{P}_{k+1}^{-} = F \hat{P}_k^{+} F^T. \tag{1.43}$$

Für diese Diplomarbeit wurden drei verschiedene solcher Bewegungsmodelle implementiert. Sie sind in den folgenden Abschnitten erläutert.

1.3.1 Pece 2002

In [Pece & Worrall 2002] entspricht der Fahrzeugzustand $\mathbf{x} = (t_x, t_y, \varphi, v, \dot{\varphi}, a)^T$ genau dem aus Abschnitt 1.2.1. Es gibt einige zeitlich unveränderliche Parameter $\mathbf{y} = (\tau, \sigma_p, \sigma_{\dot{\varphi}}, \sigma_a)$. Die Prädiktionsfunktion lautet

$$\mathbf{f}(\mathbf{x}_k, y, \Delta t) = \mathbf{f}(\mathbf{x}_k) = \begin{pmatrix} t_{x,k} + \Delta t\, v_k \cos\varphi_k \\ t_{y,k} + \Delta t\, v_k \sin\varphi_k \\ \varphi_k + \Delta t\, \dot{\varphi}_k \\ v_k + \Delta t\, a_k \\ \dot{\varphi}_k\, e^{-\frac{\Delta t}{\tau}} \\ a_k\, e^{-\frac{\Delta t}{\tau}} \end{pmatrix}, \tag{1.44}$$

woraus für diese Arbeit die Matrix

$$F = \begin{pmatrix} 1 & 0 & -\Delta t\, v_k \sin\varphi_k & \Delta t\, \cos\varphi_k & 0 & 0 \\ 0 & 1 & \Delta t\, v_k \cos\varphi_k & \Delta t\, \sin\varphi_k & 0 & 0 \\ 0 & 0 & 1 & 0 & \Delta t & 0 \\ 0 & 0 & 0 & 1 & 0 & \Delta t \\ 0 & 0 & 0 & 0 & e^{-\frac{\Delta t}{\tau}} & 0 \\ 0 & 0 & 0 & 0 & 0 & e^{-\frac{\Delta t}{\tau}} \end{pmatrix} \tag{1.45}$$

gewonnen wurde. In [Pece & Worrall 2002] ist diese Matrix jedoch nicht enthalten, es findet sich lediglich in einer Vorversion [Pece & Worrall 2000] eine abweichende Matrix

$$F = \begin{pmatrix} 1 & 0 & 0 & \Delta t\,\cos\varphi_k & \Delta t\,v\sin\varphi_k & 0 \\ 0 & 1 & 0 & \Delta t\,\sin\varphi_k & -\Delta t\,v\cos\varphi_k & 0 \\ 0 & 0 & 1 & 0 & \Delta t & 0 \\ 0 & 0 & 0 & 1 & 0 & \Delta t \\ 0 & 0 & 0 & 0 & e^{-\frac{\Delta t}{\tau}} & 0 \\ 0 & 0 & 0 & 0 & 0 & e^{-\frac{\Delta t}{\tau}} \end{pmatrix}, \tag{1.46}$$

welche dort sowohl für die Kovarianzberechnung als auch für die Prädiktion verwendet wird (d.h. $\mathbf{f}(\mathbf{x}_k, \mathbf{y}, \Delta t) = F\mathbf{x}_k$). An anderer Stelle wird zudem eine alternative Prädiktion eingeführt, nach der

$$\mathbf{f}(\mathbf{x}_k, y, \Delta t) = \mathbf{f}(\mathbf{x}_k) = \begin{pmatrix} \text{(siehe oben)} \\ \dot{\varphi}_k \left(1 - e^{-\frac{\Delta t}{\tau}}\right) \\ a_k \left(1 - e^{-\frac{\Delta t}{\tau}}\right) \end{pmatrix} \tag{1.47}$$

gilt. Auf Nachfrage stellte Arthur Pece klar, es sich bei der veränderten Matrix um einen Dokumentationsfehler handelt, während die Prädiktionsfunktion in der Tat ein Denkfehler war. Da die Zeitkonstante τ das Abklingen der Beschleunigung und Winkelgeschwindigkeit bestimmt, ist natürlich die Version $e^{-\frac{\Delta t}{\tau}}$ korrekt.

Als Zustandsrauschen verwendet man die Matrix

$$Q = \begin{pmatrix} (\sigma_p \sin\varphi_k)^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & (\sigma_p \cos\varphi_k)^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \sigma_{\dot{\varphi}}^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & \sigma_a^2 \end{pmatrix}. \tag{1.48}$$

1.3.2 Koller 1992

[Koller 1992] verwendet gemäß Anhang B.3 seiner Arbeit keine unveränderlichen Parameter (d.h. ydim = 0) sowie den Zustandsvektor $\mathbf{x} = (t_x, t_y, \varphi, v, \dot{\varphi})$. Die Prädiktionsfunktion lautet

$$\mathbf{f}(\mathbf{x}_k, y, \Delta t) = \mathbf{f}(\mathbf{x}_k) = \mathbf{x}_k + \begin{pmatrix} \Delta t\,v_k\,S \\ \Delta t\,v_k\,C \\ \Delta t\,\dot{\varphi}_k \\ 0 \\ 0 \end{pmatrix} \tag{1.49}$$

und deren Jacobimatrix

$$F = \begin{pmatrix} 1 & 0 & \Delta t\, v_k C & \Delta t\, S & \Delta t\, v_k \frac{\partial S}{\partial \dot{\varphi}_k} \\ 0 & 1 & \Delta t\, v_k S & -\Delta t\, C & -\Delta t\, v_k \frac{\partial C}{\partial \dot{\varphi}_k} \\ 0 & 0 & 1 & 0 & \Delta t \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{1.50}$$

mit den Abkürzungen

$$S = \frac{\sin(\varphi_k + \dot{\varphi}_k \Delta t) - \sin \varphi_k}{\dot{\varphi}_k \Delta t}, \tag{1.51}$$

$$C = \frac{\cos(\varphi_k + \dot{\varphi}_k \Delta t) - \cos \varphi_k}{\dot{\varphi}_k \Delta t}, \tag{1.52}$$

$$\frac{\partial S}{\partial \dot{\varphi}_k} = \frac{1}{\dot{\varphi}_k} \left(\cos(\varphi_k + \Delta t\, \dot{\varphi}_k) - S\right) \text{ und} \tag{1.53}$$

$$\frac{\partial C}{\partial \dot{\varphi}_k} = \frac{1}{\dot{\varphi}_k} \left(-\sin(\varphi_k + \Delta t\, \dot{\varphi}_k) - C\right). \tag{1.54}$$

Obwohl dieses Bewegungsmodell mittlerweile im IAKS durch das im nächsten Absatz beschriebene Modell von [Schwarz 1997] ersetzt wurde, macht es dennoch Sinn, dieses Bewegungsmodell einzubeziehen, da die Unterschiede gegenüber dem Modell von [Pece & Worrall 2002] gering, aber interessant sind:

- Pece und Worrall verwenden die Beschleunigung als zusätzliche Variable im Zustandsvektor.
- Koller erreicht eine genauere Prädiktion der Position, da er die Bewegung als Kreisbahn modelliert und berechnet, während Pece und Worrall eine lineare Bewegung ansetzen, bei sie nach jedem Schritt der Winkel anpassen. Dies entspricht einer Bewegung auf einem Vieleck mit der Seitenlänge $\Delta t\, v_k$.

Da die meisten Bildfolgen eine relativ geringe Bewegung pro Δt besitzen, kann man den Einfluß des 2. Punktes vermutlich vernachlässigen, sodass ein Verbesserung oder Verschlechterung bei Wechsel des Bewegungsmodells auf den zusätzlichen Zustandsparameter a zurückzuführen sein wird.

1.3.3 Schwarz 1997

[Schwarz 1997] ersetzt die Winkelgeschwindigkeit $\dot{\varphi}$ im Fahrzeugzustand durch den Lenkwinkel ψ, womit er eine Abhängigkeit der Winkeländerung von der Fahrtgeschwindigkeit erreicht. Dies macht nicht nur intuitiv Sinn (der Lenkwinkel ist schließlich eine

direkt vom Fahrzeugführer manipulierte Stellgröße), sondern man erzwingt hierdurch zusätzlich, dass das Modell für ein stationäres Fahrzeug auch keine rauschbedingte Drehung vollziehen kann. Um das Bewegungsmodell zu beschreiben benötigt man einen zeitlich unabhängigen Parameter, den Achsabstand L. Die Prädiktionsfunktion lautet dann

$$\mathbf{f}(\mathbf{x}_k, y, \Delta t) = \mathbf{f}(\mathbf{x}_k) = \mathbf{x}_k + \begin{pmatrix} \Delta t\, v_k\, \cos(\varphi_k + \psi_k) \\ \Delta t\, v_k\, \sin(\varphi_k + \psi_k) \\ \alpha \\ 0 \\ 0 \end{pmatrix} \tag{1.55}$$

mit der zugehörigen Jacobimatrix

$$F = \begin{pmatrix} 1 & 0 & -\Delta t\, v_k \sin(\varphi_k + \psi_k) & \Delta t\, \cos(\varphi_k + \psi_k) & -\Delta t\, v_k\, \sin(\varphi_k + \psi_k) \\ 0 & 1 & \Delta t\, v_k \cos(\varphi_k + \psi_k) & \Delta t\, \sin(\varphi_k + \psi_k) & \Delta t\, v_k\, \cos(\varphi_k + \psi_k) \\ 0 & 0 & 1 & M & N \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{1.56}$$

und den Abkürzungen

$$\alpha = \arctan\left(\Delta t\, v_k\, \frac{\sin\psi_k}{L}\right), \tag{1.57}$$

$$M = \frac{\Delta t \sin\psi_k}{L\left(1 + \left(\frac{\Delta t\, v_k \sin\psi_k}{L}\right)^2\right)} \quad \text{sowie} \tag{1.58}$$

$$N = \frac{\Delta t\, v_k \cos\psi_k}{L\left(1 + \left(\frac{\Delta t\, v_k \sin\psi_k}{L}\right)^2\right)}. \tag{1.59}$$

[Schwarz 1997] leitet diese Formeln jedoch unter der Voraussetzung her, dass der Fahrzeug-Referenzpunkt sich auf der Vorderachse befindet. Um eine Konsistenz mit den anderen beiden Bewegungsmodellen zu erhalten, bei denen sich der Referenzpunkt auf der Mitte der Bodenplatte befindet, müssen die Formeln korrigiert werden. Der korrigierte Zustandsvektor $\tilde{\mathbf{x}}$ lautet also

$$\tilde{\mathbf{x}} = \mathbf{x} - \frac{L}{2}\begin{pmatrix} \cos\varphi \\ \sin\varphi \\ \mathbf{0}_{3\times 1} \end{pmatrix}, \tag{1.60}$$