ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

Konzeption und Realisierung eines mobilen Frameworks zur markerinduzierten Darstellung von interaktiven 3D-Prozessmodellen

Bachelorarbeitan der Universität Ulm

Vorgelegt von:
Johann Albach
johann.albach@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Rüdiger Pryss

2015

Fassung 3. August 2015

Satz: PDF-LaTeX 2$_\varepsilon$

Kurzfassung

Augmented Reality verknüpft Realität mit virtueller Information. Schnellere Hardware in mobilen Geräten ermöglicht Augmented Reality Applikationen in mobilen Umgebungen produktiv zu nutzen. Für diese Verknüpfung wird ein mobiles Framework benötigt, welches die Darstellung und Interaktion von markerinduzierter Information ermöglicht.
Im Folgenden wird solch ein Framework konzipiert und realisiert. Die Ziele dieses Frameworks beinhalten die Erkennung von Markern, die Unterscheidung dieser und eine performante Visualisierung von Prozessmodellen mit Interaktionsmöglichkeiten. Das **Processor** getaufte Android-Framework verwendet das **Imagine** Framework [Har13], welches **openCV für Android** für die Bildverarbeitung nutzt, um Marker zu erkennen. **Processor** erweitert die Fähigkeiten des **Imagine** Frameworks und bietet ein solides Rendering System, mit Hilfe von **OpenGL ES 2.0**, für die Visualisierung von **ADEPT** Prozessmodellen [DKR+95] an. Das finale Framework wird zusätzlich mit den Anforderungen verglichen um mögliche Verbesserungen vorzuschlagen.

Danksagung

Natürlich dankt der Autor seinem Betreuer Rüdiger Pryss für seine Unterstützung. Andreas Schmid verdient besondere Danksagung, da ohne ihn, mir die Möglichkeit, dieses Thema auszusuchen, womöglich verborgen bleiben würde. Außerdem danke ich Stefan Dimitrijevic für sein Jahre langes Interesse im Bereich der Computer Grafik und den regen Informationsaustausch, welcher auch mein Interesse für dieses Gebiet aufrecht hielt.

Inhaltsverzeichnis

1

Motivation

Business Process Management war lange Zeit den Desktopumgebungen vorbehalten. In den letzten Jahren tendieren jedoch viele Applikationen zu einem Übergang in die mobile Umgebung. Diese Tendenz zeichnet sich auch im Bezug auf BPM ab [PMR14]. Der Wunsch besteht darin, die Möglichkeit zu besitzen, auf mobilem Weg Prozesse einzusehen, zu organisieren oder bearbeiten zu wollen. Auf Grund von hardwareseitigen Einschränkungen war dies nur Desktop-Computern vorbehalten, doch mobile Geräte sind heute schon in der Lage komplexe Berechnungen durchzuführen und grafisch aufwendige Darstellungen zu zeigen. Deshalb bieten sich Smartphones und Tablets auch für BPM an [PMR13].

Um im mobilen Umfeld mit Prozessen produktiv arbeiten zu können, bedarf es eines Frameworks, welches mit limitierten Bildschirmmaßen umgehen kann und die Interaktion dennoch nicht einschränkt. Dieses Framework wird im Rahmen dieser Arbeit entwi-

ckelt und bietet ebenso unterschiedliche Interaktionsmöglichkeiten für Prozesse oder Prozessabschnitte im dreidimensionalen Raum an.

2

Verwandte Arbeiten

Es existieren zahlreiche BPM Lösungen [DKK14], aber mit dreidimensionaler Visualisierung sinkt das Angebot deutlich. Bereits existierende Arbeiten zur Visualisierung von 3D Business Prozess Modellen können grundsätzlich in drei Bereiche unterteilt werden: (1) Business Prozess Modellierung, (2) 3D Modellierung und (3) Layout-Methoden.
Verschiedenste Methoden zur Modellierung von Business Prozessen mit komplexen Objekten und den jeweiligen Events wurden vorgestellt [DB07, LG07, LOG03]. Jedoch mangelt es diesen an Darstellungsfähigkeiten, vor allem, wenn Information versteckt werden muss, um die Anzahl der sichtbaren Objekte zu reduzieren.
Die Verlagerung von Information ist eine beliebte Technik, um die Komplexität von Geschäftsmodellen zu reduzieren [Sch99, MS06, BRB07]. Um die zusätzliche Information aufzurufen, muss die Ansicht gewechselt werden, was den Benutzer mehr Aufwand kostet. Damit die Information zwischen den Ansichten verknüpft und die Beziehung dazwischen verstanden werden kann, muss zusätzliche Zeit investiert werden.

Einige Tools nutzen die dritte Dimension zur Geschäftsprozessmodellierung [KGM99, BES00, KP04, Rö07, WBR10], um die Integration und Interaktion von neuen Objekten zu unterstützen, wobei diese Tools sich jedoch an Desktop-Umgebungen richten. Der hier vorgestellte Ansatz geht einen Schritt weiter und stellt ein mobiles Framework zur Darstellung von interaktiven 3D-Prozessmodellen vor.
Für die effiziente Darstellung von Prozessmodellen bedarf es eines schnellen Algorithmus, welcher das Layout von Prozessen festlegt. Da der Input aus AristaFlow BPM suite [DKR+95, DR09, DRRM+09] in diesem Rahmen schon einen planaren Graphen liefert, liegt es nahe, einen entsprechenden Algorithmus zu verwenden, welcher auch Überschneidungen der Kanten verhindert. Sugiyama's Algorithmus für Layered Graph Drawing [ESK05] erwies sich als relativ einfach zu implementierender Layout-Algorithmus und zeigte auch die gewünschten Resultate.

3

Anforderungen

Es werden unterschiedliche Anforderung an das, zu entwickelnde, Framework gestellt. Diese sind in zwei Kategorien eingeteilt.

3.1. Notwendige Anforderungen

Die Tabelle 3.1 listet die wichtigsten Anforderungen an das Framework auf, welche benötigt werden, damit es, hinsichtlich des Funktionsumfangs, als vollständig betrachtet werden kann.

Kriterium	Beschreibung
Markererkennung	Marker sollen zuverlässig erkannt und unterschieden werden
Leistungsfähiger 3D-Renderer	Eine schnelle OpenGL ES 2.0 Rendering-Pipeline, welche hunderte Objekte mit flüssiger Framerate zeichnet
Flexibilität bei 3D-Modellen	3D-Modelle sollen flexibel einsetzbar und leicht zu integrieren sein
Einfacher XML-Prozess-Parser	Der Prozess-Parser soll leicht erweiterbar und robust funktionieren
Schnelle 3D-Modell-Generierung	Die 3D-Repräsentation eines Prozesses soll in kurzer Zeit laden und zur Darstellung bereit sein
Interaktion	Verschieben, Rotieren, Zoomen und Auswahl des Prozesses um weitere Information zu erhalten

Tabelle 3.1.: Notwendige Anforderungen, damit das Framework funktional komplett ist

3.2. Zusätzliche Anforderungen

Außerdem gibt es Kriterien, welche nicht zwingend notwendig sind, jedoch trotzdem interessant sind und den Funktionsumfang oder die Bedienung enorm erleichtern und erweitern können. Tabelle 3.2 listet diese auf.

Kriterium	Beschreibung
Pausieren der Markererkennung	Das Pausieren der Markererkennung soll dem Benutzer die Bedienung erleichtern. Die Kamera muss nicht ständig auf die Marker gerichtet werden.
Prozessdaten aus dem Internet	Prozessdaten könnten zusätzlich zum lokalen Speicher auch aus dem Internet geladen werden, damit mehr Flexibilität bei den Einsatzmöglichkeiten entsteht.

Tabelle 3.2.: Zusätzliche Anforderungen, außerhalb des minimal benötigten Funktionsumfangs

4

Architektur

Der Aufbau kann grob in zwei Teile unterteilt werden. Auf der einen Seite arbeitet das leicht modifizierte Imagine-Framework, wobei sich nun einige Komponenten direkt im Processor-Framework befinden. Imagine erstellt, unter Verwendung von OpenCV für Android, einen Kamera-View und versucht aus dem Stream von Kamerabildern Marker zu erkennen und zu unterscheiden.

4.1. Architekturentwurf

Gegenüber von Imagine arbeitet auf der anderen Seite OpenGL ES 2.0 umgeben von einer einfachen Grafik-Engine in Kombination mit einem XML-Parser. Die Grafik-Komponente kümmert sich um die Verwaltung von Grafikobjekten. Dazu gehört sowohl

das Laden von 3D-Modellen, als auch eine Sortierung dieser in eine Datenstruktur, welche eine Leistungssteigerung bewirkt, da die Zugriffszeiten reduziert werden. Der XML-Parser beinhaltet Routinen zum Laden, Parsen und Sortieren von ADEPT-Prozessen, welche als XML-Dateien vorliegen.
Abbildung 4.1 gibt einen groben Überblick über die Komplexität des implementierten Frameworks und Abbildung 4.2 zeigt den Aufbau der Framework-Architektur.

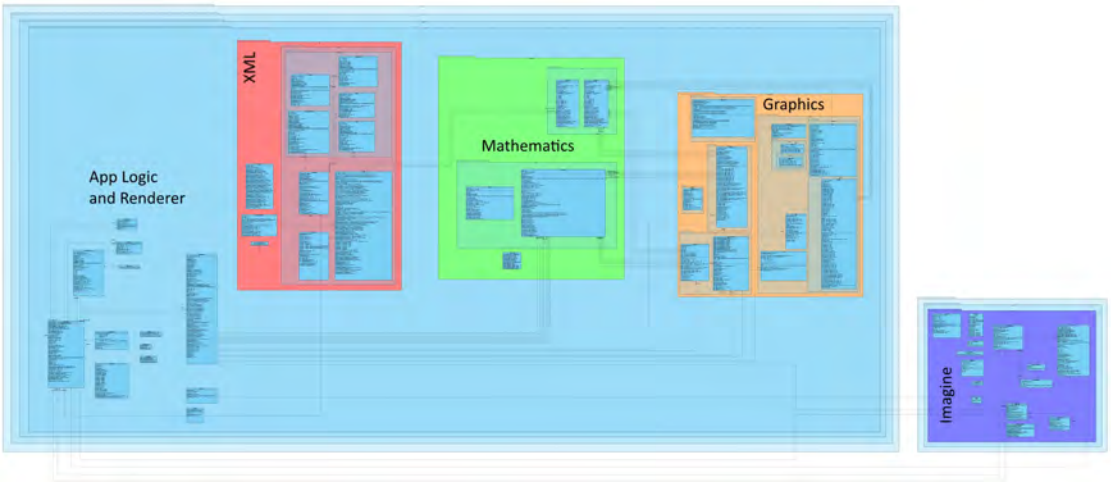

Abbildung 4.1.: Zwei Hauptkomponenten des Frameworks. Links, in blau, das Processor-Package, bestehend aus XML, Mathematics, Graphics, Renderer und Applikationslogik. Rechts, in lila, das Imagine-Package

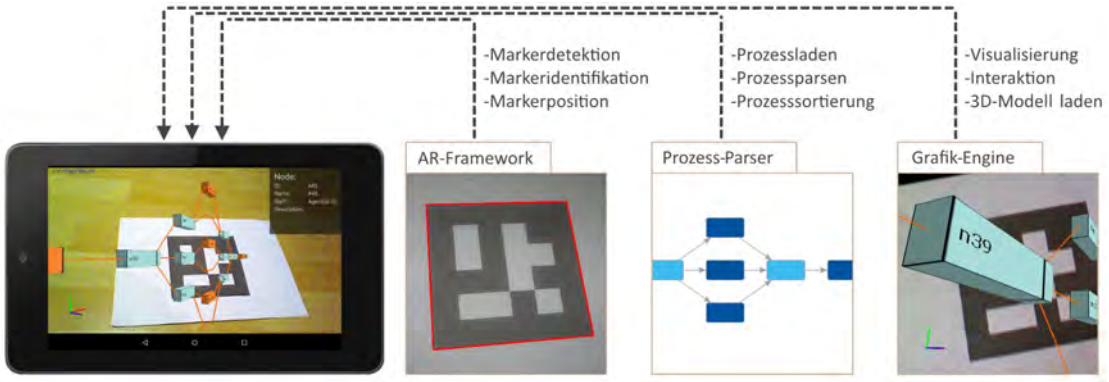

Abbildung 4.2.: Aufbau der Architektur des Processor-Frameworks

Der interessante Teil ist hier die Grafikkomponente samt Subkomponenten, welche den Großteil von Processor ausmachen. Diese Komponenten lassen sich in die grafische

Komponente (siehe Abbildung 4.3) und die Datenhaltung mit Datengenerierung der Prozessmodelle (siehe Abbildung 4.4) gliedern. Die Datenhaltung ist grundsätzlich sehr einfach gehalten. Alle Prozesskomponenten haben ihre eigenen Klassen, welche ihre zugehörigen Attribute beinhalten. Diese Komponenten werden beim Laden eines ADEPT-Prozessmodells mit Informationen befüllt und repräsentieren den Graphen des Prozesses. Die Repräsentation besteht hauptsächlich aus Kanten und Knoten. Nach Abschluss des Ladevorgangs, bzw. des Parsens der Prozessmodelle, liegen nun unsortierte Knoten und Kanten vor.
Beim nächsten Schritt werden die Knoten und Kanten sortiert. Dabei wird versucht Kantenüberschneidungen zu vermeiden. Nach erfolgreicher Sortierung werden für jeden Knoten und jede Kante eine dreidimensionale Repräsentation erstellt. Im Falle der Knoten wird ein 3D-Modell geladen. Kanten werden durch eine Linie dargestellt. Die Process-Klasse beinhaltet nach dem Ladevorgang ein Mapping zwischen Prozesskomponenten in reiner Datenform und Darstellungsform. Dieses Mapping ist wichtig, um später herauszufinden, welches 3D-Modell auf dem Bildschirm, welchem Datenobjekt im Prozess zugehört.
Die Entscheidung, die Daten so nahe am Renderer zu halten, lässt sich einfach begründet. Aus Performanzgründen gehören die Darstellungsdaten nahe zum Renderer. Da die Darstellung aus den Prozessdaten münden, liegt der Gedanke nahe, diese ebenfalls nahe an den Darstellungsdaten zu halten. Ebenso wäre die Trennung dieser spätestens beim Abfragen der Zugehörigkeit, also dem Mapping zwischen Daten und ihrer Darstellung, problematisch. Der Hauptthread für das Userinterface müsste mit dem Renderthread synchronisiert werden, weshalb sich die reine Prozessmodellinformation und die jeweilige Darstellung im Renderthread befinden.
Wird das Mapping nun aus dem Userinterfacethread abgefragt, stellt der Renderthread diese Information bereit, indem das gesuchte Objekt übermittelt wird. Dieses Objekt kann nun ohne Probleme dazu genutzt werden, um Information im nativen Userinterface von Android zu integrieren.

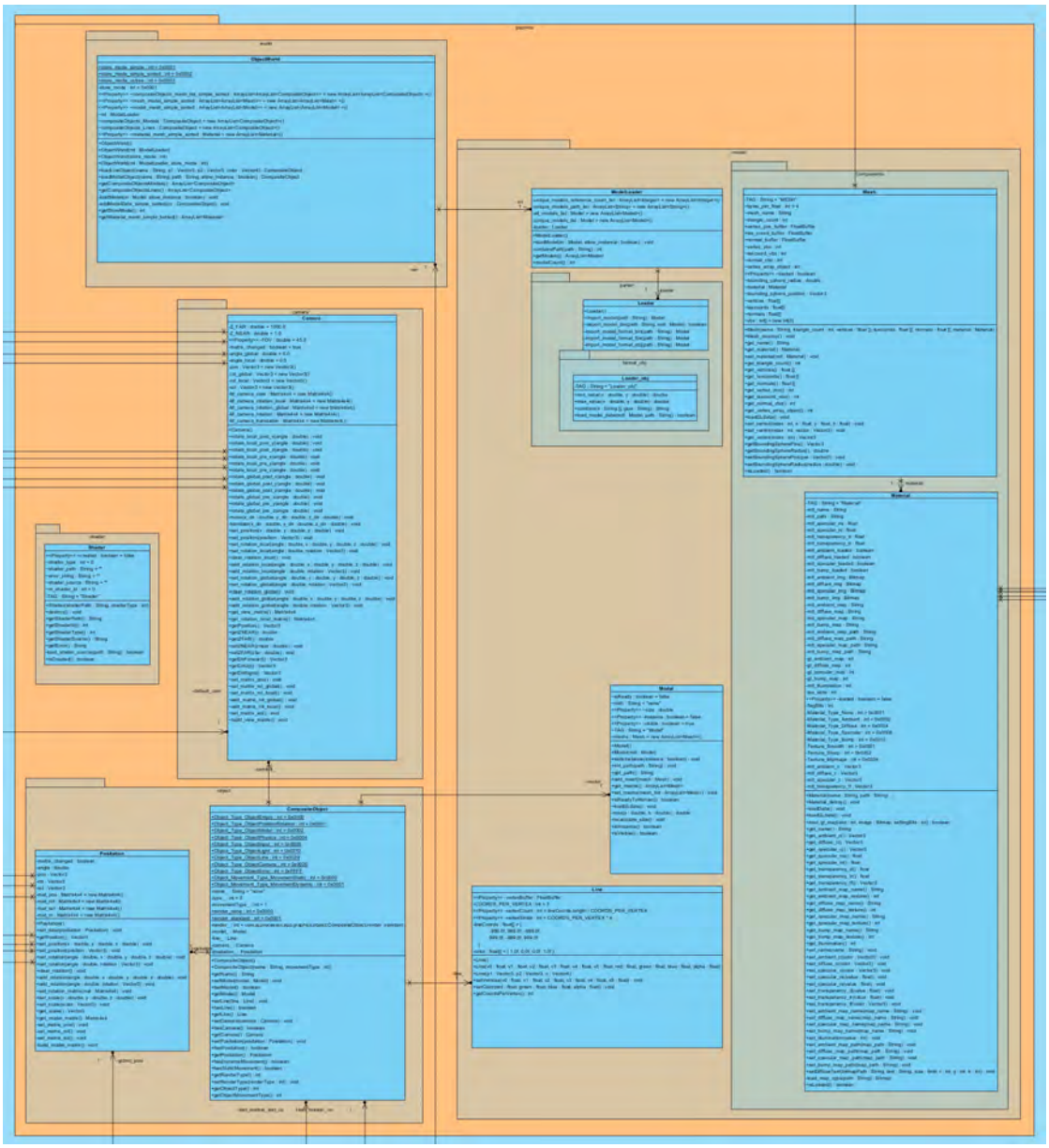

Abbildung 4.3.: Die grafische Komponente des Processor-Frameworks in UML

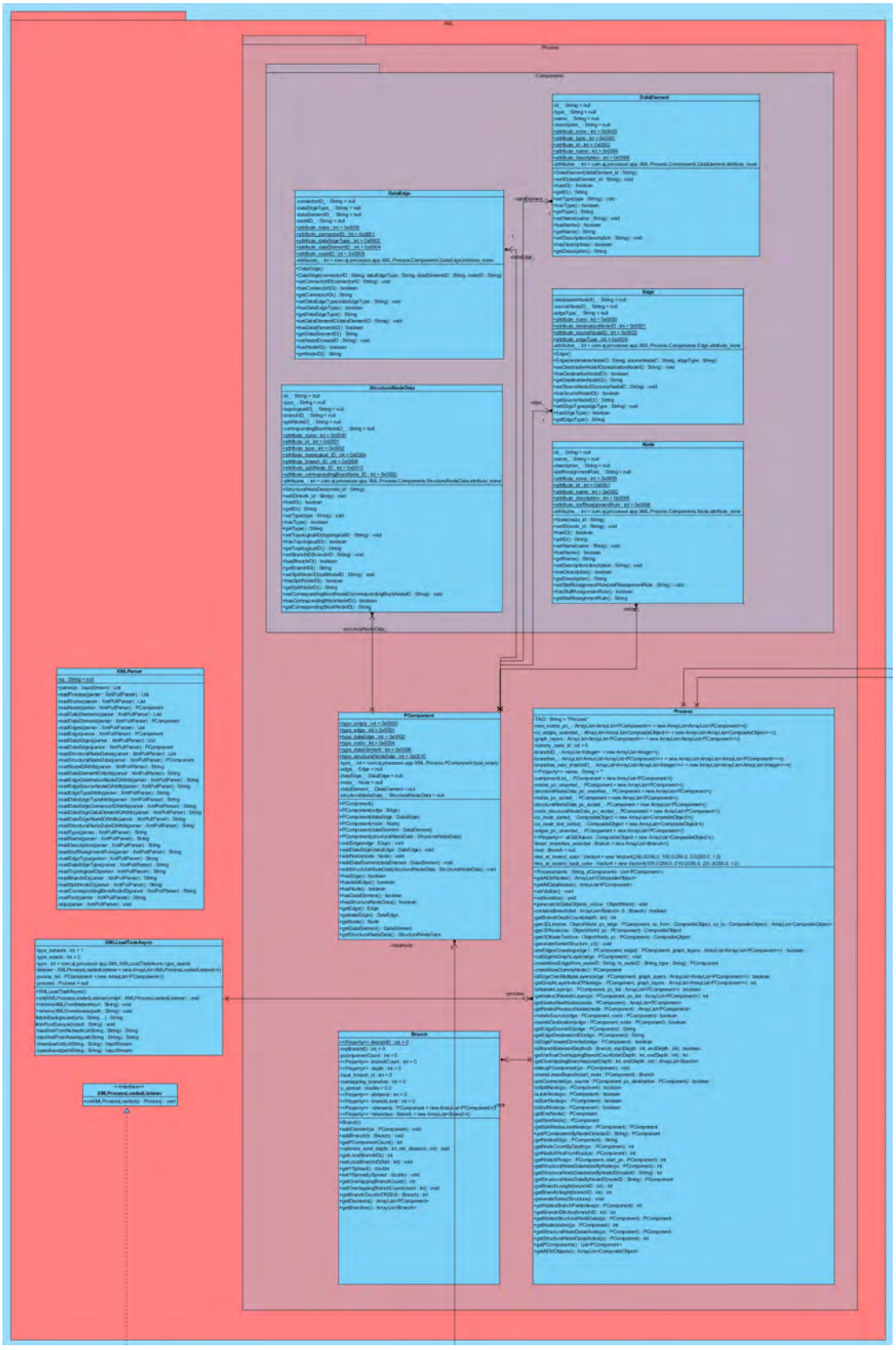

Abbildung 4.4.: Die XML Komponente des Processor-Frameworks in UML

4.2. Multithreading

Die einzelnen Komponenten arbeiten jeweils in eigenständigen Threads, um mehr Parallelität zu erzielen. Im Durchschnitt befinden sich zur Laufzeit zwei Threads in Arbeit. Einerseits der Haupthread für die Android Oberfläche und anderseits der OpenGL ES 2.0 Thread, welcher für das Darstellen des Prozessmodells zuständig ist. Ist die Markererkennung aktiviert, läuft ein zusätzlicher Thread, der nur für die Prozeduren der Binärisierung, Kantenerkennung und letztendlich Markererkennung benutzt wird. Außerdem wird beim Start des OpenGL-Threads, die Prozessinformation geladen. Dieser Vorgang wird ebenfalls für jedes Prozessmodell in einem eigenen Thread abgearbeitet. Der Prozessladevorgang könnte auch zu einem späteren Zeitpunkt erfolgen, mit dem Hintergedanken, dass Prozesse, z.B. erst dann geladen werden sollen, wenn sie sichtbar werden. Hier sind Threads sinnvoll, da somit das Hauptprogramm, also die übrigen Threads, nicht blockiert werden. Jedoch wäre das verzögerte Laden beim Testen hinderlich und wurde im finalen Framework vernachlässigt.

4.3. Darstellungsobjekte

Der Renderer ist sehr schlicht gehalten und bietet Funktionen zum Laden und Visualisieren von 3D-Modellen. Zum aktuellen Zeitpunkt gibt es zwei Arten von Modellen. Einerseits Linien und andererseits texturierte 3D-Geometrie, welche als Dateien vorliegen. Die Möglichkeit 3D-Objekte als externe Ressourcen zu laden, bietet zusätzlich Flexibilität, da diese Objekte in externen Tools erstellt werden können. Die Unterscheidung zwischen 3D-Objekt und Linie bleibt auch bei der Datenhaltung dieser im Renderer bestehen. Dies ist sinnvoll, um unnötige Funktionsaufrufe, sowohl auf der CPU, als auch der GPU, einzusparen. Der unnötige Aufwand hätte sich in zusätzlichen Shader-Wechsel geäußert.
3D-Objekte, bzw. Modelle, bestehen aus Texturen und Meshs. Eine Mesh entspricht einer dreidimensionalen Geometrie jeglicher Form. Zu dieser Geometrie gehören eine oder mehrere Texturen, so genannte Texture Maps. Die Modelle werden zusätzlich nach

ihren Subkomponenten sortiert, um Performanz zu steigern. Ein mehrdimensionales Mapping zwischen Textur, Mesh und Modell erreicht diesen Effekt. Der Vorteil dieser Sortierung wird ersichtlich, wenn man mehrere gleiche Modelle läd. Der Renderer versucht schon geladene Modelle zu instantiieren, was sich als Referenz auf diese äußert. Sind nun relativ viele, gleiche Modellobjekte in einem Prozessmodell zu visualisieren, muss man nur die Darstellungsdaten für das erste Objekt auf der GPU laden. Alle folgenden Objekte referenzieren auf die selbigen Daten und nutzen die schon geladene Repräsentation. Somit entfallen Textur - und Geometriewechsel auf der GPU und die Aufrufe zu diesen auf der CPU. Eine detailliertere Erklärung zur Optimierung der Renderprozedur folgt in Kapitel 5.

5

Implementierung & Implementierungsaspekte

Im Fokus steht die Implementierung eines funktionsfähigen Prototypen, weshalb es eine Fülle an Komponenten gibt, die näher betrachtet werden könnten. Alle Komponenten genau zu erläutern, würde den Rahmen dieser Arbeit sprengen und somit werden nur einige wichtige Bestandteile genauer betrachtet.
Bevor ein Benutzer überhaupt mit der grafischen Repräsentation eines Prozessmodells interagieren kann, müssen zunächst andere Konfigurationsschritte erledigt werden, weswegen es wichtig ist das folgende Berechnungsmodell (vgl. Abbildung 5.1) näher zu erläutern.
Der Renderer benötigt grafische Repräsentationen der einzelnen Bestandteile eines Prozesses. Diese können mit diversen 3D-Modellierungs-Tools erstellt werden, wobei beispielhaft 3ds Max 2015 für die 3D-Modelle in dieser Bachelorarbeit verwendet wurde.

Die Modelle müssen in das Wavefront OBJ Format exportiert werden. Dieses speichert die Geometrien und Materialien in einem einfach lesbarem Textformat neben den Texturen ab. Dieser Schritt ist wichtig, da der Renderer zu diesem Zeitpunkt nur das Wavefront OBJ Format kennt.
Nun da der Renderer alle visuellen Daten hat, fehlt nur noch die Information bezüglich des Prozesses, welcher gezeichnet werden soll. Solch ein Prozess wird z.B. aus Aristaflow BPM Suite exportiert [DKR+95, DR09, DRRM+09]. Als XML vorliegend, kann ein Prozess geladen, geparsed, sortiert und visualisiert werden.
Sind soweit beide Input-Seiten des Frameworks mit Exporten abgedeckt, werden Prozeduren für den Import der Daten benötigt. Das Hauptaugenmerkmal liegt hier auf Performanz hinsichtlich optimierter Datenstrukturen, welche schnelle Zugriffszeiten und schnelles Darstellen ermöglichen. Einige Optimierungen dieser Art werden in den folgenden Sektionen näher betrachtet.

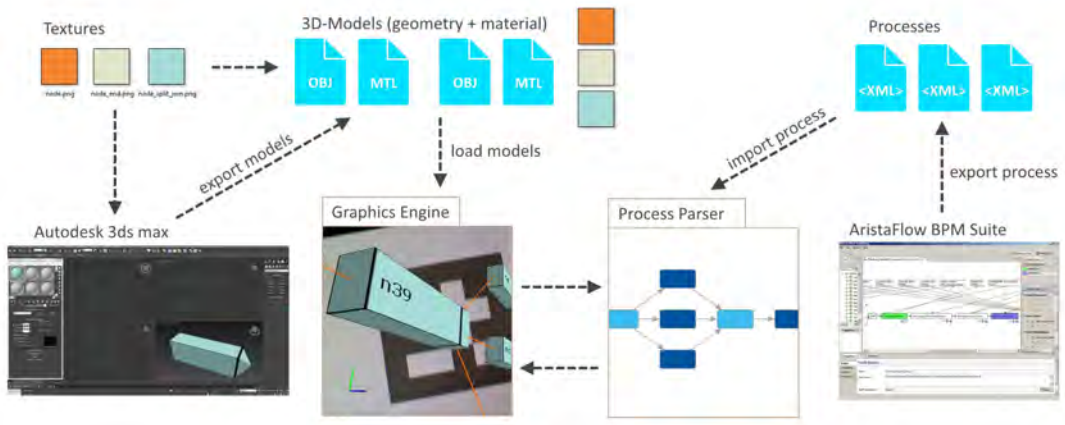

Abbildung 5.1.: Berechnungsmodell

5.1. Vom naiven zum optimierten Renderer

Der wichtigste Aspekt des Frameworks liegt auf dem performantem Zeichnen von Prozessmodellen, oder anders betrachtet, einer Sammlung von Darstellungsobjekten.
Im Grunde können zwei Faktoren die Geschwindigkeit des Zeichnens beeinträchtigen.

Einerseits bremsen zu viele CPU-Operationen die Performanz und andererseits zu viele Operationen auf der GPU. Um dieses Problem zu umgehen, muss die Datenhaltung optimiert werden, damit der Zugriff auf Daten beschleunigt werden kann. Zusätzlich lässt sich die Anzahl an Operationen auf der CPU, welche nötig sind, um die GPU-Operationen durch zuführen, bzw. einzuleiten, reduzieren. Die Reduktion dieser CPU-Befehle hat folglich auch positive Auswirkungen auf die Anzahl der Befehle, die auf der GPU ausgeführt werden müssen. Um die GPU und CPU unter Verwendung von OpenGL ES 2.0 zu entlasten, muss zunächst verstanden werden, wie Objekte damit gezeichnet werden.

5.1.1. Die 3D Objekte

Hierbei ist der Aufbau eines 3D-Objektes interessant. 3D-Objekte sind in zwei Kategorien unterteilt: (1) Linien und (2) 3D-Modelle. Der Aufbau von Linien ist simpel gehalten. Zwei Punkte stellen die geometrischen Daten dar und eine Farbe vervollständigt diese. 3D-Modelle sind komplexer und setzen sich aus Geometrien und Texturen zusammen. Um das Ganze ein wenig zu vereinfachen, besitzt jede zusammenhängende Geometrie, auch Mesh genannt, ein Material. Materialien definieren zusätzliche Parameter, welche z.B. die Beleuchtung beeinflussen. Ein Material besteht im Framework aus bis zu vier Texture Maps, jedoch wird nur die diffuse Texture Map zum Zeichnen verwendet, da der Renderer auf komplexe Beleuchtung und Schattierung verzichtet.
Aus diesen Einzelteilen lassen sich einfache geometrische Gebilde erstellen. Im Framework können sich 3D-Modelle aber auch aus mehreren Meshs zusammensetzen. Abbildung 5.2 zeigt, wie sich ein 3D-Modell aus seinen einzelnen Komponenten zusammensetzt.

Aus Speicher und Performanzgründen wird schon an den 3D-Modellen versucht zu optimieren. Besitzen Modelle die gleiche Textur, muss diese nicht neu geladen werden. Folglich ändert sich die vorherige Abbildung ein wenig und es werden Referenzen auf schon vorhandene Texturen und sogar komplette Modelle erstellt, falls die Möglichkeit dazu gegeben ist. In Abbildung 5.3 teilen sich zwei Modelle Texturen. Ein Modell ist eine Instanz eines schon geladenen Modells und besitzt somit nur eine Referenz auf die Geometrien.

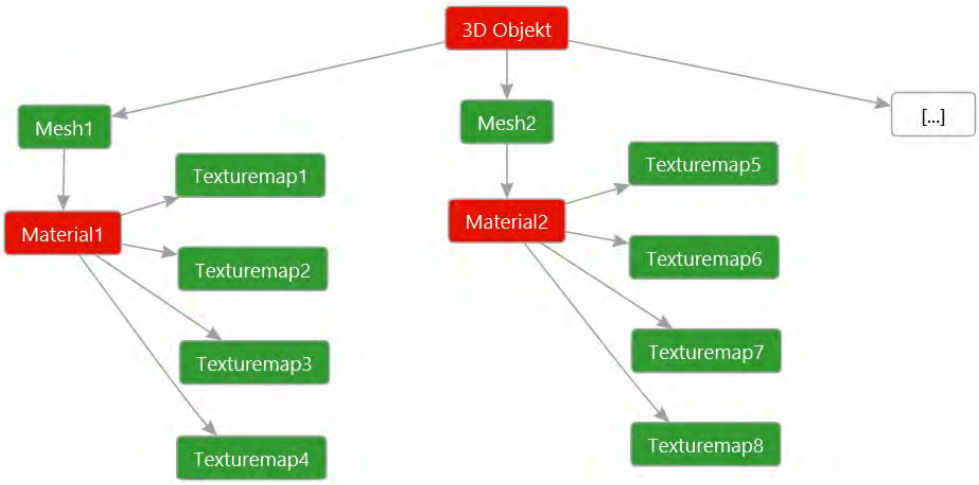

Abbildung 5.2.: Das 3D-Modell und seine Komponenten; Rote Komponenten befinden sich komplett oder zum größten Teil im RAM und werden von der CPU verarbeitet; Grüne Komponenten befinden sich hingegen fast ausschließlich auf der GPU, bzw. dem VRAM.

5.1.2. Entlastung der GPU und CPU

Bevor Objekte gezeichnet werden, müssen die, zum Objekt gehörenden, Daten gebunden werden. Seien diese Daten schon auf der GPU geladen, und liegen entsprechend konfiguriert vor, damit sie einsatzbereit sind, dann muss auch das entsprechende Zeichenprogramm, das so genannte Shaderprogramm, gebunden werden. Erst jetzt ist OpenGL bereit, um die gebundenen Daten zu verarbeiten, und Pixel im Framebuffer

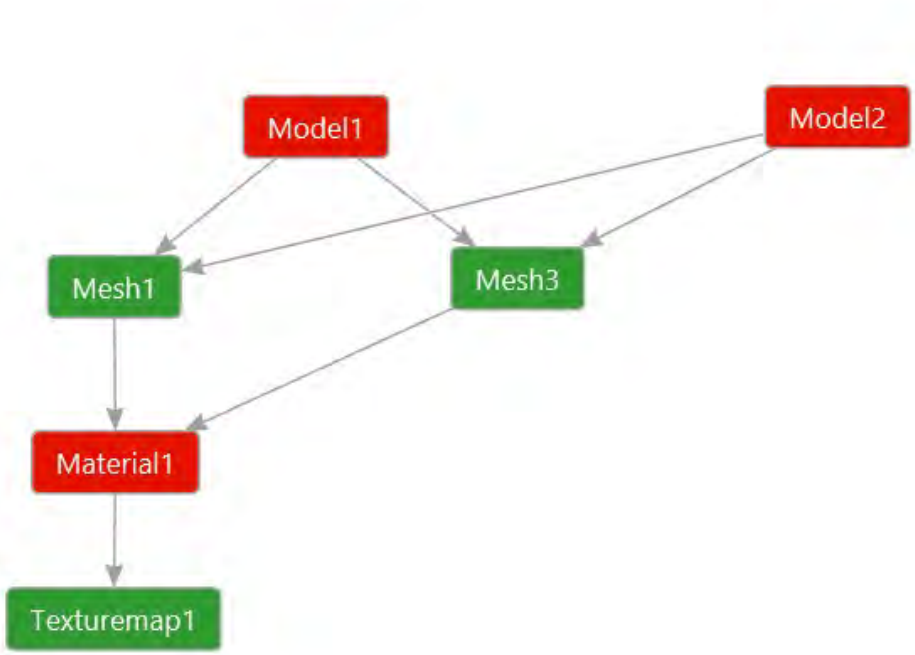

Abbildung 5.3.: 3D-Modelle mit beispielhaften Referenzen auf Komponenten für Speicheroptimierung bei Verwendung von gleichen Daten

abzulegen. Die naive Zeichenprozedur, siehe Abbildung 5.4, würde für jedes, zu zeichnende, Objekt die zugehörigen Daten und das entsprechende Shaderprogramm binden.

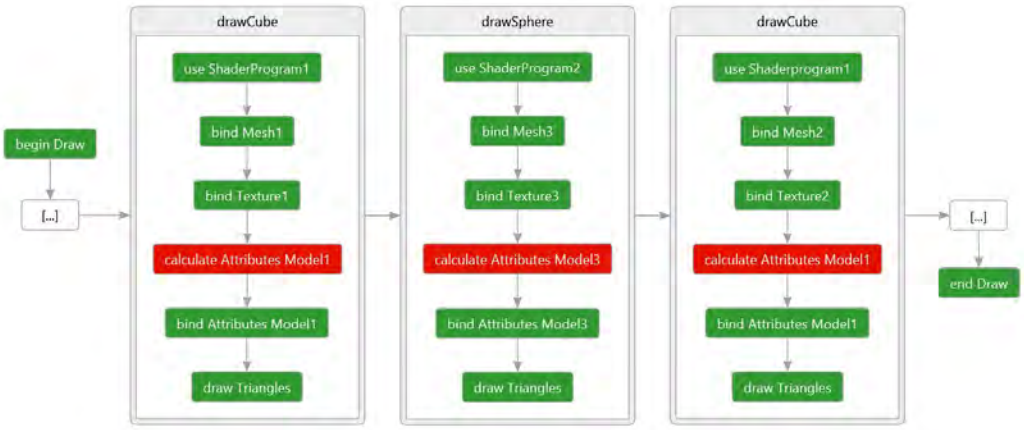

Abbildung 5.4.: Der naive Renderer; Grüne Funktionsaufrufe sind GPU intensiv, benötigen aber auch CPU Zeit; Rote Funktionsaufrufe sind CPU intensiv.

Dies ist sehr ineffizient, da nicht jedes Objekt unterschiedlich hinsichtlich seiner visuellen Daten ist. Solange sich nur Attribute, wie z.B. Position, Rotation und Skalierung, ändern, können Objekte, welche die selben Daten und das selbe Shaderprogramm nutzen, gebunden bleiben, und müssen nicht neu gebunden werden. Berücksichtigt man diese Tatsache, können somit viele Objekte mit dem selbigen Datensatz, aber unterschiedlicher Attribute, kompakt und effizient gespeichert und folglich gezeichnet werden. Beispielsweise können nun hunderte würfelartige Geometrien in den Framebuffer geschrieben werden, jedoch benötigt dies nur noch einen geometrischen Datensatz eines Würfels auf der GPU. Attribute müssen weiterhin für jedes Objekt gespeichert werden. Diese sind in der Regel jedoch kompakter und fallen somit nicht stark ins Gewicht. Nimmt man jetzt noch eine effiziente Berechnung dieser Attribute hinzu, wird die Anzahl der CPU-Befehle noch weiter reduziert.
Um solche Neubindungen von GPU-Daten zu verhindern, werden zunächst die Objekte nach ihren Shaderprogrammen gruppiert. Dadurch werden Linien mit ihrem eigenem Programm gezeichnet und Geometrie mit Texturen besitzen ebenso ihr eigenes Pro-

gramm.
Als Resultat dieser Gruppierung muss pro Gruppe nur noch einmal das Shaderprogramm gebunden werden. Abbildung 5.5 zeigt den groben Aufbau der verbesserten Prozedur. Ein positiver Nebeneffekt ist die Reduzierung von CPU-Befehlen, welche für die Bindung der Shaderprogramme benötigt wäre.

Geht man jetzt einen Schritt weiter und sortiert die Geometrien mit Textur nach ihrer Textur, also gruppiert diese unter Berücksichtigung ihres Objektes, der Textur und der Geometrie, können wieder Befehle für die Bindung dieser Daten eingespart werden.
Im Prototypen wird nach Texturen gruppiert. Dadurch lassen sich Texturbindungen einsparen, sobald mehrere Geometrien die selbe Textur nutzen. Beim Laden der Modelle wird diese Optimierung angestoßen und Algorithmus A.2 sortiert die einzelnen Modell-Komponenten in die jeweiligen Arrays. Ebenso wäre es möglich auch nach Geometrie zu gruppieren. Dies wurde jedoch vernachlässigt, da Texturen, betrachtet man den benötigten Speicher, größer als die später verwendeten Geometrien sind und deshalb ein Texturwechsel aufwendiger als ein Geometriewechsel ist. Eine gängige Methode wäre es auch Texturen unterschiedlicher Geometrien zu einer Einzigen zusammenzufassen und nur diese zu nutzen, was jedoch den Aufwand Modelle zu erstellen anhebt. In Abbildung 5.6 sieht man eine Vereinfachung der, im Renderer verwendeten, Prozedur, welche durch die sortierten und somit gruppierten Modell-Komponenten iteriert.

Zuletzt wird noch eine Optimierung auf der CPU Seite durchgeführt. Die Berechnung einiger Attribute einzelner Modelle kann, falls dies für jedes Modell oder sogar jede Geometrie durchgeführt wird, enorm viel Zeit beanspruchen. Die Frage, die sich jetzt stellt ist: Was sind diese Attribute und wieso sind diese so CPU intensiv? Hier handelt es sich um 4x4-Matrizen. Diese bestimmen die Position, Rotation und Skalierung der Modelle und müssen mit einander multipliziert werden. Zusätzlich dazu, kommen die Kameratransformationen für Position, Rotation und Perspektive.
Alle genannten Transformationen lassen sich relativ einfach optimieren, wenn ein kleiner Paradigmenwechsel vollzogen wird. Normalerweise würde man jedem Objekt eine absolute Transformation im 3D-Raum zuweisen. Dies bedeutet aber, dass jede Kameratransformation für jedes Modell eine Neuberechnung seiner Transformationen nach sich

Abbildung 5.5.: Erste Verbesserung des Renderers; Blaue Funktionsaufrufe sind reduziert worden und werden somit nur ein mal pro Shaderprogramm benötigt.

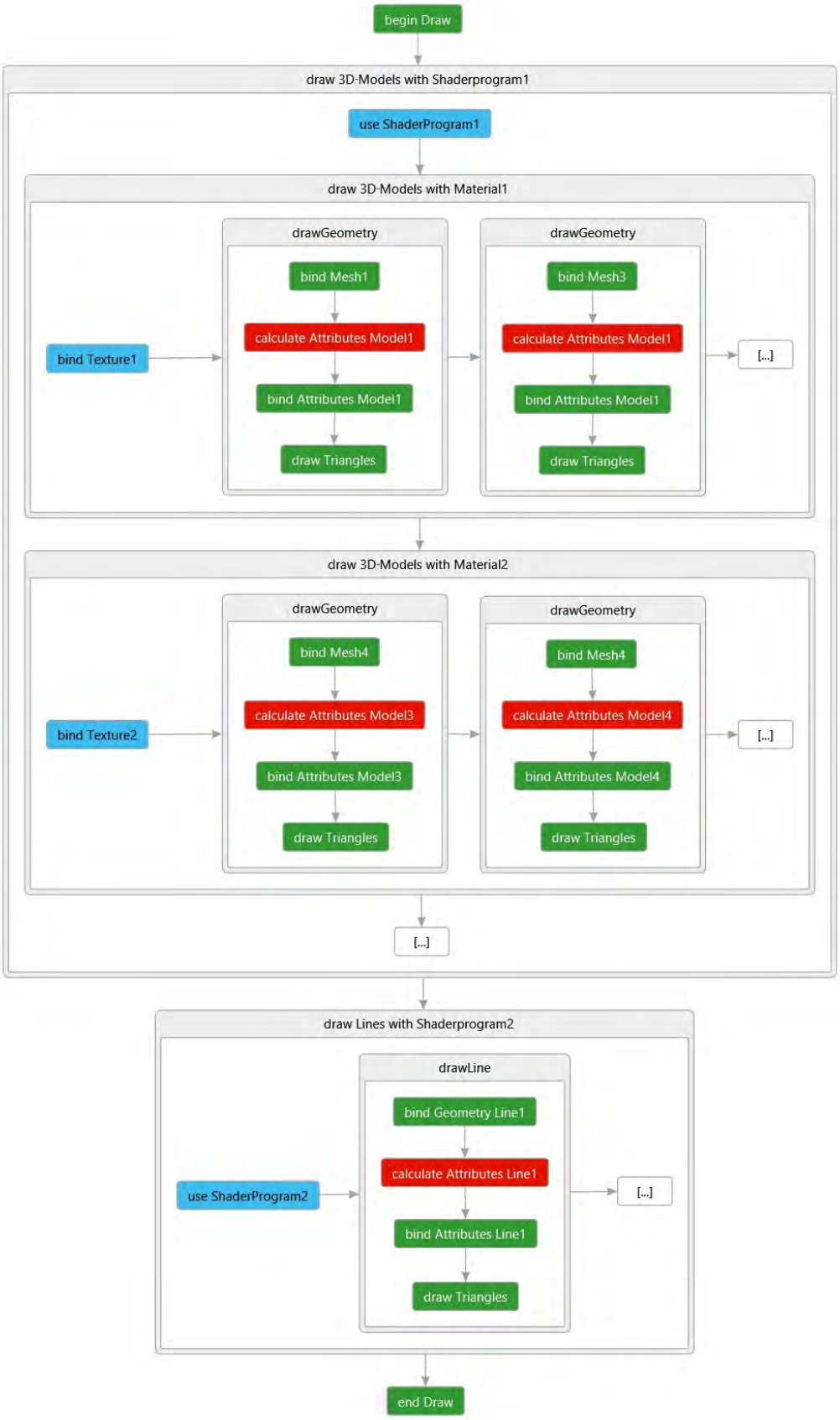

Abbildung 5.6.: Der finale Renderer; Zusätzliche Verbesserungen in Form von reduzierten Texturbindungen sind hinzugekommen; Diese sind ebenfalls blau hinterlegt.

zieht. Durch eine andere Anschauung der Kameratransformation lassen sich aber auch globale Transformationen durchführen, welche alle Modelle, bzw., alle Modelltransformationen mit transformieren. Somit sind die Modelltransformationen relativ und müssen nur noch ein einziges Mal berechnet werden, z.B. beim Anordnen der einzelnen Objekte im Raum. Abbildung 5.7 zeigt, wie sich diese optimierten Transformationen in den finalen Renderer integrieren.

All diese Optimierungen erlauben es, relativ kosteneffizient, eine sehr große Anzahl an gleichen Objekten zu zeichnen. Besitzt, z.B. der geplante, visuelle Output nur drei unterschiedliche Geometrien mit drei unterschiedlichen Texturen, können hunderte dieser Objekte im finalen Bild erscheinen. Dabei wird aber nur einmal das Shaderprogramm, dreimal die Geometrie und dreimal die Textur gewechselt. Genau so sieht auch der Speicher aus. Auf der GPU liegen nur drei Texturen und drei Geometrien. Dies stellt eine enorme Performanzsteigerung, im Vergleich zur naiven Renderprozedur, dar. Der Render-Algorithmus A.3 zeigt den einfachen Renderer, im Vergleich zum sortierten und optimierten Renderer.

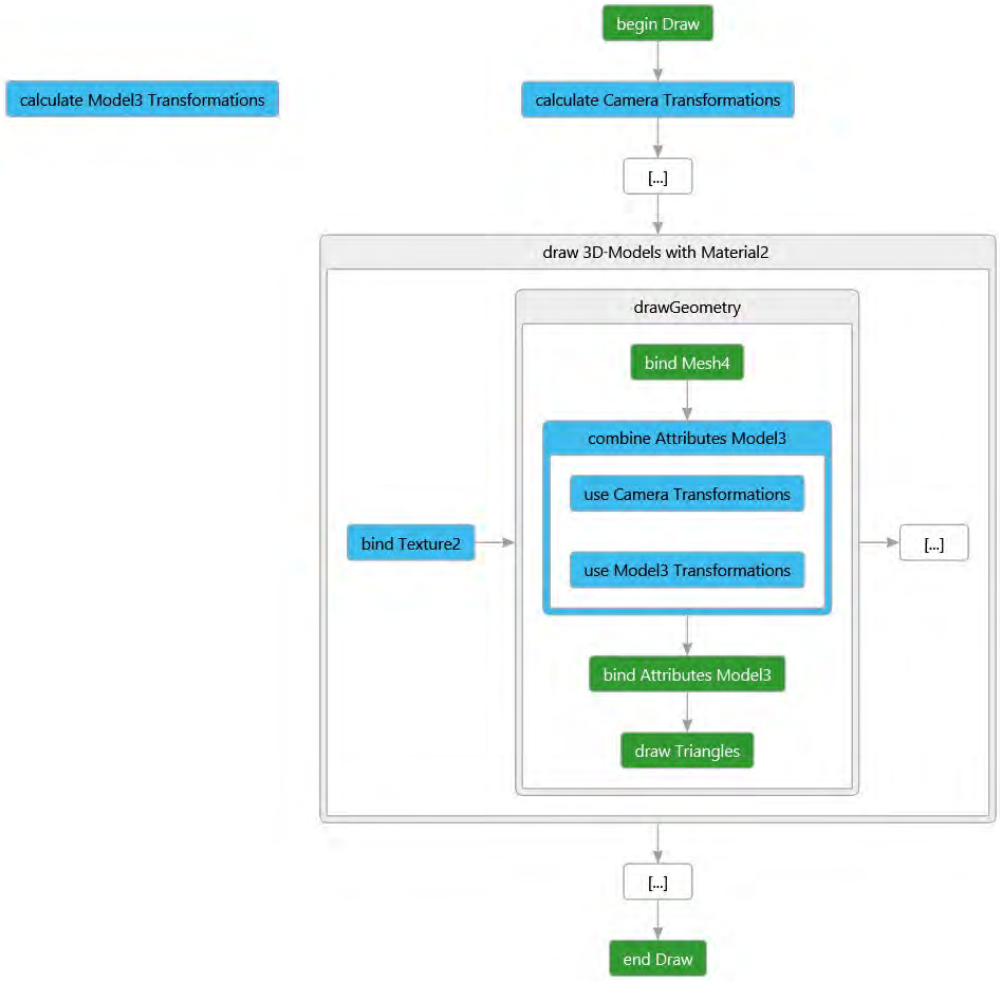

Abbildung 5.7.: Optimierte Transformationen; Berechnungen der Modelltransformationen befinden sich nicht mehr in der Renderschleife; Kameratransformationen werden nur noch einmal pro Frame berechnet; Pro Geometrie wird nur noch Kamera und Modelltransformation zusammengesetzt.

5.2. Das Prozessmodell

Das Prozessmodell besteht aus Knoten und Kanten. Knoten haben unterschiedliche Darstellungsformen, da es Joins, Splits, Tasks, Start und Endknoten gibt. Abbildung 5.8 listet alle verwendeten Knotentypen auf.

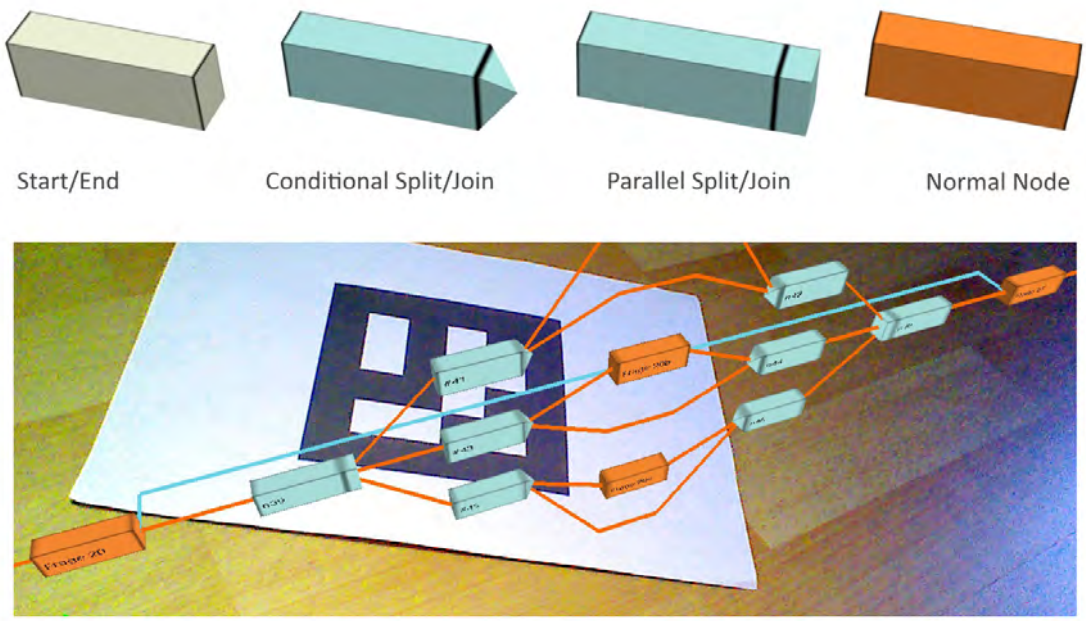

Abbildung 5.8.: Von links nach rechts sind die Knotentypen aufgelistet: Start/End Node, Conditional Split/Join, Parallel Split/Join, Normal Node

Kanten sind im Prototypen nur in zwei Kategorien unterteilt: Vorwärts gerichtete Kanten und Rücksprungkanten. Vorwärts gerichtete Kanten befinden sich in der finalen Darstellung des Graphen auf der Ebene der Knoten. Die Rücksprungkanten sind jedoch in der dritten Dimension nach hinten verschoben, um die Übersicht zu behalten. Beide Kantentypen unterscheiden sich visuell noch zusätzlich durch unterschiedliche Farben.

5.2.1. ADEPT XML Parser

Der Parser folgt zum größten Teil den Empfehlungen von Android Developers [And13]. Dieser liest die XML-Datei in einem eigenen Thread ein, und erstellt für die einzelnen Graphkomponenten Objekte des jeweiligen Types und füllt dessen Attribute. Ist die Datei komplett ausgelesen, werden die erstellten Objekte in den Prozesscontainer gespeichert und dieser sortiert den Graphen, wie es im folgenden Abschnitt beschrieben wird. Ist die Sortierung abgeschlossen, terminiert der Thread und der Graph kann in die Darstellungsform überführt werden.

5.2.2. Layered Graph

ADEPT-Prozesse [DKR+95, DR09, DRRM+09] sind eigentlich azyklisch, besitzen jedoch Rücksprungkanten. Um die finale Darstellung des Graphen möglichst überkreuzungsfrei zu visualisieren, bedarf es hier der Hilfe einer Strategie, welche einfach und zuverlässig dieses bewerkstelligt. Layered Graphs [ESK05] erweisen sich als relativ nützlich, und sind simpel in ihrer Funktionsweise. Abbildung 5.9 zeigt einen Graphen, der in dem folgenden Abschnitten sortiert wird, da seine Komponenten nicht sortiert geladen werden.

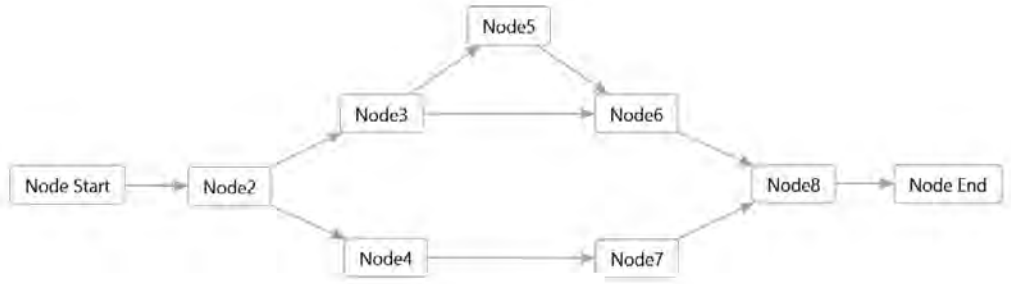

Abbildung 5.9.: Der, zu sortierende, Beispielgraph

Das Funktionsprinzip ist wie folgt. Der Startknoten ist der Anfang des Graphen und stellt die erste Ebene, bzw. Layer, dar. Der anknüpfende Knoten bildet die zweite Ebene. Alle

weiteren Knoten folgen diesem Prinzip, dass Nachfolger in die nächste Ebene abgelegt werden. Vertikale Verbindungen werden ausgeschlossen, da Folgeknoten immer im nächsten Layer landen müssen. Rücksprungkanten werden in diesem Prozess vorerst nicht berücksichtigt. Abbildung 5.10 zeigt den Graphen, nachdem die Knoten in die einzelnen Ebenen gespeichert wurden.

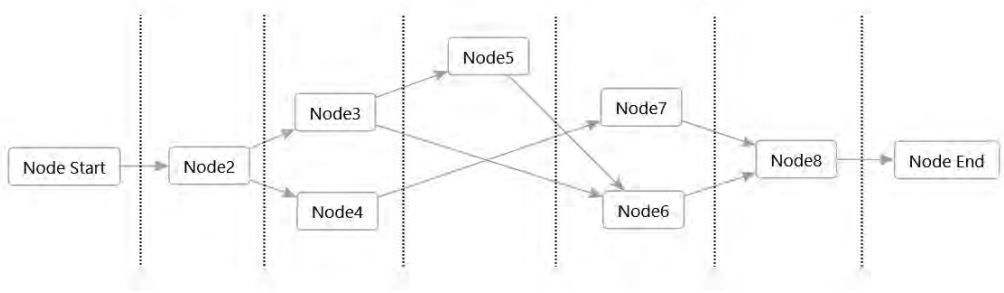

Abbildung 5.10.: Die Knoten befinden sich nun in den einzelnen Ebenen, können aber Überschneidungen in den Kanten aufweisen.

Als zweites wird überprüft, ob Kanten sich über mehrere Ebenen erstrecken. Ist dies der Fall, wird die Kante unterteilt und ein leerer Dummy-Knoten wird in die Ebene dazwischen eingefügt. Auch hier werden Rücksprungkanten nicht berücksichtigt. Der Graph in Abbildung 5.11 enthält nun diese zusätzlichen Knoten.

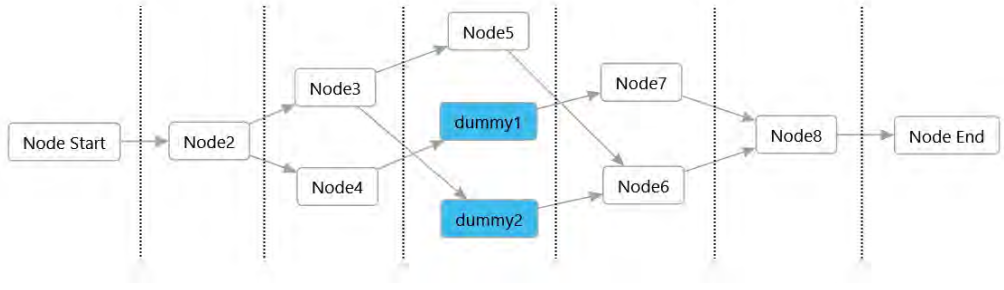

Abbildung 5.11.: Zusätzliche Knoten verhindern, dass Kanten sich über mehrere Ebenen erstrecken.

Jetzt werden die Knoten, samt zusätzlichen Dummy-Knoten, in ihren Ebenen sortiert, um möglichst Kantenüberschneidungen zu den vorherigen Ebenen zu vermeiden. Die Sortierung verläuft von der ersten bis zur letzten Ebene und geschieht im finalen Framework unter einer oberen Grenze für die Sortierungen pro Ebene. Diese Grenze dient dazu, die Berechnungsdauer in eine zeitliche Schranke zu packen und eventuelle Endlosschleifen zu vermeiden. Abbildung 5.12 ist nun pro Ebene sortiert, um Überkreuzungen zu den vorherigen Ebenen zu vermeiden.

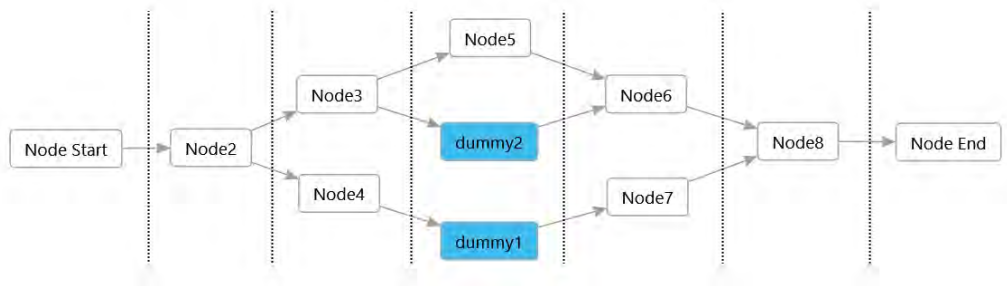

Abbildung 5.12.: Die einzelnen Ebenen sind sortiert, um überkreuzte Kanten zu vermeiden.

Sind die Ebenen sortiert, werden beim Darstellen des Graphen, die zusätzlich hinzugefügten Knoten nicht gezeichnet und die zwei, vorher verbundenen, Knoten werden durch eine neue Kante, durch die Dummy-Knoten verlaufend, verbunden. Die Rücksprungkanten können ohne besondere Veränderungen wieder hinzugenommen werden. Diese sind später, bei der Darstellung, sowieso in die dritte Dimension verschoben und können maximal untereinander kreuzen. Der Graph ist nun bereit zum Darstellen und kann, so, wie er in Abbildung 5.13 abgebildet ist, gezeichnet werden. Im Algorithmus A.1 findet sich die Sortierung des Graphen, nach dem zuvor beschriebenen Verfahren, wieder.

5.2.3. Mapping zwischen Darstellung und Modell

Um herauszufinden, welcher Knoten sich an welcher Position auf dem Bildschirm befindet, benötigt es ein Mapping zwischen Darstellung und Modell. Dieses Mapping wurde

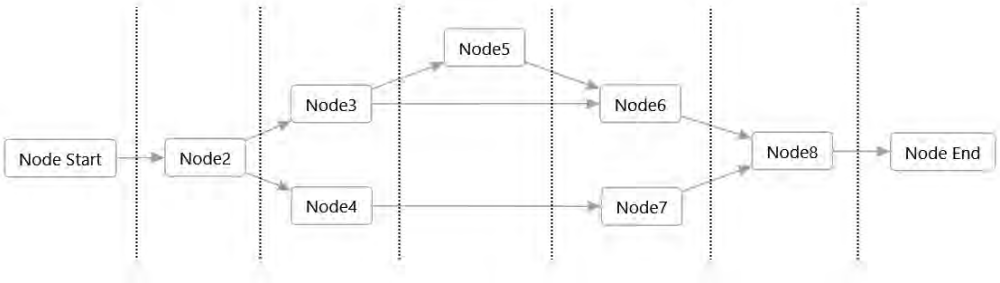

Abbildung 5.13.: Die Überkreuzungen sind behoben, und der Graph ist bereit zum Zeichnen.

durch jeweils zwei Listen realisiert, wobei die Listen so sortiert sind, dass der Index beider Listen dazu verwendet wird, um vom Modell zum Darstellungsobjekt oder umgekehrt zu gelangen. Nachdem der Bildschirm berührt wurde, findet eine Suche nach Knoten statt, welche dem, als Strahl projizierten, Berührpunkt am nächsten ist. Wurde ein Knoten gefunden, wird der Index gesucht und mittels diesem Index die Modellinformation abgerufen und angezeigt.

6

Vorstellung der Anwendung

6.1. Konfiguration

Beim Start der Applikation landet man, wie auf Abbildung 6.1 zu sehen, bei den Einstellungsmöglichkeiten, die jeweils zur Markererkennung gehören und Marker mit Prozessmodellen verbinden können.
Die Einstellungsmöglichkeiten für die Markererkennung wurden aus dem Imagine-Framework übernommen, aber noch zusätzlich übersichtlicher gruppiert.
Die Verbindung von Marker und Prozessmodellen besteht aus einem einfachen Mapping zwischen der ID des Markers und dem Pfad des Prozessmodells.

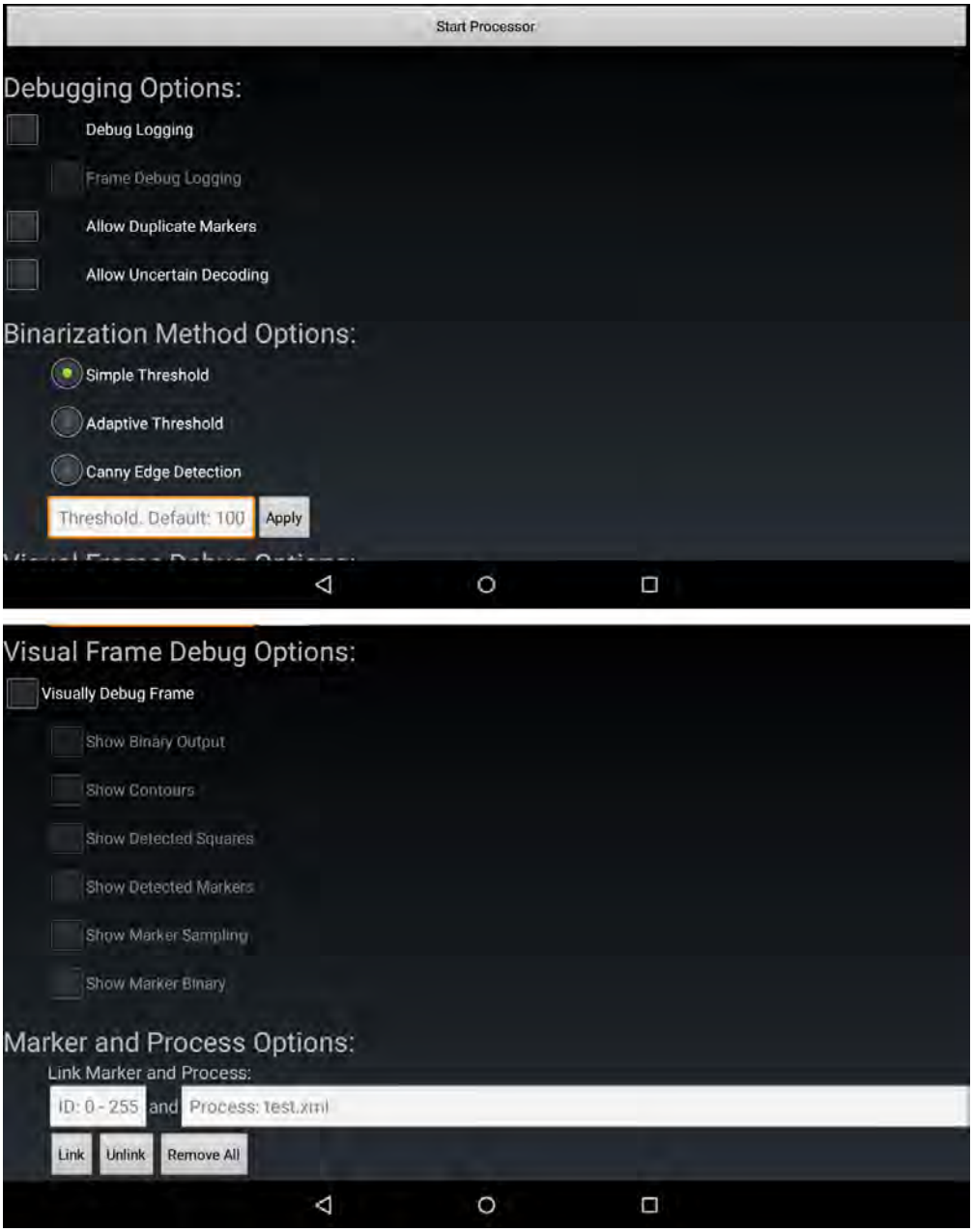

Abbildung 6.1.: Einstellungsmöglichkeiten bevor man die Markererkennung startet

6.2. Komplexe Geometrie

In Abbildung 6.2 wird ein, aus den Ressourcen geladenes, 3D-Objekt an der Stelle, an dem der zugehörige Marker erkannt wurde, gezeichnet. Das Objekt besteht aus einer Geometrie und einem Material, welches wiederum aus einer Textur aufgebaut ist. Dies ist der Vorreiter des finalen Prototypen, da Prozessmodelle sich aus genau solchen Einzelteilen zusammensetzen.
Zusätzlich sind einige Debug-Optionen in der linken oberen Ecke aktiviert.

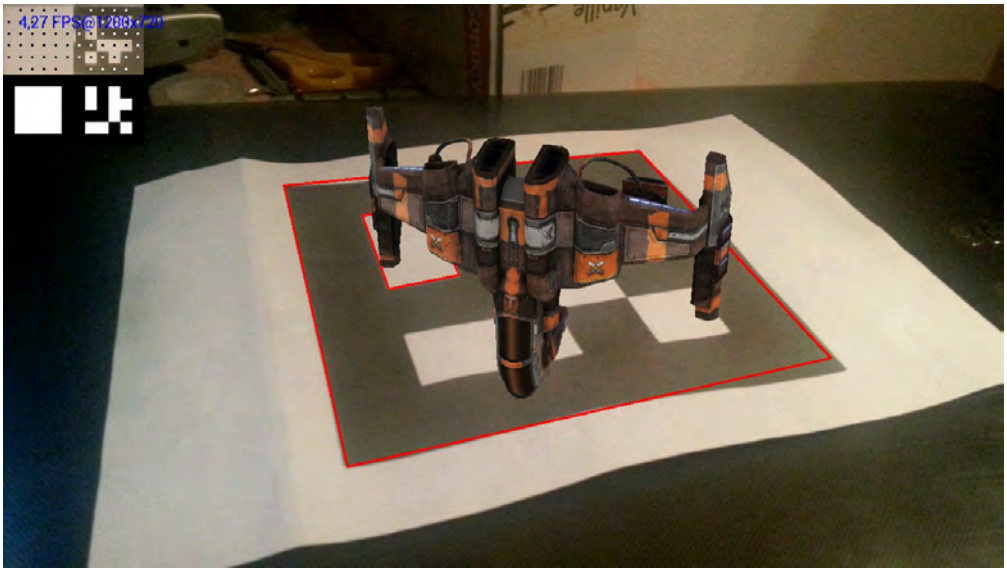

Abbildung 6.2.: Geladenes 3D-Objekt mit Textur über erkanntem Marker

6.3. Visualisierung eines Prozessmodells

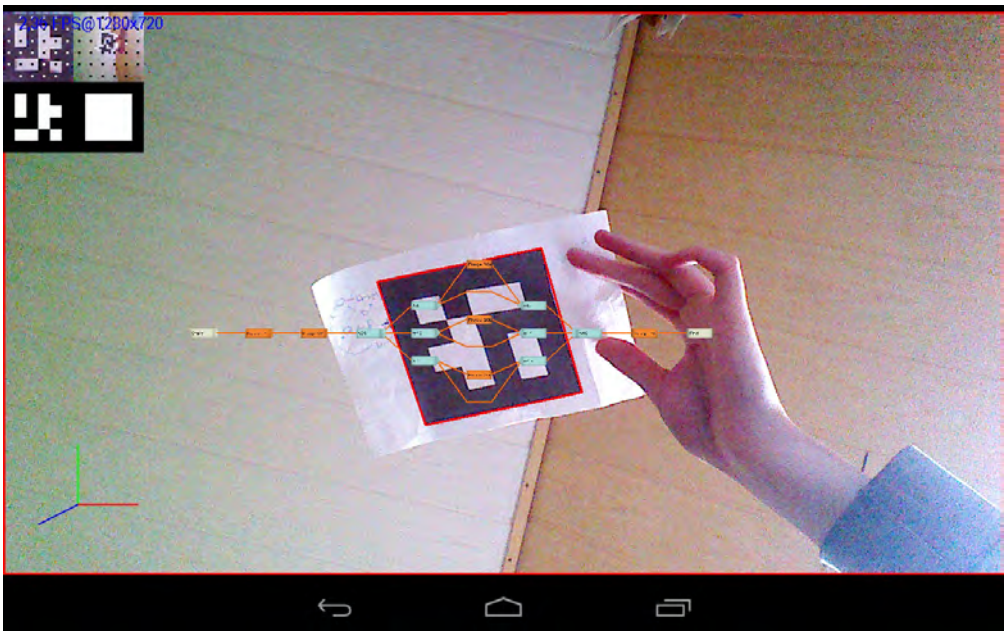

Abbildung 6.3.: Prozessmodell auf dem dazugehörigen Marker

Als nächster logischer Schritt folgt in Abbildung 6.3 ein komplettes Prozessmodell. Dieses wird zunächst sortiert und aus einzelnen 3D-Objekten zusammengesetzt.

6.4. Interaktionsmöglichkeiten

Mit dem sichtbaren Prozess kann, ähnlich wie bei 3D-Modellierungstools, interagiert werden. Die Interaktion besteht aus Rotation, Skalierung und Verschiebung des ganzen Modells (siehe Abbildung 6.4). Um zwischen den Interaktionen zu wechseln, muss nur ein Druck auf das Gizmo in der unteren linken Ecke erfolgen. Das Gizmo zeigt auch zusätzlich an, in welchem Interaktionsmodus man sich befindet.

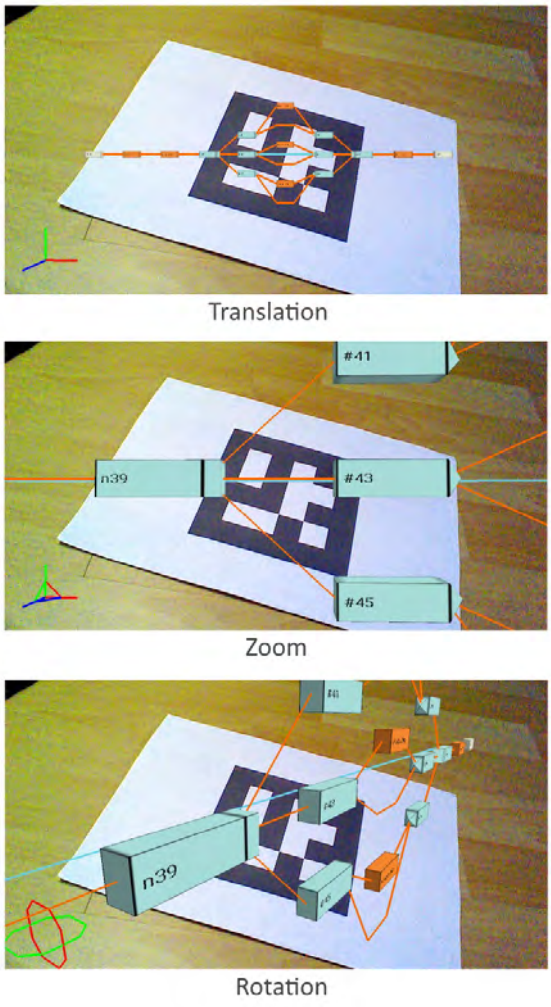

Abbildung 6.4.: Interaktionsmöglichkeiten mit einem Prozessmodell

6.5. Zusätzliche Information zu einzelnen Tasks

Zusätzlich kann weitere Information zu den einzelnen Tasks abgerufen werden. Dazu muss nur einer dieser gedrückt werden, und eine einfache Strahl-Kugel-Kollisionserkennung wird durchgeführt, um den richtigen Task unter dem Druckpunkt zu finden.
Die zusätzliche Information wird in Abbildung 6.5 illustriert und besteht aus weiteren Attributen des Tasks.
Durch Berühren der linken oberen Ecke, lässt sich die Markererkennung pausieren und wieder aufnehmen. Dies erleichtert das Interagieren mit erkannten Prozessmodellen, da somit die Erkennung nicht aufrecht erhalten werden muss. Die Hand, welche das Gerät hält, muss die Kamera nicht mehr auf den Marker richten und wird entlastet.

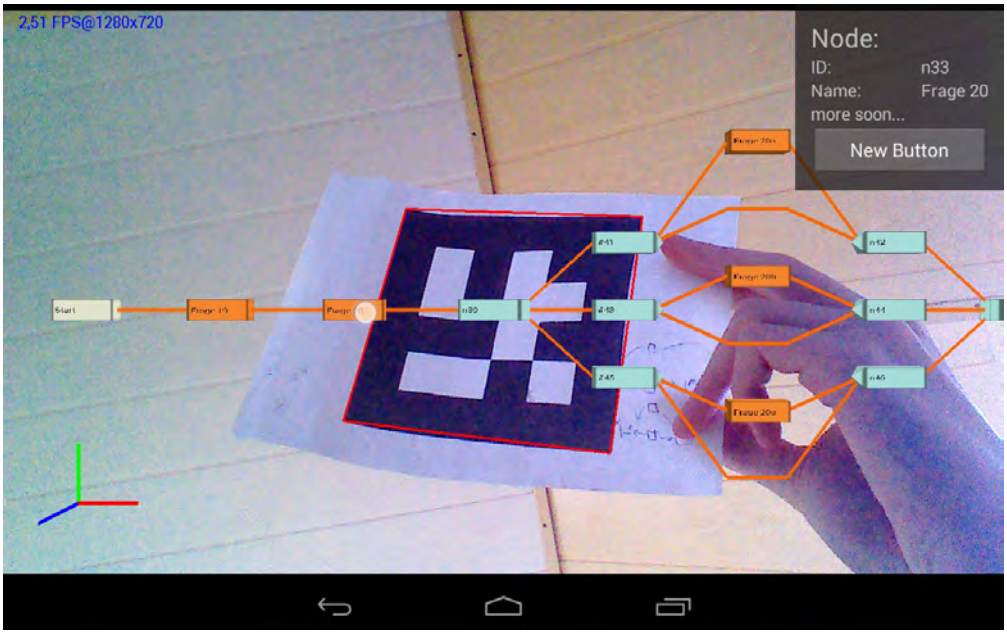

Abbildung 6.5.: Information zu einem Task

7

Anforderungsabgleich

Eine Bewertung der zuvor gestellten Anforderungen befindet sich in den nächsten beiden Tabellen. Die Bewertungsskala erstreckt sich von - - bis + +, wobei - - als mangelhaft und + + als sehr gut zu verstehen sind.

7.1. Notwendige Anforderungen

Notwendige Funktionen des Frameworks, damit es als vollständig betrachtet werden kann, sind in Tabelle 7.1 enthalten.

Kriterium	Beschreibung	Bewertung
Markererkennung	Marker sollen zuverlässig erkannt und unterschieden werden	+
Leistungsfähiger 3D-Renderer	Eine schnelle OpenGL ES 2.0 Rendering-Pipeline, welche hunderte Objekte mit flüssiger Framerate zeichnet	+
Flexibilität bei 3D-Modellen	3D-Modelle sollten flexibel einsetzbar und leicht zu integrieren sein	+
Einfacher XML-Prozess-Parser	Der Prozess-Parser soll leicht erweiterbar und robust funktionieren	+
Schnelle 3D-Modell-Generierung	Die 3D-Repräsentation eines Prozesses sollte in kurzer Zeit laden	++
Interaktion	Verschieben, Rotieren, Zoomen und Auswahl des Prozesses um weitere Information zu erhalten	++

Tabelle 7.1.: Bewertung der notwendigen Anforderungen, damit das Framework funktional komplett ist

7.2. Zusätzliche Anforderungen

Zusätzliche Kriterien, welche den Funktionsumfang erweitern, befinden sich in Tabelle 7.2.

Kriterium	Beschreibung	Bewertung
Pausieren der Markererkennung	Das Pausieren der Markererkennung soll dem Benutzer die Bedienung erleichtern. Die Kamera muss nicht ständig auf die Marker gerichtet werden.	++
Prozessdaten aus dem Internet	Prozessdaten könnten zusätzlich zum lokalen Speicher auch aus dem Internet geladen werden, damit mehr Flexibilität bei den Einsatzmöglichkeiten entsteht.	+

Tabelle 7.2.: Bewertung der zusätzlichen Anforderungen, außerhalb des minimal benötigten Funktionsumfangs

8

Zusammenfassung & Ausblick

Abschließend wird eine umfassende Zusammenfassung mit möglichen Verbesserungen gegeben und vorstellbare Anwendungsfälle für zukünftige Anwendungsgebiete dargelegt.

8.1. Zusammenfassung

Ziel dieser Arbeit war es, einen funktionsfähigen Prototypen zu erstellen, welcher im mobilen Umfeld Marker erkennt, und zugehörige, dreidimensionale Prozessmodelle darstellen kann. Außerdem stellt die Interaktion mit den dargestellten Prozessmodellen einen wichtigen Aspekt dar, worunter auch zusätzliche Information zu den einzelnen Prozesskomponenten fällt. Grundsätzlich kann dieses Ziel als erfüllt betrachtet werden, da der Prototyp in seiner aktuellen Form einsetzbar ist.

Der Prototyp arbeitet im Grunde sehr zuverlässig, dennoch besitzt er einige Defizite. Ein Kritikpunkt ist die Markererkennung. Diese ist etwas langsam, obwohl sie schon einen eigenen Thread zur Verfügung hat. Dies kann an OpenCV für Android liegen, oder an der Implementierung im verwendeten Imagine-Framework. Eine eigene Lösung könnte performanter ausfallen, wenn man, z.B. auch die Kantenerkennungsprozeduren als OpenGL Shader implementiert hätte. Dadurch könnte dann zwar die Performanz beim Zeichnen der Prozessmodelle beeinträchtigt werden, jedoch ist der Renderer gut optimiert und bietet somit noch viel Spielraum auf der GPU, der genutzt werden kann.
Der zweite Punkt ist der 3D-Modell-Parser. Der vollständige Funktionsumfang des OBJ-Formates wird nicht genutzt und somit kann es vorkommen, dass der Designprozess irritierend sein kann. Nach dem Import eines Objekts, welches in externen Tools erstellt wurde, kann es vorkommen, dass die Geometrie falsch interpretiert wird. Die Folge davon ist ein deformiertes Modell. Die Lösung kann eine zusätzliche Bibliothek sein, welche die benötigten Formate beherrscht oder man erweitert den bestehenden Parser.
Der Renderer selbst birgt auch einen Mangel. Das Zeichnen von Schriftarten kann weder von Mipmaps profitieren, noch von Antialiasing, was zu starken Artefakten führt. Distance field fonts wären hierfür eine angebrachte Lösung.
Zuletzt kann man den ADEPT-Prozessmodell-Parser bemängeln. Er unterstützt nicht den vollen Umfang der Spezifikation, wobei der Parser sich relativ einfach erweitern lässt. Jedoch war es für den Prototypen nicht erforderlich nach allen möglichen Prozesskomponenten zu parsen.
Alle aufgelisteten Defizite fallen nicht stark ins Gewicht, da sie die Nutzung nur minimal beeinträchtigen. Sie würden zwar der Vollständigkeit des Frameworks dienen, sind aber nicht zwingend erforderlich, und wurden deshalb vernachlässigt.

8.2. Ausblick

Augmented Reality in Kombination mit Business Prozess Modellen bietet, vor allem im mobilen Bereich, neue Möglichkeiten zur Interaktion mit Information, welche nur in Desktopumgebungen zugänglich ist. Durch die Erweiterung auf drei Dimensionen erhält

man noch mehr Interaktionsfreiheiten und die Bedienung fällt intuitiver aus. Beispielsweise könnte medizinisches Personal einen Behandlungsprozess schnell und einfach betrachten, bearbeiten oder beurteilen. Generell könnten medizinische Abteilungen ihre Tätigkeiten komplett, schnell und einfach, ohne Umweg zu Stift und Papier, auf mobilen Geräten verwalten [PMLR15, SSP+14, PTR10, PTKR10]. Wird zusätzlich noch Vitalsensorik der einzelnen Patienten in das System integriert, kann diese dazu verwendet werden, weitere Information zu verarbeiten [SSP+13]. Nicht nur der medizinische Sektor, sondern auch die Psychologie und Pädagogik würde von dieser Art der Handhabung der jeweiligen Aktivitäten profitieren [SSPR15, SRLP+13, PLRH12]. Werkstätten hätten die Möglichkeit komplizierte Wartungs-, Reparatur- und Montagearbeiten anhand von klar strukturierten Prozessen durchzuführen. Processor, als Prototyp, zeig jetzt schon auf, in welche Richtung sich BPM bewegen könnte. Es bleibt abzuwarten, ob sich dieser Trend in naher Zukunft durchsetzen wird.

A

Quelltexte

In diesem Anhang sind einige wichtige Quelltexte aufgeführt.

```
1  //////////////////////////////////////////////////////////////////////////////////
2  //   src\main\java\com\aj\processor\app\XML\Process [Line 753]
3  //
4  //                          CREATE LAYERED GRAPH STRUCTURE...
5  //
6  //
7  //////////////////////////////////////////////////////////////////////////////////
8
9  //         graph_layers
10
11 //get the start_flow node and create linear branch
12 PComponent start_node = getStartNode();
13 //[...]
14 ArrayList<PComponent> used_nodes = new ArrayList<PComponent>();
15 ArrayList<PComponent> layer_nodes = new ArrayList<PComponent>();
16 layer_nodes.add(start_node);
17
18 boolean work = true;
19 while(work){
20     graph_layers.add(layer_nodes);
```

```
21      ArrayList<PComponent> next_nodes = new ArrayList<PComponent>();
22      for(PComponent pc : layer_nodes){
23          ArrayList<PComponent> new_nodes = getNodesNextNodes(pc);
24          for(PComponent new_pc : new_nodes){
25              if(!used_nodes.contains(new_pc)){
26                  used_nodes.add(new_pc);
27                  next_nodes.add(new_pc);
28              }
29          }
30      }
31      layer_nodes = next_nodes;
32      if(next_nodes.size() == 0){
33          work = false;
34      }
35  }
36  //[...]
37
38
39  //we have filled the layers...
40  //now we need to stretch them out if needed...
41  //basically check if destination nodes are in the same layer as source nodes...
42  int layer_index = 0;
43  while(layer_index < graph_layers.size()){
44      ArrayList<PComponent> layer = graph_layers.get(layer_index);
45      boolean found = true;
46      while(found){
47          found = false;
48          int found_index = 0;
49          for(PComponent pc : layer){
50              //get previous nodes...
51              ArrayList<PComponent> previous_nodes = getNodesPreviousNodes(pc);
52              for(PComponent p_pc : previous_nodes){
53                  if(isNodeInLayer(p_pc, layer)){
54                      found = true;
55                      debugPComponent(pc);
56                      break;
57                  }
58              }
59              if(found){
60                  break;
61              }
62              found_index += 1;
63          }
64          if(found) {
65              //found one... use the index to move it one layer down
66              //get element to move down
67              PComponent element_to_move = layer.get(found_index);
68              layer.remove(found_index);
69              //check if next layer is existing
70              if ((layer_index + 1) >= graph_layers.size()) {
71                  //add new layer
72                  graph_layers.add(new ArrayList<PComponent>());
73              }
74              //add the found element to the next layer
75              graph_layers.get(layer_index + 1).add(element_to_move);
76          }
77      }
78      layer_index += 1;
79  }
80  //[...]
```

```
81  //check if we have edges which go over multiple nodes...
82  //divide those and insert dummy nodes...
83  /*       o = node         @ = dummyNode
84
85          ___                  ___
86           o                    o
87          -|-                  -|-
88           |       becomes      @
89          -|-                  -|-
90           o                    o
91          ___                  ___
92  */
93  ArrayList<PComponent> cut_edges = new ArrayList<PComponent>();
94  for(PComponent edge : edges_pc_unsorted_){
95      //TODO: check this...
96      //only check forward directed edges
97      if(isEdgeForwardDirected(edge)) {
98          if (isEdgeOverMultipleLayers(edge, graph_layers)) {
99              //we need to cut this edge...
100             cut_edges.add(edge);
101         }
102     }
103 }
104 //[...]
105 for(PComponent edge : cut_edges){
106     cutEdgeInGraphLayer(edge);
107 }
108 //[...]
109 //check all edges if they are crossing
110 //[...]
111 int max_loops = 1000;
112 int loop = 0;
113 boolean crossing_edges = true;
114 while(crossing_edges){
115     crossing_edges = false;
116     for(PComponent edge_l_1 : edges_pc_unsorted_) {
117         if (edge_l_1.hasEdge()) {
118             if (edge_l_1.getEdge() != null) {
119                 //only check forward directed edges
120                 if(isEdgeForwardDirected(edge_l_1)) {
121                     for (PComponent edge_l_2 : edges_pc_unsorted_) {
122                         if (edge_l_2.hasEdge()) {
123                             if (edge_l_2.getEdge() != null) {
124                                 //only check forward directed edges
125                                 if(isEdgeForwardDirected(edge_l_1)) {
126                                     if (areEdgesCrossing(edge_l_1, edge_l_2, graph_layers)) {
127
128                                         //ok edges are crossing, get their destination layer indices
129                                         //and swap their position in the layers
130
131                                         crossing_edges = true;
132
133                                         //get destination nodes
134                                         String destination_node_1_id = getEdgeDestinationID(edge_l_1);
135                                         String destination_node_2_id = getEdgeDestinationID(edge_l_2);
136
137                                         PComponent destination_node_1 = getStructuralNodeDataByNodeID(
138                                             destination_node_1_id
139                                         );
140                                         PComponent destination_node_2 = getStructuralNodeDataByNodeID(
```

```
141                                         destination_node_2_id
142                                     );
143
144                                     if ((destination_node_1 == null) || (destination_node_2 == null)) {
145                                         continue;
146                                     }
147
148                                     //get destination Nodes indices  of layer in graph_layers...
149                                     int graph_layer_index_destination_node_1 = getGraphLayerIndexOfNode(
150                                         destination_node_1,
151                                         graph_layers
152                                     );
153
154                                     //found nodes in layers
155                                     if (graph_layer_index_destination_node_1 >= 0) {
156                                         //swap pos in layer
157
158                                         ArrayList<PComponent> layer = graph_layers.get(
159                                             graph_layer_index_destination_node_1
160                                         );
161                                         int node_1_index = getIndexOfNodeInLayer(
162                                             destination_node_1,
163                                             graph_layers.get(graph_layer_index_destination_node_1)
164                                         );
165                                         int node_2_index = getIndexOfNodeInLayer(
166                                             destination_node_2,
167                                             graph_layers.get(graph_layer_index_destination_node_1)
168                                         );
169
170                                         layer.set(node_1_index, destination_node_2);
171                                         layer.set(node_2_index, destination_node_1);
172
173                                     }
174
175                                 }
176                             }
177
178                         }
179                     }
180                 }
181             }
182         }
183     }
184 }
185
186 loop += 1;
187 if(loop >= max_loops){
188     crossing_edges = false;
189 }
190 }
```

Algorithmus A.1: Layered Graph Sortierung

```
1  //////////////////////////////////////////////////////////////////////////////
2  //  src\main\java\com\aj\processor\app\graphics\world\ObjectWorld [Line 157]
3  //
4  //                Renderer Modell-Import sortiert
5  //
6  //////////////////////////////////////////////////////////////////////////////
7  //adding the model and its data into the simple_sorted arrays
8  private void addModelData_simple_sorted(CompositeObject co){
9      //first get he model if it has any...
10     if(!co.hasModel()){
11         return;
12     }
13     Model mdl = co.getModel();
14
15     //now get the model's meshs
16     ArrayList<Mesh> mdl_meshs = mdl.get_meshs();
17
18
19     if(mdl.isInstance()) {
20         //iterate trough the meshs and check if they are already sorted in...
21         //or in other words, if the material was already used in our lists...
22         for (Mesh mesh : mdl_meshs) {
23             //get the material...
24             Material mesh_mtl = mesh.get_material();
25             //check if the material was already used...
26             boolean sort_in = false;
27             int sort_in_index = -1;
28             for (int j = 0; j < material_mesh_simple_sorted.size(); j++) {
29                 if (mesh_mtl.get_name().equalsIgnoreCase(material_mesh_simple_sorted.get(j).get_name())) {
30                     //we found the material, so it was sorted in...
31                     //init the addition of the model data to the existing lists...
32                     sort_in_index = j;
33                     sort_in = true;
34                     break;
35                 }
36             }
37             if (sort_in) {
38                 //add the model data at the index of the material...
39                 mesh_model_simple_sorted.get(sort_in_index).add(mesh);
40                 model_mesh_simple_sorted.get(sort_in_index).add(mdl);
41                 compositeObjects_mesh_list_simple_sorted.get(sort_in_index).add(co);
42             } else {
43                 //model/material is not found in the lists, create a new entry
44                 material_mesh_simple_sorted.add(mesh_mtl);
45                 mesh_model_simple_sorted.add(new ArrayList<Mesh>());
46                 model_mesh_simple_sorted.add(new ArrayList<Model>());
47                 compositeObjects_mesh_list_simple_sorted.add(new ArrayList<CompositeObject>());
48                 int size_index = mesh_model_simple_sorted.size() - 1;
49                 mesh_model_simple_sorted.get(size_index).add(mesh);
50                 model_mesh_simple_sorted.get(size_index).add(mdl);
51                 compositeObjects_mesh_list_simple_sorted.get(size_index).add(co);
52             }
53         }
54     }
55     else{
56         //our model is not an instance so just sort the model's data in....
57         for (Mesh mesh : mdl_meshs) {
58             //get the material...
59             Material mesh_mtl = mesh.get_material();
```

```
60
61              material_mesh_simple_sorted.add(mesh_mtl);
62              mesh_model_simple_sorted.add(new ArrayList<Mesh>());
63              model_mesh_simple_sorted.add(new ArrayList<Model>());
64              compositeObjects_mesh_list_simple_sorted.add(new ArrayList<CompositeObject>());
65              int size_index = mesh_model_simple_sorted.size() - 1;
66              mesh_model_simple_sorted.get(size_index).add(mesh);
67              model_mesh_simple_sorted.get(size_index).add(mdl);
68              compositeObjects_mesh_list_simple_sorted.get(size_index).add(co);
69          }
70      }
71  }
```

Algorithmus A.2: Renderer: sortierter Modell-Import

```
1  ////////////////////////////////////////////////////////////////////////////////
2  //   src\main\java\com\aj\processor\app\OpenGLEngine [Line 338]
3  //
4  //                          Renderer SIMPLE vs SORTED...
5  //
6  ////////////////////////////////////////////////////////////////////////////////
7
8  /*
9          RENDER MODE SIMPLE
10 */
11 if(ow.getStoreMode() == ObjectWorld.store_mode_simple) {
12
13     //MODELS
14     GLES20.glEnableVertexAttribArray(locPositionSimple);
15     GLES20.glEnableVertexAttribArray(locTexCoordSimple);
16     GLES20.glEnableVertexAttribArray(locNormalSimple);
17     for (CompositeObject co : ow.getCompositeObjectsModels()) {
18         if (co.hasModel() && (co.getRenderType() == CompositeObject.render_standard)) {
19             Positation posi = co.getPositation();
20             m_m = posi.get_model_matrix();
21             pvm_m = pv_m.multiply(m_m);
22
23             //this call bind model data every call!!!
24             drawModel(co.getModel());
25         }
26     }
27     GLES20.glDisableVertexAttribArray(locPositionSimple);
28     GLES20.glDisableVertexAttribArray(locTexCoordSimple);
29     GLES20.glDisableVertexAttribArray(locNormalSimple);
30     //MODELS DONE
31 }
32 /*
33         RENDER MODE SIMPLE SORTED
34  */
35 else if(ow.getStoreMode() == ObjectWorld.store_mode_simple_sorted){
36     //get all the data first
37     ArrayList<ArrayList<CompositeObject> > compositeObjects_mesh_list_simple_sorted = ow.
         getCompositeObjects_mesh_list_simple_sorted();
38     ArrayList<ArrayList<Mesh> > mesh_model_simple_sorted = ow.getMesh_model_simple_sorted();
39     ArrayList<ArrayList<Model> > model_mesh_simple_sorted = ow.getModel_mesh_simple_sorted();
40     ArrayList<Material> material_mesh_simple_sorted = ow.getMaterial_mesh_simple_sorted();
41
42     //MODELS
43     GLES20.glEnableVertexAttribArray(locPositionSimple);
44     GLES20.glEnableVertexAttribArray(locTexCoordSimple);
45     GLES20.glEnableVertexAttribArray(locNormalSimple);
46
47     GLES20.glUseProgram(programSimple);
48
49     int mtl_index = 0;
50     for (Material mtl : material_mesh_simple_sorted) {
51         //set up the texture
52         // Set the active texture unit to texture unit 0.
53         GLES20.glActiveTexture(GLES20.GL_TEXTURE0);
54         // Bind the texture to this unit.
55         GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, mtl.get_diffuse_map_texture());
56         // Tell the texture uniform sampler to use this texture
57         // in the shader by binding to texture unit 0.
58         GLES20.glUniform1i(locTextureSimple, 0);
```

```
59
60          int co_index = 0;
61          for(CompositeObject co : compositeObjects_mesh_list_simple_sorted.get(mtl_index)) {
62              if (co.hasModel() && (co.getRenderType() == CompositeObject.render_standard)) {
63                  Positation posi = co.getPositation();
64                  m_m = posi.get_model_matrix();
65                  pvm_m = pv_m.multiply(m_m);
66
67                  // Pass in the position information
68                  Mesh mesh = mesh_model_simple_sorted.get(mtl_index).get(co_index);
69
70                  if (mesh.isLoaded()) {
71
72                      GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, mesh.get_vertex_vbo());
73                      GLES20.glVertexAttribPointer(locPositionSimple, 3, GLES20.GL_FLOAT, false, 0, 0);
74
75                      GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, mesh.get_texcoord_vbo());
76                      GLES20.glVertexAttribPointer(locTexCoordSimple, 3, GLES20.GL_FLOAT, false, 0, 0);
77
78                      GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, mesh.get_normal_vbo());
79                      GLES20.glVertexAttribPointer(locNormalSimple, 3, GLES20.GL_FLOAT, false, 0, 0);
80
81                      //TRANSPOSE
82                      render_mat = pvm_m.getFloatArray(true);
83
84                      GLES20.glUniformMatrix4fv(locMVPMatrixSimple, 1, false, render_mat, 0);
85                      if (mesh.get_triangle_count() == 0) {
86                          return;
87                      }
88                      GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, mesh.get_triangle_count() * 3);
89                  }
90              }
91              co_index += 1;
92          }
93          mtl_index += 1;
94      }
95      GLES20.glDisableVertexAttribArray(locPositionSimple);
96      GLES20.glDisableVertexAttribArray(locTexCoordSimple);
97      GLES20.glDisableVertexAttribArray(locNormalSimple);
98      //MODELS DONE
99  }
100
101 //LINES
102 GLES20.glEnableVertexAttribArray(locPositionLine);
103 GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, 0);
104 for (CompositeObject co : ow.getCompositeObjectsLines()) {
105     if (co.hasLine() && (co.getRenderType() == CompositeObject.render_standard)) {
106         Positation posi = co.getPositation();
107         pvm_m = pv_m.multiply(posi.get_model_matrix());
108         drawLine(co.getLine());
109     }
110 }
```

Algorithmus A.3: Renderer: einfach oder sortiert

B

Glossar

3D	Bezieht sich auf den dreidimensionalen Raum.
3D-Modell	Dreidimensionale Repräsentation eines Objektes; Meistens zusammengestellt aus mehreren Geometrien.
Android	Von Google entwickeltes mobiles Betriebssystem.
Augmented Reality	Computergestützte Realitätswahrnehmung durch zusätzliche Information.
Distance field fonts	Pixel einer Font-Textur enthalten die kürzeste Distanz zu Pixeln, welche Information zur Font haben; Schriftarten leiden dadurch nicht mehr unter Artefakten bei unterschiedlichen Distanzen zur Kamera.
Framebuffer	Buffer, welcher ein gezeichnetes, vom Computer generiertes, Bild enthält.

Geometrie	Monolithisches Objekt im dreidimensionalen Raum; Meistens aus Dreiecken aufgebaut; Bilden Modelle ab.
Imagine	Framework zur Markererkennung für Android.
Marker	Eindeutig identifizierbares und unterscheidbares Muster, welches in digitalen Bildern zur Positionsbestimmung genutzt wird.
Mipmaps	Texturen werden in mehreren Auflösungen gespeichert um bei unterschiedlicher Distanz zur Kamera Textur-Artefakte in Form von flackernden Pixeln zu vermeiden.
OpenCV	Programmbibliothek mit Algorithmen zur Bildverarbeitung.
OpenGL ES 2.0	Plattform übergreifende Bibliothek für hardware beschleunigte Grafik für eingebettete Geräte.
Processor	Name des, im Rahmen dieser Arbeit entwickelten, Frameworks.
Shader	Bestandteil eines Shaderprogramms; Programmierbare Stufe einer Grafikbibliothek; Hier spezifisch OpenGL ES 2.0; Vertex - oder Fragmentshader.
Shaderprogramm	Besteht aus Vertex - und Fragmentshader. Enthält implementierte Stufen der Grafikpipeline einer Grafikbibliothek. Definiert wie Fragmente und Geometrien gezeichnet werden; Hier spezifisch OpenGL ES 2.0.
Task	Eine Aufgabe oder Aktivität in einem Prozessmodell; Im Framework als Normal Node genannt.
Textur	Zweidimensionales Bild; Wird auf Geometrien projiziert; Auch als Texture Map bezeichnet; Hier hauptsächlich diffuse Textur.
VBO	Vertex Buffer Object; Speicher auf der GPU; Enthält meistens Vertices, Texturcoordinaten und Normalen.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[And13] *Android Developers: Parsing XML Data*. `http://developer.android.com/training/basics/network-ops/xml.html`. Version: 2013. – Aufgerufen: 05.01.2015

[BES00] BALLEGOOIJ, A. ; ELLIENS, A. ; SCHÖNHAGE, B.: *3D Gadgets for Business Process Visualization – A Case Study*. In: Proceedings of the fifth symposium on Virtual reality modelling language, 2000

[BRB07] BOBRIK, R. ; REICHERT, M. ; BAUER, T.: *View-based process visualization*. In: 5th International Conference on Business Process Management, Springer, 2007

[DB07] DECKER, G. ; , A. Grosskopf ; BARROS, A.: *A Graphical Notation for Modeling Complex Events in Business Processes*. In: Proceedings of the 11th IEEE international Enterprise Distributed Object Computing Conference, IEEE Computer Society, Washington, DC, 2007

[DKK14] DRAWEHN, Jens ; KOCHANOWSKI, Monika ; KÖTTER, Falko: *Business Process Management Tools 2014*. Fraunhofer Verlag, 2014

[DKR+95] DADAM, P. ; KUHN, K. ; REICHERT, M. ; BEUTER, T. ; NATHE, M.: ADEPT: Ein integrierender Ansatz zur Entwicklung flexibler, zuverlässiger kooperierender Assistenzsysteme in klinischen Anwendungsumgebungen. In: *Proc. GI-Jahrestagung (GISI '95)*, 1995

[DR09] DADAM, P. ; REICHERT, M.: The ADEPT Project: A Decade of Research and Development for Robust and Flexible Process Support - Challenges

and Achievements. In: *Computer Science - Research and Development* 23 (2009), Nr. 2

[DRRM+09] DADAM, P. ; REICHERT, M. ; RINDERLE-MA, S. ; GOESER, K. ; KREHER, U. ; JURISCH, M.: Von ADEPT zur AristaFlow BPM Suite - Eine Vision wird Realität: "Correctness by Constructionünd flexible, robuste Ausführung von Unternehmensprozessen. University of Ulm, Faculty of Electrical Engineering and Computer Science, January 2009 (UIB-2009-02). – Technical Report

[ESK05] EIGLSPERGER, M. ; SIEBENHALLER, M. ; KAUFMANN, M. ; PACH, J. (Hrsg.): *An Efficient Implementation of Sugiyama's Algorithm for Layered Graph Drawing*. Springer Berlin Heidelberg, 2005 (Lecture Notes in Computer Science)

[Har13] HARTMANN, T.: *Implementation of a Java Framework for Marker Based Detection in Augmented Reality*. 2013

[KGM99] KRALLMANN, H. ; GU, F. ; MITRITZ, A.: *ProVision3D - Eine Virtual Reality Workbench zur Modellierung, Kontrolle und Steuerung von Geschäftsprozessen im virtuellen Raum*. Wirtschaftsinformatik 41(1)(in German), 1999

[KP04] KINDLER, E. ; PÁLES, C.: *3D-Visualization of Petri Net Models: Concept and Realization*. In: Proceedings of International Conference of Applications and Theory of Petri Nets, Springer, 2004

[LG07] LI, L. ; , J. Hosking ; GRUNDY, J.: *Visual Modelling of Complex Business Processes with Trees, Overlays and Distortion-based Displays*. In: Proceedings of the IEEE Symposium on Visual Languages and Human Centric Computing VLHCC, IEEE Computer Society, Washington, DC, 2007

[LOG03] LENZ, K. ; OBERWEIS, A. ; GRUNDY, J.: *Inter-Organizational Business Process Management with XML Nets*. In: Petri Net Technology for Communication-Based Systems, Advances in Petri Nets volume 2472 of LNCS, Springer-Verlag, 2003

[MS06] MENDLING, J. ; SIMON, C.: *Business Process Design by View Integration*. In: Business Process Management Workshops, Springer, 2006

[PLRH12] PRYSS, R. ; LANGER, D. ; REICHERT, M. ; HALLERBACH, A.: Mobile Task Management for Medical Ward Rounds - The MEDo Approach. In: *1st Int'l Workshop on Adaptive Case Management (ACM'12), BPM'12 Workshops*, Springer, September 2012 (LNBIP 132)

[PMLR15] PRYSS, R. ; MUNDBROD, N. ; LANGER, D. ; REICHERT, M.: Supporting medical ward rounds through mobile task and process management. In: *Information Systems and e-Business Management* 13 (2015), February, Nr. 1

[PMR13] PRYSS, R. ; MUSIOL, S. ; REICHERT, M.: Collaboration Support Through Mobile Processes and Entailment Constraints. In: *9th IEEE Int'l Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom'13)*, IEEE Computer Society Press, October 2013

[PMR14] PRYSS, R. ; MUSIOL, S. ; REICHERT, M.: Integrating Mobile Tasks with Business Processes: A Self-Healing Approach. In: *Handbook of Research on Architectural Trends in Service-Driven Computing*. 2014

[PTKR10] PRYSS, R. ; TIEDEKEN, J. ; KREHER, U. ; REICHERT, M.: Towards Flexible Process Support on Mobile Devices. In: *Proc. CAiSE'10 Forum - Information Systems Evolution*, Springer, 2010 (LNBIP 72)

[PTR10] PRYSS, R. ; TIEDEKEN, J. ; REICHERT, M.: Managing Processes on Mobile Devices: The MARPLE Approach. In: *CAiSE'10 Demos*, 2010

[Rö07] RÖLKE, H.: *3-D Petri nets*. In: Petri Net Newsletter, 2007 (72)

[Sch99] SCHEER, A.-W.: *ARIS – Business Process Modeling*. Springer Verlag, Berlin, 1999

[SRLP+13] SCHOBEL, J. ; RUF-LEUSCHNER, M. ; PRYSS, R. ; REICHERT, M. ; SCHICKLER, M. ; SCHAUER, M. ; WEIERSTALL, R. ; ISELE, D. ; NANDI, C. ; ELBERT, T.: A generic questionnaire framework supporting psychological studies

with smartphone technologies. In: *XIII Congress of European Society of Traumatic Stress Studies (ESTSS) Conference*, 2013

[SSP+13] SCHOBEL, J. ; SCHICKLER, M. ; PRYSS, R. ; NIENHAUS, H. ; REICHERT, M.: Using Vital Sensors in Mobile Healthcare Business Applications: Challenges, Examples, Lessons Learned. In: *9th Int'l Conference on Web Information Systems and Technologies (WEBIST 2013), Special Session on Business Apps*, 2013

[SSP+14] SCHOBEL, J. ; SCHICKLER, M. ; PRYSS, R. ; MAIER, F. ; REICHERT, M.: Towards Process-Driven Mobile Data Collection Applications: Requirements, Challenges, Lessons Learned. In: *10th Int'l Conference on Web Information Systems and Technologies (WEBIST 2014), Special Session on Business Apps*, 2014

[SSPR15] SCHOBEL, J. ; SCHICKLER, M. ; PRYSS, R. ; REICHERT, M.: Process-Driven Data Collection with Smart Mobile Devices. In: *Web Information Systems and Technologies - 10th International Conference, WEBIST 2014, Barcelona, Spain, Revised Selected Papers*. Springer, 2015 (LNBIP)

[WBR10] WEST, S. ; BROWN, R. A. ; RECKER, J. C.: *Collaborative business process modeling using 3D virtual environments*. In: Proceedings of the 16th Americas Conference on Information Systems : Sustainable IT Collaboration around the Globe, Association for Information Systems (AIS), 2010

Name: Johann Albach Matrikelnummer: 755269

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Johann Albach