Institut für Visualisierung und Interaktive Systeme
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3300

Molekularvisualisierung und Remote-Steuerung auf mobilen Endgeräten

Jing Sheng

Studiengang:	Informatik
Prüfer:	Prof. Dr. Thomas Ertl
Betreuer:	Dipl.-Inf. Michael Krone
begonnen am:	20. Februar 2012
beendet am:	21. August 2012
CR-Klassifikation:	H.5.2, I.3.2, I.3.2, J.3

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1 Einleitung

Die vorliegende Diplomarbeit beschreibt die Entwicklung eines Visualisierungsprogramms von Moleküldaten, das auf einem Android-Tablet-PC lauffähig ist. Die Android-Applikation soll hierbei an die Visualisierungssoftware MegaMol angelehnt sein und grundlegende Proteinvisualisierungen darstellen können. Außerdem soll das System über eine Netzwerkverbindung mit MegaMol kommunizieren können. Die Android-Software agiert dann als Client, welcher erstens die gleichen Daten darstellt wie MegaMol und zweitens auch zur Steuerung der PC-/Powerwall-Visualisierung verwendet werden kann. Vorgesehene Interaktionen sind die Kamerasteuerung über Touchscreen-Gesten und die Modifikation der Visualisierungsparameter des MegaMol-Hauptprogramms ausgehend vom Mobilgerät.

Für die graphische Darstellung auf dem Mobilgerät soll die Leistungsfähigkeit aktueller Android-Geräte mit Nvidia Tegra 3 Chipsatz hinsichtlich der Darstellung von 3D-Graphik evaluiert werden. Die Entwicklung der Android-Applikation erfolgt in OpenGL ES 2.0 in Verbindung mit Java; die Implementierung der Programmteile im MegaMol-Framework erfordert C/C++. Nach Möglichkeit wird die am Institut entwickelte Hilfsbibliothek VisLib eingesetzt.

Die Aufgaben erstrecken sich insgesamt über viele verschiedene Bereiche: von Visualisierungstechnik bis zur Applikationsentwicklung auf Mobilgeräten; von graphischer Programmierung bis zur Remote-Kontrolle über eine Netzwerkverbindungen. Die schriftliche Ausarbeitung führt Schritt für Schritt durch die einzelnen Schwerpunkte.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Motivation und Vorüberlegung erläutert die Motivation dieser Diplomarbeit und die Vorüberlegung der Grundbausteine.

Kapitel 3 – Android-Grundlagen erklärt Grundlagen der Entwicklung auf Android-Systemen. Folgende Fragen sollen beantwortet werden: Welche Tools benötigt man für die Entwicklung von Android-Applikationen? Aus welchen Komponenten besteht eine Android-Applikation? Welche Ressourcen gibt es neben dem Quellcode?

Kapitel 4 – Grundlagen von OpenGL ES führt ein in die Entwicklung von OpenGL ES. Die Fragestellung lautet hier: Wo ist der Ursprung von OpenGL ES zu suchen und wie ist der Zusammenhang zu OpenGL? Darüberhinaus werden die Unterschiede zwischen

verschiedenen OpenGL-ES-Versionen vorgestellt und schliesslich wird die Anwendung von OpenGL ES in der Android-Applikationsentwicklung knapp beschrieben.

Kapitel 5 – Architektur der gesamten Arbeit erklärt die Architektur der gesamten Arbeit, aufgeteilt in zwei Teile. Ein Großteil behandelt die Struktur der Applikation. Hier werden zusätzlich diejenigen Android-APIs erläutert, welche eine wichtige Rolle in der Applikation spielen. Um mit der Applikation reibungslos zusammenarbeiten zu können benötigt MegaMol zusätzliche Klassen, welche ebenfalls in diesem Kapitel vorgestellt werden. Der zweite Teil umfasst diese Erweiterungen.

Kapitel 6 – Schwierigkeiten und Lösungen Die Applikationsentwicklung für MegaMol ist auch mit Schwierigkeiten verbunden. Ausgewählte Probleme und Lösungsmöglichkeiten werden hier ausführlich diskutiert.

Kapitel 7 – Ausblick stellt Punkte vor, an welche eine zukünftige Entwicklung der Applikation anknüpfen kann.

Kapitel 8 – Zusammenfassung fasst schließlich die Ergebnisse der Arbeit zusammen.

2 Motivation und Vorüberlegung

2.1 Motivation

Die Diplomarbeit hat einen starken Zusammenhang zu der an der Universität Stuttgart entwickelten Visualisierungssoftware MegaMol™. MegaMol visualisiert punktbasierte Molekül-Datensätze. Die Benutzeroberfläche kann dabei mit Maus bzw. Space Mouse gesteuert werden.

Im Rahmen dieser Diplomarbeit wurde eine Android-Applikation entwickelt, im Folgenden auch *Applikation* bezeichnet, welche unter Anderem zur Fernsteuerung von MegaMol dient. Das Tablet bietet sich hierfür als Remote-Steuerung an, wenn MegaMol nicht an einem Desktop-PC eingesetzt, sondern als Demo an eine Leinwand projiziert oder von einem versteckten Server an einer Powerwall präsentiert wird.

Außer der Realisierung der Datenübertragung durch eine Socketverbindung zwischen Tablet und Server ist die Remote-Visualisierung eine weitere Entwicklungsmöglichkeit. Remote-Visualisierung hießt, dass der Server-PC die Dreieck-Struktur des Moleküls bzw. ein ganzes Bild berechnet und solang eine Internetverbindung vorhanden ist, kann das Tablet das vom MegaMol-Server berechnete Ergebnis darstellen, egal wo das Tablet sich befindet.

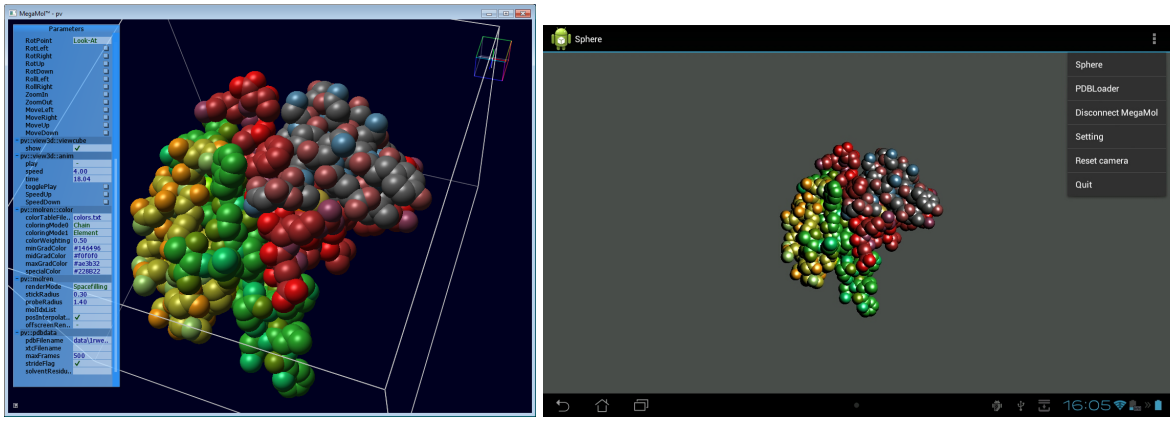


Abbildung 2.1: MegaMol und Applikation - Links: Benutzeroberfläche von MegaMol Rechts: Tablet-Applikation

Auch wenn sie nicht mit dem MegaMol-Server verbunden ist, kann die Applikation eine einfache Molekül-Visualisierung auf dem Tablet-PC darstellen. Die Applikation besitzt

eine eigene Funktion, um Molekül-Datensätze von der Online-Datenbank [BWF+00] im sogenannten PDB-Format herunterzuladen und zu visualisieren.

2.2 Ausstattung/Hardware

Als Hardware für diese Arbeit wurde ein Tablet-PC, im Folgenden auch *Tablet* bezeichnet, mit Android-Betriebssystem als Remote-Gerät ausgewählt.

Das Vorhaben war, eine Applikation für ein mobiles Gerät zu entwickeln. Im Gegensatz zu einem Smartphone, das herkömmlicherweise über eine recht kleine Bildschirmdiagonale verfügt, bietet ein Tablet weitaus mehr Raum für Datenvisualisierung und Bedienung.

Außer der höheren Bildschirmgröße bietet ein aktuelles Tablet im Vergleich zu fast allen Smartphones bessere Hardware und höhere Verarbeitungsleistung an. Das verwendete Tablet ist ein Asus Transformer Prime TF201 (Abbildung 2.2). Einige für diese Arbeit wichtigen Eigenschaften sind in Tabelle 2.1 aufgelistet.

Prozessor	NVIDIA® Tegra® 3 Quad-Core CPU
Betriebssystem	Android™ 4.0.3
Bildschirm	10.1 Zoll LED Backlight WXGA (1280x800) Bildschirm unterstützt 10-Finger Multi-Touch
Arbeitsspeicher	1GB
Speicherplatz	32GB
Wireless Data Network	WLAN 802.11 b/g/n@2.4GHz Bluetooth V3.0+EDR
Akku	Pad: 12 Stunden; 25Wh Lithium-Polymer Pad mit Dock: 18 Stunden; 25Wh + 22Wh Lithium-Polymer
Gewicht	Pad: 598g Pad mit Dock: 1135g

Tabelle 2.1: Spezifikationen des Asus Transformer Prime TF201 (Quelle: http://www.asus.com/Eee/Eee_Pad/Eee_Pad_Transformer_Prime_TF201/#specifications)

NVIDIA Tegra 3, mit Codenamen Kal-El, ist der erste Quad-Core-Mobilprozessor. Er wurde Februar 2011 angekündigt und erschien im November 2011 auf dem Markt. Die schnelle Akzeptanz seitens der Tablet-Hersteller führte dazu, dass Tegra 3 zur Zeit in 8 Tablet-Modellen verbaut wird. Zusammen mit dem Dual-Core-Mobilprozessor Tegra 2 gibt es nun

Abbildung 2.2: ASUS Transformer Prime TF201 (Quelle: http://www.asus.com/Eee/Eee_Pad/Eee_Pad_Transformer_Prime_TF201/)

bereits knapp 20 verschiedene Tablet-Modelle aller führenden Android-Tablet-Hersteller, die NVIDIAs Tegra-Serie benutzen.

Der Einsatz von Multi-Core-Prozessoren bringt speziell im Hinblick auf Mobilgeräte viele Vorteile mit sich, wie z.B. höhere Leistung und niedrigeren Stromverbrauch. Besonders im Multitasking oder bei Aufgaben mit mehreren Threads kommt dieser Mehrwert zum Tragen. Für detaillierte Benchmark-Ergebnisse siehe [Dua10] und [Qua11].

Tegra-Prozessoren bieten neben einigen offiziellen Extensions von OpenGL ES zusätzlich auch eigene an [NVI11]. OpenGL ES wird für die graphische Teilaufgaben der Applikation verwendet und die erwähnten Extensions sind in vielen Fällen hilfreich, wie Kapitel 6.2 belegt.

Der aktuelle Stand der Software ist Android 4.0.3. Android 4.1 wurde vor Kurzem veröffentlicht, ein Softwareupdate von ASUS ist allerdings noch nicht verfügbar.

2.3 Highlights des Android-Systems

Seit der Vorstellung des ersten Android-Smartphones im Jahre 2008 entwickelte sich die Plattform stetig zum heutigen meistverkauften Smartphone-Betriebssystem. Ende 2010 war Android bereits das zweitbestverkaufte Smartphone-System und im Jahre 2011 nahm Android einen weitere Hürde mit dem Einstieg in den Tablet-PC-Markt. Die zu Beginn 2012 veröffentliche Plattform 4.0 vereinheitlichte schließlich die zuerst getrennt geführte Smartphone- und Tablet-Linie. Hinzukommt, dass Android im 1. Quartal 2012 bereits mehr als die Hälfte des weltweiten Smartphone-Absatzes einnahm. Tabelle 2.2 zeigt die Anteile im Zeitraum 2009 bis zum 1. Quartal 2012.

	Q1 2009	Q3 2009	Q1 2010	Q3 2010	Q1 2011	Q3 2011	Q1 2012
Android	1,6%	3,5%	9,6%	25,3%	36,4%	52,5%	56,1%
Bada	-	-	-	1,1%	1,9%	2,2%	2,7%
iOS	10,5%	17,1%	15,4%	16,6%	16,9%	15%	22,9%
RIM	20,6%	20,7%	19,7%	15,4%	13%	11%	6,9%
Symbian	48,8%	44,6%	44,2%	36,3%	27,7%	16,9%	8,6%
Windows*	10,2%	7,9%	6,8%	2,7%	2,6%	1,5%	1,9%
Sonst	8%	6,2%	4,3%	2,6%	1,5%	0,9%	0,9%

Tabelle 2.2: Anteile der Betriebssysteme am Smartphone-Absatz vom 2009 bis 2012, jeweils 1. und 3. Quartal (Quelle: Gartner)
* Inklusive Windows Mobile und Windows Phone 7

Es ist jedoch auch wichtig, ein weiteres Detail anzumerken: Obwohl verschiedene Statistiken einen rasanten Zuwachs im Smartphone-Markt zeigen, ist dies allerdings nicht zwingend mit Gewinn gleichzusetzen, da Android auch häufig auf günstigen Einsteiger-Smartphones zu finden ist.

Im Ländervergleich nahm insbesonders der Anteil der Android-Benutzer in Deutschland rasant zu. Von 2011 bis 2012 verdoppelte sich hier der Anteil.

	Q1 2011	Q1 2012
Android	17%	40%
iOS	21%	22%
RIM	4,5%	3%
Symbian OS	42%	24%
Windows	11%	7%
Sonstige	4,5%	4%

Tabelle 2.3: Anteile der Betriebssysteme an aktivierten Smartphones in Deutschland, jeweils 1. Quartal in 2011 und 2012 (Quelle: comScore)

Trotz dem kürzlich vorgestelltem Windows Phone 8, welches Microsoft vorraussichtlich zu einem höheren Marktanteil verhelfen wird, soll Android laut aktuellen Prognosen auch im Jahre 2016 noch die Hälfte des Weltmarktes einnehmen:

	2012	2016	CAGR
Android	61%	52,9%	9,5%
iOS	20,4%	19%	10,9%
RIM	6%	5,9%	12,1%
Windows*	5,2%	19,2%	46,2%
Sonstige	7,2%	3%	-5,4%

Tabelle 2.4: Prognose zu Marktanteil der mobilen Betriebssysteme am Smartphone-Absatz 2012, 2016 und die Compound Annual Growth Rate (Quelle: IDC Worldwide Mobile Phone Tracker, 6.6.2012)
* Inklusive Windows Mobile und Windows Phone 7

Hierbei stellt sich die Frage, wieso Android so beliebt bei Smartphone-Herstellern und Kunden ist.

[Gar11] merkt an, dass Android sehr viel vom Open-Source-Betriebssystem Linux profitiert, auf welches Android aufsetzt. Ähnlich wie Linux lässt sich Android auf unterschiedlichen Hardwarearchitekturen kompilieren. Aus diesem Grunde kann jeder Benutzer oder Hardware-Hersteller das System weiterentwickeln und Android an eine Vielzahl von Smartphone-Geräten anpassen. [Ble12] erwähnt einen weiteren, offensichtlichen Vorteil von Open-Source-Software: sie ist kostenlos. Das ermöglicht Dritten, Einsteiger-Smartphones mit niedrigem Preis anzubieten.

Vor diesem Hintergrund ist es nicht verwunderlich, dass es mehrere Hundert unterschiedliche Smartphone-Modelle mit Android gibt (viele davon sind Multi-Core-Geräte), wohingegen sich iOS und Windows Phone 7 auf ca. 20 Modelle beschränken. Konkurrenz belebt den Markt und bringt damit auch die Weiterentwicklung von Android voran. Davon profitiert vor allem auch der Endbenutzer.

Entscheidend für die Akzeptanz eines Smartphone-Betriebssystems sind für die Benutzer außer dem Preis auch die verfügbaren Applikationen.

Die Entwicklung von Android-Applikationen ist nicht kostenpflichtig, siehe [Ble12], während Entwickler für andere Plattformen generell eine jährliche Gebühr einkalkulieren müssen, um die Applikationen auf dem Applikation-Marktplatz verfügbar zu machen. Auch die einmalige Anmeldungsgebühr des Google Play Stores kann der Entwickler vermeiden, da es eine Vielzahl an Stores oder Markets für Android-Applikationen gibt und diese frei wählbar sind. Infolge der freien Wahl eines Applikation-Marktplatzes hat der Benutzer auch die Freiheit, Applikationen ohne eine Registrierung bei Google zu installieren.

Android besitzt jedoch auch Nachteile. Die oben beschriebene Freiheit erhöht die Wahrscheinlichkeit, bösartige Applikationen anzutreffen. Jedem Betreiber eines Applikation-Marktplatzes ist die Überwachung und Kontrolle seines Angebots selbst überlassen. Kommt es zu Komplikationen, steht unter Umständen dessen Ruf auf dem Spiel.

Ein weiteres negatives Merkmal ist eine Folge der Gerätevielfalt. Soll eine Applikation alle denkbaren Hardwareausstattungen und unterschiedlichen Android-Versionen abdecken, steigt der Schwierigkeitsgrad der Entwicklung rapide an. Auch Debugging und Fehlerbehebungen werden dadurch komplexer.

3 Android-Grundlagen

3.1 Installation der Entwicklungstools

Die Android-Entwicklung stützt sich auf die Programmiersprache Java und der Systemumfang an Bibliotheken entspricht dem der Java Standard Edition.

Bleske [Ble12] nennt folgende Voraussetzungen, die erfüllt sein müssen, um eine Android-Applikation entwickeln zu können: Das Java Software Development Kit (auch *Java-SDK* oder *JDK*), das Android Software Development Kit (auch *Android-SDK*) und ein Android-Emulator. Eine echte Android-Hardware ist in vielen Fällen obligatorisch, da es Funktionen gibt, welche mit dem Emulator nicht zu simulieren sind, wie z.B. Sensoren. OpenGL ES wird vom Emulator unter SDK Tools Revision 17, die am Anfang April 2012 veröffentlicht wurde, bzw. SDK API Level 15 Revision 3 noch nicht unterstützt. In diesem Fall ist der Einsatz eines Android-Gerätes erforderlich. Auch für realitätsnahe Tests ist ein Endgerät daher unabdingbar.

In dieser Arbeit wird das Java-Entwicklungstool Eclipse verwendet. Neben dem Standard-Android-SDK bietet Android für Eclipse ein „Android Developer Tool"-Plug-In (auch abgekürzt als ADT), das bei der Kompilierung einer Android-Applikation unterstützt.

Die Installation des Android-SDK umfasst auch den Emulator. Außer Büchern wie [Gar11] und [Ble12] gibt es zahlreiche Online-Tutorials über die Installation der Android-Entwicklungstools. Wichtig dabei ist Auswahl der gewünschten Emulator-Plattform. Abbildung 3.1 zeigt den Android-SDK-Manager und Emulator für Plattform 4.0.3.

3.2 Application Package

Eine auf einem Android-Gerät ausführbare Applikation liegt in Form einer Application-Package-Datei (auch *APK-Datei*) vor. Sie besteht aus den folgenden drei Teilen [Gar11]:

1. **Dalvik-Binärdateien** sind durch die virtuelle Maschine Dalvik konvertierter Binärcode, sie sind auf dem Android-System direkt ausführbar.
2. **Ressourcen** enthält Bilder, Audio, Video und sämtliche XML-Dateien, welche die Applikation benötigt.
3. **Native Bibliotheken** enthalten nativen Code aus anderen Programmiersprachen wie z.B. C/C++ oder OpenGL-Bibliotheken. Dieser Teil ist optional.

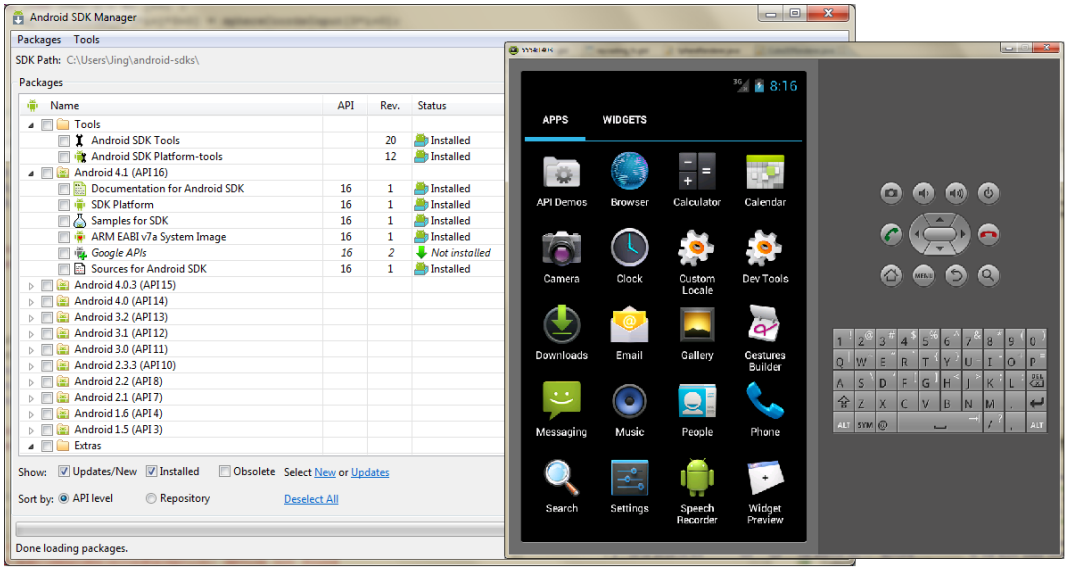


Abbildung 3.1: Android-SDK-Manager und Emulator

Alle drei Teile werden durch das Android-SDK in eine einzige APK-Datei kompiliert und sind damit auf Android-Geräten ausführbar.

3.3 Begrifflichkeiten

Bestandteile eines Android-Projektes

Bei der Erstellung eines neuen Android-Projektes bekommt man ein Entwicklungspackage mit bestimmten Bestandteilen. Die Struktur ähnelt der aus Abbildung 3.2.

Eine der wichtigsten Bestandteile einer Applikation ist die **AndroidManifest.xml**-Datei (auch *Manifest-Datei*). Sie informiert das System, aus welchen Komponenten die vorliegende Applikation besteht. Außerdem enthält die Manifest-Datei grundlegende Informationen über die Applikation sowie das minimale API Level (API: application programming interface), welches im kommenden Abschnitt vorgestellt wird; dazu zählen auch Informationen zu Zugriffsberechtigungen auf bestimmte Elemente oder Hardware, Voraussetzung an Hardware- und Software-Eigenschaften, die erforderten Bibliotheken und weitere Daten.

Die **R.java**-Datei wird automatisch generiert, wenn in Eclipse „Project → Build Automatically“ ausgewählt ist. Sie ist die Ressource-Index-Datei und notiert alle Änderung von Dateien im Ressource-Ordner.

Der **res**-Ordner (Ressource-Ordner) enthält Ressourcen wie Bilder, Icons, Audio, Video und andere Elemente, die das Projekt benötigt. Neben den meisten XML-Dateien befinden sich hier auch Shader-Dateien.

Die **main.xml**-Datei ist ein weiterer wichtiger Teil eines Projekts. Sie wird verwendet, um das Haupt-Layout festzulegen. In der vorliegenden Arbeit übernimmt jedoch OpenGL ES die Darstellung der graphischen Oberfläche und daher muss main.xml nicht editiert werden.

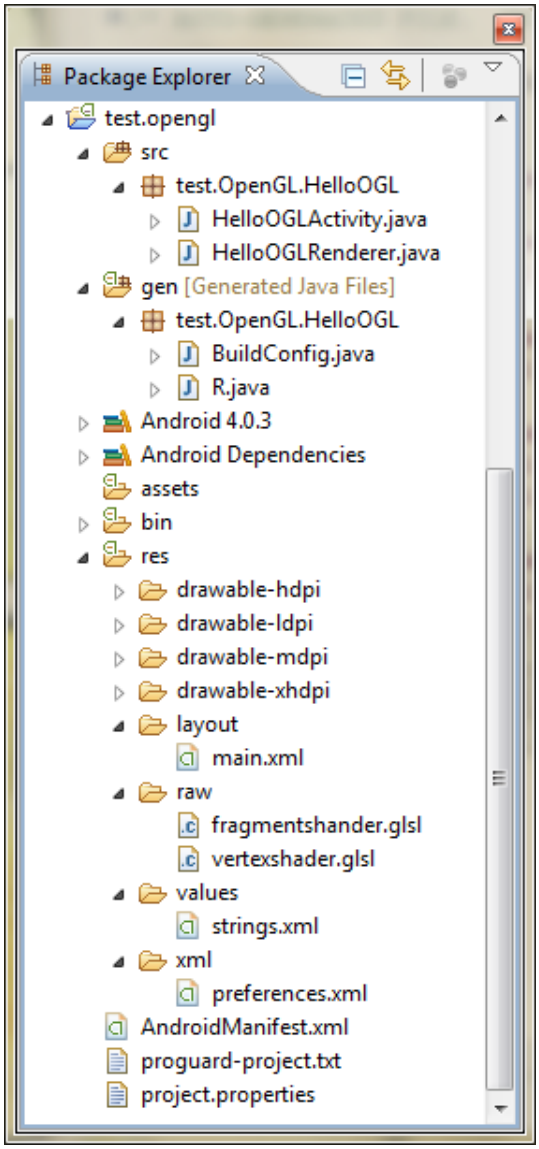


Abbildung 3.2: Struktur eines Android-Projekts

string.xml-Datei enthält die Texte, die die Applikation benutzt. Per Schüsselwort kann auf alle Texte zugegriffen werden, jedoch ist es auch möglich, Text direkt in einer Layout-Datei zu definieren, wenn er nicht mehrmals verwendet werden muss.

Alle benutzerdefinierten XML-Dateien sollen im optionalen **xml**-Ordner gespeichert werden. Der Großteil der XML-Dateien definiert das Layout von Activities oder Views.

Ein weiterer optionaler Ordner ist **raw**, in dem Shader-Dateien abgelegt werden. Dieser ist wichtig für die Verwendung von OpenGL ES.

Dalvik Virtual Machine

In der Android-Entwicklung begegnet man häufig der **Dalvik Virtual Machine**, einer virtuellen Maschine, die speziell für Android konzipiert wurde: sie berücksichtigt die spezifischen Einschränkungen von mobilen Geräten, wie z.B. die Lebensdauer von Batterien.

Dalvik benutzt ein eigenes Binärcode-Format, das sich von Java-Binärcode unterscheidet. Daher kann keine Java-Klasse direkt auf einem Android-System verwendet werden. Stattdessen muss Java-Code in der Dalvik-VM kompiliert und als Dalvik-Binärdatei in einer APK-Datei gespeichert werden.

API Level

In der Android-Entwicklung kommt häufig die Bezeichnung **API Level** (API: application programming interface) vor. Was ist ein API Level?

Auf der Android-Developer-Webseite[1] wird es folgendermaßen definiert:

„API Level is an integer value that uniquely identifies the framework API revision offered by a version of the Android platform.“ (Das API-Level ist ein Integerwert, der die Revision der Framework-API eindeutig beschreibt, welche von einer Version der Android-Plattform angeboten wird.)

Jede Aktualisierung des API Levels soll kompatibel zu früheren API Levels sein. Manche Funktionen werden jedoch nach einem API Level Upgrade auf *deprecated* gesetzt – wie man zum Beispiel im kommenden Abschnitt über Preference sehen kann. Derartige Funktionen wurden bislang *noch* nicht entfernt und sind damit weiterhin verwendbar. In seltenen Fällen können Funktionen aus Sicherheitsgründen oder wegen Bedenken zu Robustheit gestrichen werden.

Applikationsentwickler müssen das API Level beachten, weil es festlegt, welche Geräte mit welcher Plattform die Applikation ausführen können. In dieser Arbeit wurde das API Level auf 15 gesetzt, da die Applikation auf den Einsatz eines Tablets abzielt. Der Großteil aktueller Tablets wird mittlerweile vom Hersteller mit Android-Plattform 4.0.x oder höher versorgt (auf dem verwendeten ASUS Transformer Prime T201, läuft aktuell Android Version 4.0.3). Vor diesem Hintergrund ist die Festlegung auf API Level 15 gerechtfertigt.

Tabelle 3.1 zeigt den Zusammenhang zwischen API Level und Android-Versionen. Das API Level der MegaMol-Applikation ist 15.

Version	API Level	Codename
1.0	1	
1.1	2	
1.5	3	Cupcake
1.6	4	Donut
2.0	5	Eclair
2.0.1	6	
2.1.x	7	
2.2.x	8	Froyo
2.3, 2.3.1, 2.3.2	9	Gingerbread
2.3.3, 2.3.4	10	
3.0.x	11	Honeycomb
3.1.x	12	
3.2	13	
4.0, 4.0.1, 4.0.2	14	Ice Cream Sandwich
4.0.3, 4.0.4	15	
4.1, 4.1.1	16	Jelly Bean

Tabelle 3.1: API Level bezüglich der Plattform-Versionen

[1]Quelle: http://developer.android.com/guide/topics/manifest/uses-sdk-element.html#ApiLevels

Zu Beginn der Erstellung eines App-Projekts erfragt das SDK das API Level, es kann jedoch auch später in Manifest.xml neu bestimmt werden.

3.4 Applikationskomponenten

Als Android-Applikationsentwickler ist es sinnvoll, alle Komponenten einer Applikation zu kennen. Die offizielle Definition auf der Developer-Webseite[1] beschreibt diese als wesentliche Bausteine der Applikation. Jede Komponente bildet für das System einen Eingriffspunkt in die Applikation, jedoch kann der Benutzer nicht auf alle Komponenten zugreifen.

Eine Applikation enthält eine oder mehrere Komponenten. Manche Komponenten können Abhängigkeiten zu anderen Komponenten besitzen, allerdings existiert jede Komponent einer Applikation als eine eigene Einheit und spielt deshalb auch eine eigene Rolle. Die Gesamtmenge aller Komponenten bildet die vollständige Funktionalität einer Applikation.

Frühere Versionen zählten insgesamt fünf Komponenten, wie in den Bücher [Gar11] und [Ble12] beschrieben, jedoch wurde dies vor Kurzem durch eine Aktualisierung auf lediglich vier reduziert, wie die Developer-Webseite[1] erwähnt. Hierbei wurde die Komponente Intents gestrichen. Die übrigen, weiterhin bestehenden lauten Activites, Services, Content Providers und Broadcast Receivers.

Die Beschreibung und Vorstellung der einzelnen Komponenten in den kommenden Abschnitten sind hauptsächlich angelehnt an die Developer-Webseite.

Activities

Eine *Activity* lässt sich als Klasse mit einer Benutzeroberfläche verstehen, die nur aus einem einzelnen Bildschirm (*single screen*) besteht. Activities sind der einzige Teil, der letztendlich auf dem Android-Gerät sichtbar ist. Eine Applikation kann mehrere Activities umfassen, aber zu jedem Zeitpunkt ist nur eine Activity auf dem Bildschirm zu sehen.

Unter allen Activities gibt es eine „Haupt-Activity“, welche normalerweise beim Start der Applikation angezeigt wird. Jedesmal wenn eine neue Activity gestartet wird, endet die vorherige und wird in einem Stack festgehalten. Sobald die aktuelle Activity ihre Aufgabe beendet und der Benutzer den „Zurück“-Befehl freigibt, wird die vorherige Activity wieder aus dem Stack gelesen und setzt ihre Arbeit fort.

Jede *Activity* hat zudem eine eigene Lebensdauer, siehe Abbildung 3.3. Die gesamte Lebensdauer wird umschlossen von `onCreate()` und `onDestroy()`, während der sichtbare Abschnitt zwischen `onStart()` und `onStop()` liegt. Die Zeit zwischen `onResume()` und `onPause()` wird auch Vordergrund-Lebensdauer bezeichnet, da sie den Fokus des Benutzers hat. Damit

[1] `http://developer.android.com/guide/components/fundamentals.html` letzte Änderung am 10.08.2012

die Activity problemlos zwischen verschiedenen Lebensdauern wechseln kann, müssen Callback-Methoden implementiert werden.

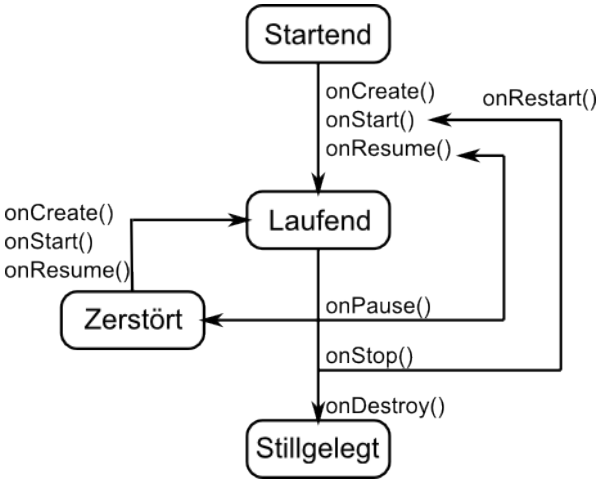


Abbildung 3.3: Activity-Lebenszyklus, vereinfacht nach Figure 1. auf http://developer.android.com/guide/components/activities.html

Für Entwickler ist `onCreate()` eine der wichtigsten Methoden einer Activity, denn sie muss zwingend implementiert werden, weil hier alle grundlegenden Initialisierungen stattfinden. Außerdem muss die Methode `setContentView()` aufgerufen werden, um das Layout der Benutzeroberfläche zu definieren.

Jede neu zum Projekt hinzufügte Activity erfordert eine eigene Anmeldung in der Manifest-Datei. Der XML-Code sieht dabei wie folgt aus:

```
<activity android:name=".Preferences"
          android:label="@string/preferences_name">
</activity>
```

Services

Services können dieselben Aktionen ausführen wie Activities, nur laufen diese ausschließlich im Hintergrund und haben daher keine Benutzeroberfläche. Services eignen sich für solche Aufgaben, die eine lange Laufzeit erfordern, oder auch für die Verarbeitung von Remote-Prozessen. Häufig auftretende Beispiele sind das Herunterladen einer Datei aus dem Internet oder das Abspielen von Musik im Hintergrund. Eine Service kann von einer anderen Komponente, wie z.B. einer Activity, gestartet und gesteuert werden.

Obwohl Services nicht im Vordergrund laufen, sind sie in einer Applikation nicht immer in separate Threads ausgegliedert.

Eine Aktualisierung der Manifest-Datei mit dem Stichwort „`service`“ ist für jeden neuen Service erforderlich.

Content Providers

Wenn Daten auf einem Android-Gerät abgelegt oder gelesen werden sollen, benötigt man einen sogenannten *Content Provider*. Dieser ist die Schnittstelle für den Datenaustausch zwischen Applikationen oder zwischen Applikation und Datenquelle. Content Provider eignen sich für umfangreiche Datenmengen.

Üblicherweise besitzt jede Applikation eine eigene, interne Datenquelle, jedoch kann der Content Provider einer Applikation auch den Zugriff anderer Applikationen gestatten, um Daten abzufragen oder zu modifizieren. Daten können im Allgemeinen überall abgespeichert werden: im lokalen Speicher, in einer SQLite-Datenbank, im Internet oder auf jeglichen Speicherplatz, auf den zugegriffen werden darf.

Um die Zugangsberechtigungen anderer Applikationen und damit die Sicherheit zu kontrollieren, bieten Content Providers eigene Methode an. Damit können auch alle externen Zugriffe verweigert werden und Daten einer Applikation nur intern les- und schreibbar gesetzt werden.

Broadcast Receivers

Broadcast Receivers empfangen Nachrichten (*broadcast*) innerhalb des ganzen Systems. Sobald ein Ereignis eintritt, wird der Empfänger aktiviert und kann daraufhin bestimmten Code ausführen.

Viele Nachrichten werden im laufenden Betrieb vom System selbst erzeugt, z.B. Meldungen über einen niedrigen Akkustand oder das Ausschalten des Bildschirms, andere werden von Applikationen gesendet. Broadcast Receivers besitzen keine Benutzeroberfläche, sondern erzeugen stattdessen Benachrichtigungen, die in der Statusleiste sichtbar werden.

Oft empfangen Broadcast Receivers Nachrichten, wie z.B. eine Anfrage, und leiten diese an diejenige Komponente weiter, welche die Aufgabe bearbeiten soll.

Intents

Obwohl sie durch eine Aktualisierung aus der Liste der Komponenten entfernt wurden, spielen *Intents* eine wichtige Rolle in einer Applikation: Sie sind Systemnachrichten, welche zwischen den Grundbausteinen ausgetauscht werden.

Drei von vier aller Komponenten – Activities, Services und Broadcast Receivers – können durch Intents aktiviert werden. Für jede Komponente gibt es einen eigenen Intent-Typen. Intents senden Nachrichten sowohl zwischen Komponenten innerhalb einer Applikation als auch zwischen Komponenten verschiedener Applikationen.

3.5 Datenspeicherung

Auch Datenspeicherung ist ein wichtiger Pfeiler in der Applikationsentwicklung. Android bietet hierbei folgende Optionen:

- **Shared Preferences**: Dies ist ein Objekt für gemeinsame Einstellungen innerhalb einer Applikation. Alle Komponenten einer Applikation können daher darauf zugreifen und die Werte werden im Format „Schlüssel–Wert“ gespeichert.
- **Interne Speicherung**: Im internen Speicher eines Gerätes werden Daten als privat bezeichnet und sind nur für die entsprechende Applikation zugreifbar. Weder andere Applikation noch der Benutzer hat direkten Zugriff.
- **Externe Speicherung**: Alle Android-Geräte bieten eine sogenannte „externe Speicherung“ für öffentliche Daten an. Die externe Speicherung kann sowohl ein Wechseldatenträger sein, z.B. eine SD-Karte, als auch eine interne Festplatte.
- **SQLite-Datenbank**: Android unterstützt SQLite-Datenbanken, die allerdings nur von derjenigen Applikation gelesen und geschrieben werden können, von der sie erzeugt wurden.
- **Netzwerkverbindung**: Daten werden durch Netzwerkoperationen auf einem eigenen Web-Server gespeichert/von einem eigenen Web-Server gelesen.

In der vorliegenden Arbeit werden die erste und die letzte Kategorie der Datenspeicherung bzw. der Datenanfrage verwendet. Der explizite Einsatz wird in Kapitel 5 und Kapitel 6 erläutert.

4 Grundlagen von OpenGL ES

In diesem Kapitel wird zunächst der Blick auf die Geschichte von OpenGL ES und seinen Design-Kriterien gerichtet; im Anschluss findet ein Vergleich zwischen den Versionen von OpenGL ES (1.x, 2.0 und 3.0) statt.

Da für die Arbeit die Verwendung von OpenGL ES 2.0 festgelegt ist, werden einige nützliche Anwendungen in der Programmierung mit OpenGL ES 2.0 und deren Shading Language vorgestellt. Der letzte Teil dieses Kapitels behandelt außerdem das Android-Framework für OpenGL ES.

4.1 OpenGL ES

4.1.1 OpenGL ES als Abstammung von OpenGL

In der Welt der Desktop-PCs gibt es zwei Standard-3D-APIs: DirectX und OpenGL (Open Graphics Library) [MGS08]. DirectX wird ausschließlich unter Microsoft Windows unterstützt und hauptsächlich in Entwicklung von 3D-Spielen unter diesem Betriebssystem verwendet. Im Gegensatz dazu ist OpenGL unabhängig vom Betriebssystem. Es steht sowohl unter Linux, Mac OS X als auch Microsoft Windows zu Verfügung. Die Anwendungsbereiche beschränken sich hierbei nicht nur auf PC-Spiele, sondern umfassen auch Teile der Benutzeroberfläche wie in Mac OS X und von zahlreichen graphischen Programmen wie CATIA (eine CAD-Applikation) und Maya (Software zu 3D-Modellierung, Animation und Rendering).

Seit der Veröffentlichung im Jahre 1992 wurde OpenGL die am häufigsten benutzte grafische API. Die Khronos Group übernahm ab 2006 die Entwicklung von OpenGL vom unsprünglichen Entwickler Silicon Graphics. Die im Jahre 2000 gegründete Khronos Group ist ein Non-Profit-Industriekonsortium, das sich auf die Erstellung von offenen Standards für Parallelrechner, graphische und dynamische Medien, welche für eine Vielzahl von Plattformen und Geräten geeignet sein sollen, konzentriert[1]. Die Khronos Gruppe besteht aus vielen Arbeitsgruppen und die OpenGL-ES-AG war die erste Arbeitsgruppe, die an einer OpenGL-Erweiterung für Mobilmedia-Standards arbeitete [PAM+08].

OpenGL ES (OpenGL for Embedded Systems) ist, wie der Namen erahnen lässt, eine vereinfachte Version von OpenGL, welche sich speziell an eingebettete Systeme richtet, wie zum Beispiel Mobiltelefone, PDAs oder Tablets.

[1] http://www.khronos.org/about

4.1.2 Design-Kriterien

Da Mobilgeräte im Gegensatz zu Desktopsystemen deutlich eingeschränkt sind in Prozessorleistung und Größe des Arbeitsspeichers, oder auch keine floation-point-Befehlseinheiten umfassen und hinsichtlich der Stromversorgung Beschränkungen unterliegen, wurde der Umfang von OpenGL ES ausgehend von OpenGL nach gewissen Kriterien reduziert bzw. bei Bedarf ergänzt. Dies umfasst folgende Punkte[MGS08][PAM+08]:

- Entfernung von Redundanz. Da die OpenGL-API sehr umfangreich und komplex ist, entfernt OpenGL ES alle redundanten Funktionen. Redundanz entsteht in OpenGL durch das Erscheinen von neuer Hardware oder Implementation von neuen Methoden, welche alte ersetzen. Beispielsweise das Rendering von Dreiecken in OpenGL kann mit sechs verschiedenen Methoden ausgeführt werden. Gibt es mehr als eine Realisierungsmethode einer Aufgabe, soll nur die nützlichste Methode behalten und auf Duplikationen verzichtet werden. Dadurch wurde OpenGL ES 1.0 beispielsweise in weniger als 50KB Binärcode implementiert.
- Beibehaltung der Kompatibilität mit OpenGL. Auch wenn die Kompaktheit eine Hauptanforderungen an OpenGL ES ist, sollte die Kompatibilität nicht vernachlässigt werden. OpenGL ES wurde so konzipiert, dass die Untermenge der embedded-Funktionalität sowohl in OpenGL als auch in OpenGL ES lauffähig ist.
- Neue Eigenschaften der Mobilgeräte abdecken. ([MGS08]S.92) Alle double-precision floating-point-Datentypen werden durch single-precision ersetzt, um die Shader besser auszunutzen.
- Benötigte Bildqualität auf kleinen Bildschirm berücksichtigen. Die meisten Mobilgeräte haben kleinere Bildschirmgrößen und dadurch weniger Pixel als herkömmliche Computermonitore. Dies beeinflusst die Konzeption.
- Bestehen von Konformitätstests. Die Implementation von OpenGL ES soll bestimmte Standards von Qualität, Genauigkeit und Robustheit erfüllen. Für diesen Fall gibt es spezielle Tests.

4.2 OpenGL-ES-Versionen

Die Khronos Group erstellte bislang drei Versionen von OpenGL ES: 1.0, 1.1 und die in der Arbeit verwendete 2.0. OpenGL ES 3.0 wurde seit geraumer Zeit geplant, deren Spezifikationen wurden allerdings erst August 2012 auf der SIGGRAPH-Konferenz vorgestellt.

Jede Version ist mit einer bestimmten OpenGL Version zu vergleichen. Die entsprechende OpenGL- und OpenGL-ES-Version ist gelistet in Tabelle 4.1:

OpenGL ES	Entsprechung in OpenGL	Bemerkungen
1.0	1.3	2003
1.1	1.5	2004
2.0	2.2/3.0	2007 API Level 8
3.0 (Halti)	3.2+/4.2	August 2012

Tabelle 4.1: OpenGL ES und OpenGL

OpenGL ES Version 1.1 ist abwärtskompatibel zu 1.0, die 1.x Versionen haben beide nur Fixed-Function-Pipelines; OpenGL ES 2.0 hat eine programmable Pipeline und unterstützt OpenGL ES Shading Language, aber es hat keine Fixed-Function-Pipeline. Das heißt Version 2.0 ist nicht mehr kompatibel zu Version 1.x. Laut [MGS08] werden hierfür vor allem drei Gründe angeführt:

- Die alte Shading-Language widerspricht dem Design-Kritierium zur „Entfernung von Redundanz“. Falls OpenGL ES 2.0 sowohl eine Fixed-Function-Pipeline als auch eine programmierbare Pipeline unterstützte, gäbe es bei einer Vielzahl von Szenarien mindestens zwei Lösungsmöglichkeiten. Dies soll unbedingt vermieden werden.
- Studien zufolge werden programmierbare und Fixed-Function-Pipelines selten gemischt benutzt. Wenn man die Freiheit einer programmierbaren Pipeline besitzt, möchte man üblicherweise aus Gründen der Flexibilitaet keine Fixed-Function-Pipeline mehr verwenden.
- Weitaus mehr Arbeitsspeicher müsste verwendet werden, wenn der Treiber von OpenGL ES 2.0 beide Pipelines unterstützen müsste. Minimale Anforderung an den Arbeitsspeicher ist einer der Kernpunkte für die Zielgruppe von OpenGL ES. Durch die Trennung der beiden Pipelines muss der Treiber von OpenGL ES 1.x nicht mehr zwingend geladen werden, wenn dieser ohnehin nicht benötigt wird.

Nach Untersuchungen von Google unterstützen 90,3% aller Android-Geräte beide Versionen[2]. Unterstützt eine Applikation ausschließlich OpenGL ES 2.0, muss die nachfolgende Deklaration in der Manifest-Datei hinzugefügt werden. Diese verhindert die Installation der Applikation auf Geräten, welche OpenGL ES 2.0 nicht unterstützen.

```
<!-- Tell the system this application requires OpenGL ES 2.0. -->
 <uses-feature android:glEsVersion="0x00020000" android:required="true"/>
```

Es liegt auf der Hand, dass Version 2.0 den Schwierigkeitsgrad und Aufwand der Programmierung erhöht, es bietet aber dahingehend auch deutlich mehr Flexibilität. Fixed-Function

[2]Quelle: http://developer.android.com/about/dashboards/index.html#OpenGL

ist eine Teilmenge der programmierbaren Pipeline. Alle Funktionen in der Fixed-Function-Pipeline können auch mit einer programmierbaren Pipeline realisiert werden. Aus diesen Gründen wird OpenGL ES 2.0 für die Diplomarbeit eingesetzt.

Die vor Kurzem angekündigte Spezifikation von OpenGL ES 3.0 nahm zahlreiche Optimierung vor im Vergleich zu Version 2.0, wie z.B. Unterstützung eines neuen, plattformübergreifenden Texturformats, Occlusion Queries, Transform Feedback, Instanced Rendering und Unterstützung für vier und mehr Renderziele und Unterstützung für Gleitkommatexturen, Tiefentexturen und Vertex-Texturen. Auch die OpenGL ES Shading Language 3.0 wurde überarbeitet und unterstützt dadurch 32-Bit-Integer- sowie 32-Bit-Gleitkommaoperationen. Trotz vieler Verbesserungen bleibt OpenGL ES 3.0 abwärtskompatibel zu Version 2.0. Mehr über OpenGL ES 3.0 ist in der Spezifikation zu finden [Khr12].

4.3 Programmierung mit OpenGL ES 2.0

OpenGL ES 2.0 besteht aus zwei Bestandteilen: die Spezifikation der OpenGL ES 2.0 API (inklusive Kernspezifikation und Extensions) und der OpenGL ES Shading Language (OpenGL ES SL). Abbildung 4.1 zeigt die graphische Pipeline von OpenGL ES 2.0.

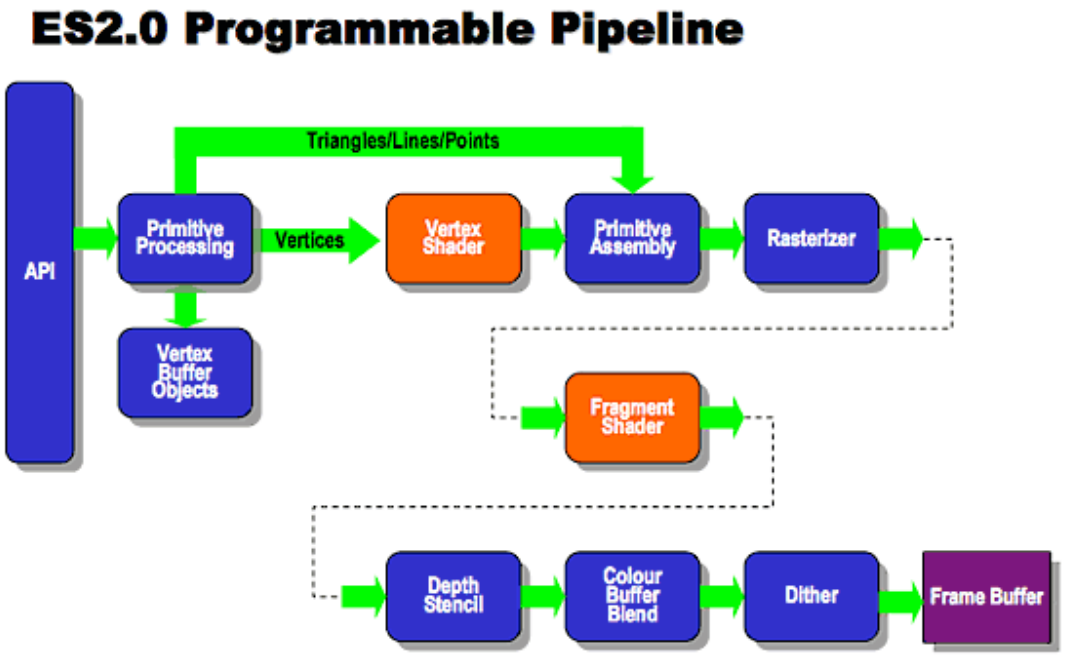
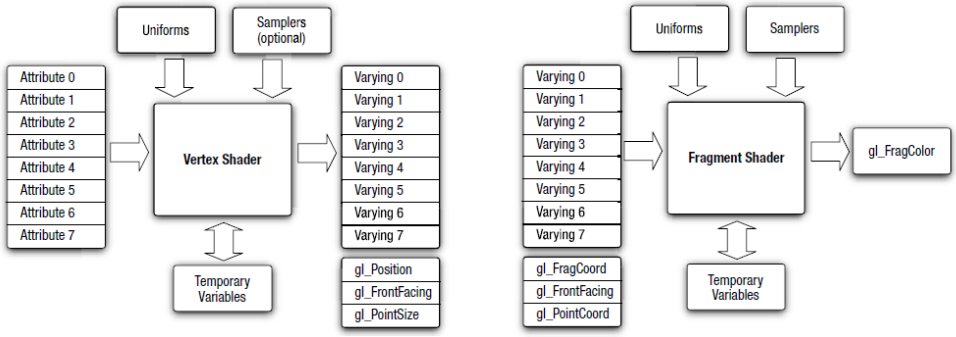


Abbildung 4.1: Graphische Pipeline von OpenGL ES 2.0 (Quelle: Khronos http://www.khronos.org/opengles/2_X/)

Die beide orangenen Blöcke in Abbildung 4.1 sind Vertex- und Fragment-Shader und bilden den programmierbaren Teil in der Pipeline. Beide Shader-Programme ersetzen den Fixed-Function-Teil aus OpenGL ES 1.x.

Shader-Programme und Initialisierung

Ähnlich zur OpenGL Shading Language (OpenGL SL) gibt es in OpenGL ES SL auch zwei Shader-Typen: Vertex- und Fragment-Shader. Beide arbeiten immer paarweise zusammen. Der Vertex-Shader bekommt Per-Vertex-Daten als Eingabe und führt Vertex-basierte Aufgaben wie z.B. Positionstransformation oder Generierung der Texturkoordinaten durch. Die Ausgabe eines Vertex-Shaders wird in einem weiteren Schritt gerastert und an den Fragment-Shader als Eingabe weitergeleitet. Dort wird für jedes Fragment ein Farbwert (`gl_FragColor`) berechnet, der anschließend auf den Bildschirm gezeichnet wird [MGS08].



(a) Vertex-Shader **(b)** Fragment-Shader

Abbildung 4.2: OpenGL ES 2.0 Vertex- und Fragment-Shader (Quelle: [MGS08])

Es gibt einige Code-Beispiele, in denen Shader-Code direkt in Java-Code geschrieben wird, weil Entwickler dadurch den Input/Output mit Android-Dateien umgehen können. Dies ist jedoch nur sinnvoll, wenn der Shader-Code sehr kurz ist. Üblicherweise soll Shader-Code in einer externen Datei (z.B. mit dem Datei-Suffix .glsl) gespeichert und über eine API geladen werden. Für die Einbindung in ein Programm gibt es in OpenGL ES ähnliche Befehle wie in der OpenGL Shader Language.

Zunächst muss für jedes Shader-Programm ein **Shader-Objekt** definiert werden, in dem der Quellcode geladen und kompiliert wird. Ein Beispiel für Laden und Kompilierung des Shader-Codes ist in Kapitel 5.1.3 zu sehen. Die Interaktionen zwischen Hauptprogramm und Shader-Programmen verwaltet ein **Program-Objekt** [MF12]. Daher umfasst ein Program-Objekt exakt zwei Shader-Objekte, in die jeweils die zusammengehörigen Vertex- und Fragment-Shader-Code kompiliert werden [MGS08].

Listing 4.1 zeigt den Code für die Erstellung und Verbindung eines Program-Objektes mit zwei bereits erfolgreich kompilierten Shader-Objekten. Die Fuktion `glLinkProgram()` überprüft, ob die Definition der Uniform-Variablen in beiden Shader-Programmen übereinstimmt,

Listing 4.1 Erstellung und Verbindung eines Program-Objektes

```
int program_object = GLES20.glCreateProgram();
GLES20.glAttachShader(program_object, vertex_shader_object);
GLES20.glAttachShader(program_object, fragment_shader_object);
GLES20.glLinkProgram(program_object);
```

ob alle Varying-Eingaben in Fragment-Shader im Vertex-Shader definiert sind usw.. Als Ergebnis erhält man ein ausführbares Shader-Programm. Danach wird `glGetAttribLocation()` verwendet, um alle Attribute-Variablen in den Shader-Programmen einzeln mit dem Program-Objekt zu verbinden. Analog gibt es die Methode `glGetUniformLocation()` für Uniform-Variablen. Jede Variable bekommt dadurch ein *Handle* mit einem eindeutigen Verbindungsindex.

Vertexbuffer-Objekte

Den größten Anteil an Eingaben eines Vertex-Shaders bilden Vertex-Daten. Für das Rendering jedes Frames müssen diese Daten in den Vertex-Shader geladen werden. Daher ist es äußerst ungünstig, wenn das Hauptprogramm dabei jedes Mal die Daten definieren und an Shader-Programme senden muss, insbesonders für große Datenmengen, wie die eines Protein-Moleküls, das meistens tausende oder noch mehr Atome umfasst.

Als Abhilfe für dieses Problem erbt OpenGL ES den Begriff Vertexbuffer-Objekt aus OpenGL, um die Vertex-Daten in einem Vertexarray-Format im graphischen Speicher zu cachen. Dies erlaubt einen direkten Datenzugriff, so lange das Objekt gerendert wird [MF12]. Aufgrund von generellen Einschränkungen der Tablet-Hardware ist die Verwendung der Vertexbuffer-Objekte in OpenGL ES deshalb sehr zu empfehlen.

Ein Beispiel wie ein Vertexarray mit dem Vertexbuffer-Objekt verbunden wird, ist in Listing 5.3 zu sehen. Mit Hilfe der Methode `glVertexAttribPointer()` wird das Vertexbuffer-Objekt mit dem entsprechenden Handle verbunden und `glEnableVertexAttribArray()` aktiviert damit das benötigte Vertexarray.

Framebuffer-Objekte

Das Framebuffer-Objekt ist ein wichtiger Bestandteil beim Einsatz von OpenGL ES. Ein Framebuffer ist wie eine temporäre Speicherung direkt vor dem Rendering. In Abbildung 4.1 wird er als letzter Block in der Pipeline aufgeführt. Somit werden im Framebuffer die unmittelbar auf einem 2D-Bildschirm renderbaren Daten gespeichert. Mit einem Framebuffer-Objekt kann nun das Rendering zwischen Off-Screen und On-Screen umgeschaltet werden. Das On-Screen-Rendering zeichnet die im Framebuffer gespeicherten Daten auf den Bildschirm, während das Off-Screen-Rendering nichts zeichnet. Die mit dem Framebuffer-Objekt verbundenen Daten können als Textur weiterverwendet werden.

Die Methode `glBindFramebuffer()` ermöglicht das Ein- und Ausschalten des Framebuffers. Wenn es ein Framebuffer-Objekt als Eingabe-Parameter gibt, werden die zu rendernden Daten mit ihm verbunden. Ist der Eingabe-Parameter „null“, werden die Daten mit keinem Framebuffer-Objekt verbunden, sondern auf den Bildschirm gerendert.

4.4 Android-Framework für OpenGL ES

Im Android-Framework gibt es zwei fundamentale Klassen, welche die Erstellung und Manipulierung der Graphik per OpenGL-ES-API zulassen: »GLSurfaceView« und »GLSurfaceView.Renderer« [dev]. Die »GLSurfaceView«-Klasse leitet die »SurfaceView«-Klasse ab und bietet eine Fläche für OpenGL-ES-Rendering an.

»GLSurfaceView.Renderer« ist das Interface des Renderings und für OpenGL-ES-Rendering-Aufgaben zuständig. Diese Klasse muss separat implementiert werden und arbeitet als ein eigener Thread, sodass das Rendering unabhängig von der Benutzeroberfläche ablaufen kann. Mit `GLSurfaceView.setRenderer()` meldet es sich beim GLSurfaceView-Objekt an.

Bei der Nutzung der »GLSurfaceView.Renderer«-Klasse werden die folgende Methoden notwendigerweise implementiert:

- **onSurfaceCreated()**: Diese Methode wird nur ein Mal am Anfang des Renderings aufgerufen und eignet sich für solche Aufgaben, die nur einmal ausgeführt werden müssen, beispielsweise die Einstellung der Umgebungsparameter oder Initialisierung der graphischen Objekte.
- **onDrawFrame()**: Jedes Mal, wenn das GLSurfaceView-Objekt neu gezeichnet werden muss, wird diese Methode verwendet, um die graphischen Objekte darzustellen.
- **onSurfaceChanged()**: Diese Methode wird benutzt, wenn die Größe des GLSurfaceView-Objekt geändert wird, z.B bei Orientierungsänderungen des Gerätes.

Alle drei Methoden erwarten die gewünschte OpenGL-ES-Version als Parameter; das heißt, wird OpenGL ES 2.0 verwendet, muss `(GL10 unused)` als Parameter übergeben werden.

5 Architektur der gesamten Arbeit

In diesem Kapitel wird die gesamte praktische Ausarbeitung ausführlich vorgestellt. Dies beinhaltet sowohl die Applikation für MegaMol (Abbildung 5.1 rechts), als auch die neuen Klassen in MegaMol, die als Schnittstelle zwischen Applikation und MegaMol Verwendung finden (Abbildung 5.1 links).

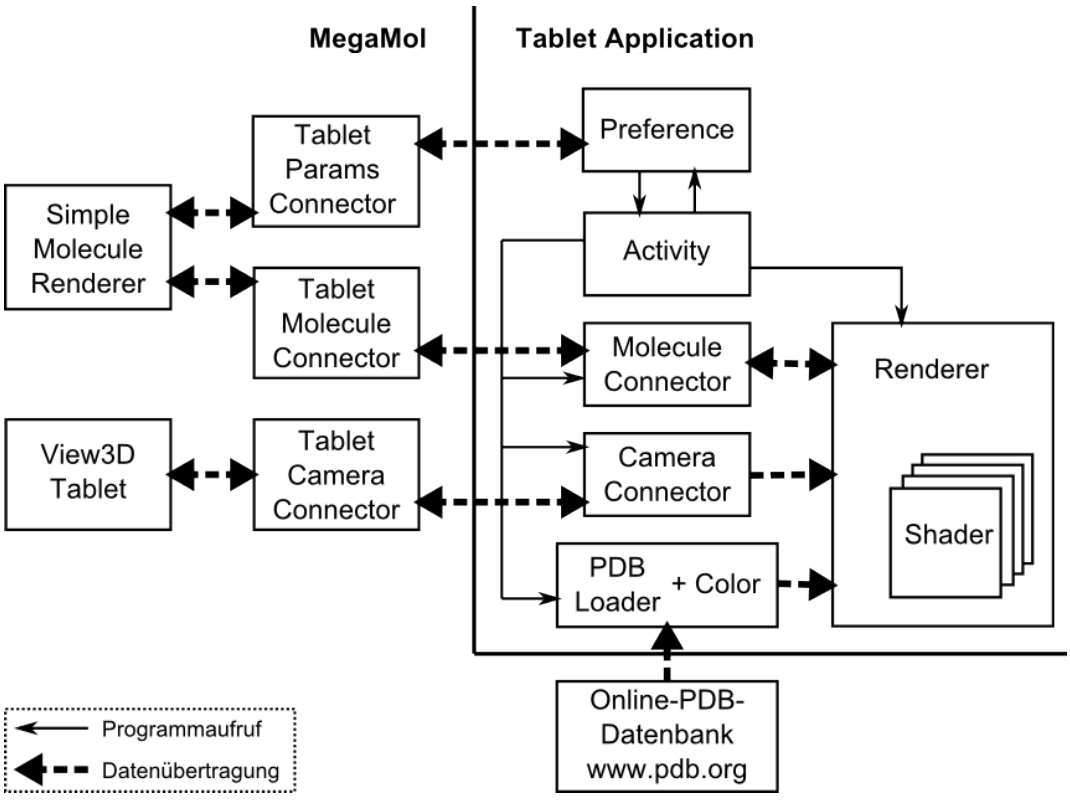


Abbildung 5.1: Die verwendete Architektur

5.1 Struktur der Applikation

Die Applikation besitzt zwei Funktionen: die Remote-Kontrolle des MegaMol-Programms, welche hauptsächlich durch »Activity« und »Preference« realisiert wird, und die Visualisierung des Moleküls, gezeichnet durch »Renderer« und »Shader«. Dazwischen ermöglichen

zwei »Connector«-Module die Datenübertragung zwischen MegaMol auf dem Server und der Applikaton auf dem Tablet. Als Alternative lädt »PDBLoader« die »PDB-Dateien« direkt von der Online-Protein-Datenbank [BWF+00] herunter und damit kann die Applikation mit Hilfe der Farb- und Radiustabelle in »Color« auch ohne Verbindung mit MegaMol Moleküle visualiseren.

5.1.1 Activity

Die Klasse »Activity« ist die Hauptbenutzeroberfläche der Applikation und stellt den Rahmen bereit, in den gerendert wird. Hier werden die GLSurfaceView und dessen Renderer erstellt. Alle in »Renderer« aufgelisteten graphischen Aufgaben werden in diesem View gezeichnet. Der Rendermodus wird auf `RENDERMODE_CONTINUOUSLY` gesetzt, weil die Visualisierung von animierten Daten einen ständig wiederholenden Renderingprozess benötigt.

»Activity« behandelt außerdem alle Benutzerinteraktionen, inkl. TouchEvent und Optionsmenü. Diese erfüllen zusammen die Anforderungen an eine Remote-Steuerung.

Multitouch und Kamerasteuerung

Die Bedienung durch einen Touchscreen, wie z.B. Drehung und Zoom, wird durch Objekte von Typ „MotionEvent“ verwaltet. MotionEvent beschreibt eine Bewegung mit zwei Faktoren, bestehend aus Bewegungscode und Koordinatenwerten. Ein Koordinatenwert notiert die Position einer Fingerbewegung und der Bewegungscode überwach, ob es eine Änderung im Zustand des Touchscreens gibt. Letzterer beantwortet auch, wieviele Pointer es momentan gibt oder ob ein Bereich gedrückt oder losgelassen wurde.

Ein Multitouchscreen verfolgt die Bewegungen jedes Fingers einzel mittels „Pointers“. Ein Pointer wird erstellt und bekommt eine eindeutige ID, sobald der Bildschirm zum ersten Mal an einem Punkt gedrückt wird (indiziert durch `ACTION_DOWN` oder `ACTION_POINTER_DOWN`). Der Pointer folgt dann der Bewegung und bleibt solang aktiv, bis der Finger die Oberfläche verlässt (indiziert durch `ACTION_UP` oder `ACTION_POINTER_UP`). Die Methoden `getX(int)` und `getY(int)` in MotionEvent fragen die Position eines Pointers (indiziert durch Pointer Index) ab. Die Anzahl der aktiven Pointer ermittelt die Methode `getPointerCount()`.

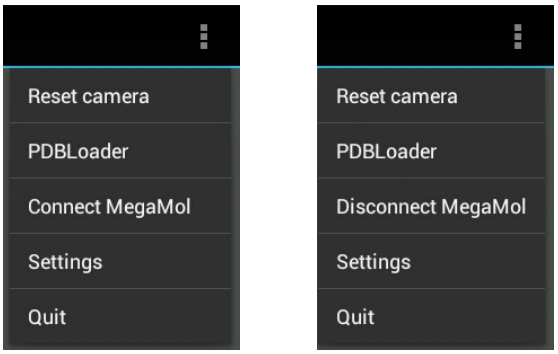
MotionEvent behandet neben dem Touchscreen auch solche „Bewegungen“, die von externen Geräten (z.B. Maus, Trackball) erzeugt werden. Die MegaMol-Applikation konzentriert sich auf die Steuerung per Touchscreen. Die Eingaben lassen sich durch Implementierung der Methode `onTouchEvent(MotionEvent event)` abarbeiten. In der Applikation entscheidet die Anzahl der Pointer die Art der Kamera-Steuerung: Gibt es nur einen Pointer, dreht sich die Kamera um Objekte; gibt es mehr als zwei Pointer, bewegt sich die Kamera nach vorne oder hinten, auch genannt „pinch to zoom“. Bei der Kameraverschiebung zählt nur die Bewegungen der ersten beiden Pointer.

Die Realisierung der Kamerasteuerung gehört zur Koodinaten-Transformation in OpenGL ES und findet daher in der »Renderer«-Klasse statt. Hier fragt »Activity« sämtliche Positionen der Pointer ab und leitet die Werte weiter an »Renderer«.

Weitere Details zur Interaktivität auf Multitouchscreens sind auf der Android-Developer-Webseite [dev] unter dem Begriff „MotionEvent“ gelistet.

Menü

Android bietet dem Applikationsentwickler eine „Menu“-API an, mit der man standardmäßig Menüs aufbauen kann. Insgesamt gibt es drei verschiedene Menütypen: *option menus*, *content menus* und *sub menus*. In der MegaMol-Applikation wird ein option menu verwendet.



(a) **(b)**

Abbildung 5.2: Menü: (a) Applikation und MegaMol sind nicht verbunden. (b) Applikation und MegaMol sind verbunden.

Abbildung 5.2 zeigt das Menü der MegaMol-Applikation. Sie bietet folgende Auswahlmöglichkeiten:

- **Reset Camera**: setzt die Kamera-Parameter auf den Initialzustand zurück.
- **PDBLoader**: lädt die gewünschte PDB-Datei aus der Online-Datenbank [BWF+00] herunter und visualiert das Molekül.
- **Connect MegaMol / Disconnect MegaMol**: verbindet bzw. trennt die Kommunikation mit dem MegaMol-Server, abhängig vom aktuellen Verbindungszustand.
- **Settings**: ruft die Einstellungs-Activity ab, siehe Kapitel 5.1.2.
- **Quit**: trennt alle exisierenden Socket-Verbindungen und beendet dann die Applikation.

Bei der Erstellung eines Menüs wird normalerweise das Layout in einer XML-Datei gespeichert. Die Methode `onCreateOptionsMenu()` ist für das Erstellen des Menüs zuständig und `MenuInflater` lädt die XML-Datei. Alternativ kann ein Menü mit einem einfachen

Layout auch direkt in `onCreateOptionsMenu()` editiert werden, so wie in der MegaMol-Applikation.

Wenn ein Element im Menü angeklickt wird, aktiviert das System die Methode `onOptionsItemSelected(MenuItem item)`, in der für jedes Element eine bestimmter Code-Baustein ausgeführt wird. Üblicherweise realisiert man dies in einer `switch... case...`-Anweisung, wobei der Schüssel die eindeutige Id-Nummer der „MenuItem“ besitzt. Diesen Wert erhält man durch Aufruf der Methode `getItemId()`.

Je nach Verbindungszustand mit dem MegaMol-Server muss ein Menüelement der MegaMol-Applikation zwischen Verbindung und Trennung umschalten, in diesem Fall wird ein dynamisches Menü benötigt. Android lässt eine Modifikation zu, indem man die Methode `onOptionsItemSelected(MenuItem item)` durch `onPrepareItemSelected(MenuItem item)` ersetzt. Diese Methode wird jedes Mal aufgerufen, wenn der Menü-Knopf betätigt wird und erzeugt somit das Menü neu.

5.1.2 Preference

»Preference« läuft als eine separate Activity, die von der Klasse „PreferenceActivity“ abgeleitet wird. Sie modifiziert die Konfigurationen und Einstellungen der Applikation und wird durch Anklicken des Menüpunktes „Settings“ (siehe Abbildung 5.2) aktiviert.

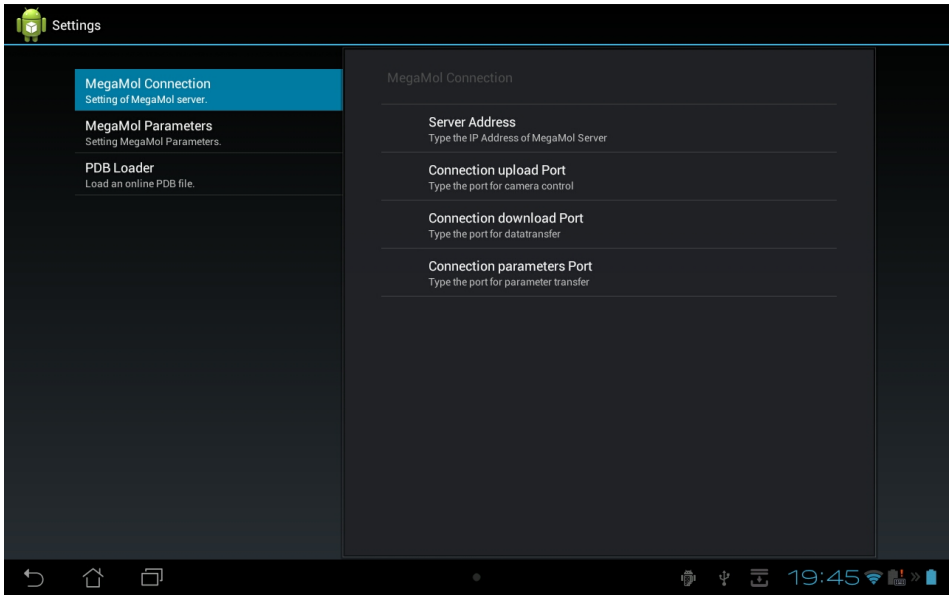


Abbildung 5.3: Preference Layout mit Header und Fragment

Grundstäzlich gibt es zwei verschiedene Layouts für eine Preference: das einspaltige und das zweispaltige. In der MegaMol-Applikation wird Letzteres verwendet, siehe Abbildung 5.3.

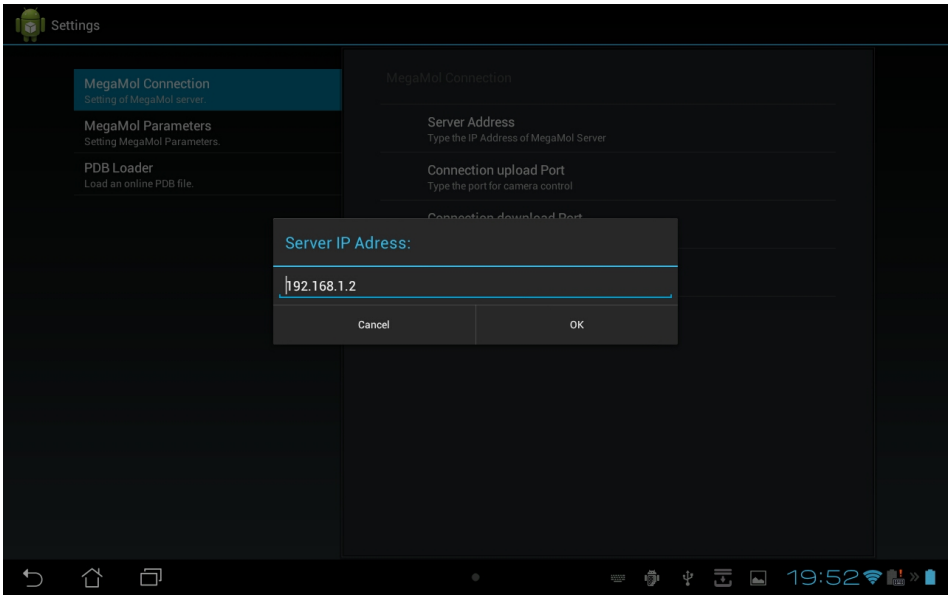


Abbildung 5.4: Einstellung des MegaMol-Servers in Preference mit einem EditTextPreference.

Die schmale Spalte links wird genutzt für *Headers* und der restliche Raum ist für *Fragments*. Ein Header ist eine Liste und jedes Element auf dieser Liste bezieht sich auf ein Fragment. Ein Fragment wiederum enthält eine Liste von Preference-Objekten, die zusammen gruppiert werden sollen. Beispielsweise enthält das in der Abbildung 5.3 gezeigte Fragment alle Einstellungen über die MegaMol-Verbindung. Ein nächstes Fragment realisiert die Remote-Einstellung der Megamol-Parameter, und das dritte Fragment enthält Einstellungen über Online-PDB-Dateien.

Das Activity-Layout wird wie gewohnt im XML-Format gespeichert. Das bedeutet: für jeden Header und jedes Fragment gibt es eine separate XML-Datei, welche das Layout definiert. Die Implementierung der Methode `onBuildHeaders()` lädt das Header-Layout während die Methode „PreferenceFragment“ für das Laden des Fragment-Layouts zuständig ist.

Soll Preference doch nur ein einspaltiges Layout besitzen, wird ausschließlich das Fragment ohne Header benutzt, falls die Applikation für Android 3.0 (API Level 11) oder höher entwickelt wird. Die Ableitung von „PreferenceActivity“ spielt damit keine Rolle mehr. Einige Methoden in „PreferenceActivity“ sind ohnehin aus heutiger Sicht nicht mehr zu empfehlen (*deprecated*).

Unter dem Preference-Objekt versteht man einstellbare Elemente. Android bietet verschiedene eingebaute Preferences an, wie z.B. *EditTextPreference* (siehe Abbildung 5.4), *CheckBoxPreference ListPreference* usw.. Für den Fall, dass diese jedoch nicht ausreichend sind, erlaubt Android dem Entwickler das Schreiben eigener Preferences und deren Verwendung in Layout-Dateien.

Jedes in der Fragment-Layout-Datei definierte Preference-Objekt muss mindestens einen eindeutigen Schlüssel *key* besitzen, da das System die Einstellungen im Schlüssel-Wert-Format in der *SharedPreferences*-Datei speichert. Außerdem legt *title* den Namen des Preference-Objektes fest. Die Definition einer Preference sieht daher folgendermaßen aus:

```
<EditTextPreference
    android:key="megamol_ip"
    android:summary="Type the IP Address of MegaMol Server"
    android:title="Server Address"/>
```

SharedPreferences ist eine Option der Android-Datenspeicherung (siehe Kapitel 3.5. Jede Applikation besitzt eine eigene SharedPreferences-Datei, auf die alle Komponenten der Applikation (Activities, Services, Broadcast Receivers und Content Providers) Zugriff haben. Dies erspart einen Teil der Kommunikation zwischen den Komponenten.

Durch Implementierung der Methoden `getSharedPreferences()` bzw. `getPreferences()` kann eine Komponente auf die Datei zugreifen. Um deren Inhalt zu lesen, bietet Android verschiedene Methoden wie `getBoolean()`, `getString()` usw. an. Ähnlich dazu gibt es beim Schreiben der Daten in SharedPreferences Methoden wie `putBoolean()`, `putString()` usw.. Darüberhinaus erfordert die Modifikation der Daten ein `SharedPreferences.Editor`-Objekt. Listing 5.1 ist ein vereinfachtes Beispiel und zeigt wie die Activity »Preference« in der MegaMol-Applikation Daten von der SharedPreferences-Datei liest und schreibt.

Listing 5.1 Daten in SharedPreferences lesen und schreiben

```
Vertex Shader Code:
	...
	public class Preferences extends PreferenceActivity {
		private SharedPreferences prefs;
		...
		@Override
		protected void onCreate(Bundle savedInstanceState) {
			super.onCreate(savedInstanceState);
			prefs = PreferenceManager.getDefaultSharedPreferences(this);
		}

		private void writeParams() {
			String temp;
			...
			SharedPreferences.Editor editor = prefs.edit();
			editor.putString("param_1", temp);
			editor.commit();
			...
		}

		private void readParams() {
			...
			String temp = prefs.getString("param_1", "0.0");
			...
		}

	}
```

Zusammen mit dem Aufbau einer Socket-Verbindung realisiert die Activity »Preference« die Remote-Editierung der Parameter: »Preference« versucht jedes Mal nach der Aktivierung eine Socket-Verbindung mit MegaMol herzustellen. Ist der Verbindungsversuch erfolgreich, fragt »Preference« die Werte von ausgewählten MegaMol-Parametern ab und schreibt diese in die SharedPreferences-Datei. Der Benutzer kann nun die Parameter editieren. Bei der darauffolgenden Deaktivierung der »Preference« sendet diese Activity die Parameter zurück zu MegaMol und schließt danach die Verbindung. Die Schnittstelle in MegaMol, welche mit »Preference« kommuniziert, ist eine neue Klasse »TabletParamsConnector«, siehe Abbildung 5.1 und Kapitel 5.2.3.

5.1.3 Renderer und Shader

Die »Renderer«-Klasse bildet zusammen mit den Shader-Dateien den wichtigsten Teil der Applikation; sämtliche Visualisierungsaufgaben werden hier ausgeführt. Renderer ist eine Implementierung von GLSurfaceView.Renderer. Aus Abschnitt 4.4 sind bereits die drei Standardmethoden in einem GLSurfaceView.Renderer bekannt: `onSurfaceCreated()`, `onDrawFrame()` und `onSurfaceChanged()`. In dieser Applikation hat `onSurfaceChanged()` keine andere Aufgabe, als die Änderung zwischen Hoch- und Querformat zu berücksichtigen. Die meisten Visualisierungsaufgaben sind aufgeteilt zwischen `onSurfaceCreated()` und `onDrawFrame()`.

onSurfaceCreated()

Hier findet die Initialiserung der Shader-Programme statt. Jedes Shader-Programm-Paar wird mit einem Program-Objekt verbunden. Das Laden des Shader-Codes durch Implementierung der I/O Methode `getResources().openRawResource(int shader_program_id)`. Der Code wird in ein Byte-Array eingelesen und gespeichert in einer String-Variable `shader_code`.

Listing 5.2 Kompilierung des Shader-Codes

```
int shader = GLES20.glCreateShader(shader_type);

GLES20.glShaderSource(shader, shader_code);
GLES20.glCompileShader(shader);

final int[] compileStatus = new int[1];
GLES20.glGetShaderiv(shader, GLES20.GL_COMPILE_STATUS, compileStatus, 0);

if (compileStatus[0] == 0) {
        String shaderErrors = GLES20.glGetShaderInfoLog(shader);
        GLES20.glDeleteShader(shader);
        shader = 0;
}
```

`shader_type` ist entweder `GLES20.GL_VERTEX_SHADER` oder `GLES20.GL_FRAGMENT_SHADER` und `shader` ist das Endergebnis. Da Shader-Programme erst zur Laufzeit kompiliert werden, bietet die Überprüfung des `compileStatus` die Möglichkeit an, im Debug-Modus durch Log-Dateien Fehler zu untersuchen.

onDrawFrame()

Zunächst wird in onDrawFrame überprüft, ob ein Neuladen der Objekte notwendig ist. Die Objektinitialisierungsmethode wird aufgerufen, sobald es Änderung der Objekt-Daten gibt. Sie wird später in diesem Abschnitt vorgestellt.

Nachdem der VertexBuffer die Objektdaten mit dem ShaderHandle verbunden hat, wickelt onDrawFrame die Transformation der Koordinatensysteme ab. Abbildung 5.5 skizziert die Reihenfolge der Koordinatentransformation.

Häufig werden die Model-View-Matrix und die Projektions-Matrix als Eingaben des Vertex-Shaders verwendet. Die Transformation von Weltkoordinaten nach Fensterkoordinaten findet in der Shader-Pipeline statt. Das Ergebnis der Clip-Koordinaten ist in der Ausgabe des Vertexshaders zu finden. Es heißt `gl_Position`. Die Perspektiv-Transformation (auch Perspektivdivision genannt) und Viewport-Transformationen werden nach dem Vertex-Shader von der Shader-Pipeline durchgeführt. Das Endergebnis aller Transformationen wird in `gl_FragCoord` festgehalten.

Um mit der Kamerasteuerung in MegaMol übereinzustimmen wird die Kamera als Objekt in Weltkoordinaten ausgedrückt. Wichtige Kamera-Parameter wie z.B. die Kamera-Position und den nach oben zeigenden Richtungsvektor, bekommt das Tablet entweder direkt von MegaMol während der Synchronisation oder sie werden unmittelbar aus der eigenen Translationsmatrix zur Koordinatenverschiebung ausgelesen.

Die Atome eines Moleküls werden mit der Größe der Bounding-Box in Weltkoordinaten skaliert und transliert. Der Mittelpunkt der Bounding-Box wird als LookAt-Punkt der Kamera an passender Stelle weitergeleitet.

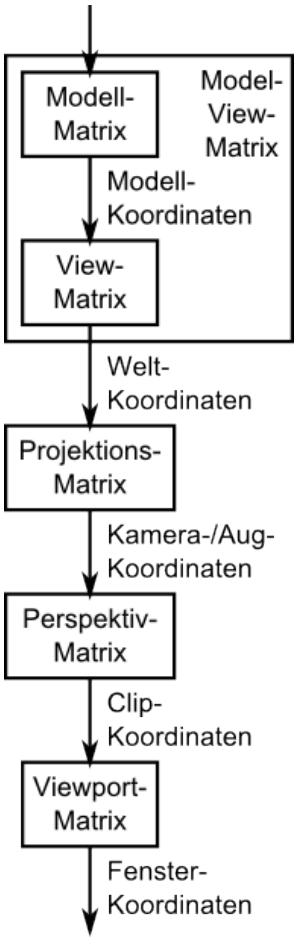


Abbildung 5.5: Koordinaten-Transformation

Matrizen wie z.B. die Model-View-Matrix, die Model-View-Projekt-Matrix usw. werden bei Bedarf als Uniform-Variablen an die Shader-Programme gesendet, um das Rendering zu unterstützen.

Das Objekt-Rendering wird in zwei Rendering-Schritten durchgeführt: Der erste Schritt führt den Tiefentest durch und der Rendering-Prozess findet in einem sogenannten Framebuffer-Objekt statt. Das Ergebnis wird jedoch nicht auf den Bildschirm gezeichnet, sondern nur innerhalb der Zwischenablage in einem Framebuffer-Objekt gespeichert. Das Rendering wiederum benutzt das Ergebnis des Tiefentests und rendert das Objekt schließlich auf den Bildschirm.

Wieso das Rendering derart aufwändig ist und was genau in den zwei Rendering-Phasen geschieht wird im Abschnitt 6.2 ausführlich diskutiert.

Sonstige Methoden

Methoden, die weitere Kontrollmöglichkeiten über die Kamera bieten:

- **resetCamera**: diese public-Methode wird direkt durch einen Menüpunkt aktiviert und setzt alle Kamera-Parameter auf ihren jeweiligen Initialwert.
- **setZoom**: setzt einen passenden Skalierungsfaktor. Dieser ist abhängig von Größe der Bounding-Box und der Kamera-Position, die durch die Touchscreen-Aktion – das MotionEvent – aktualisiert wird.
- **updateMMCamera**: diese Methode aktualisiert die MegaMol-Kamera. Sobald sich die Kamera auf dem Tablet bewegt, wird diese Methode aufgerufen, um die aktuellen Kamera-Parameter durch die Klasse Connector zu MegaMol zu senden.

Methoden, die sich um Objektinitialisierung und -änderung kümmern:

- **setObject**: eine public Methode, die bei der Menüauswahl aufgerufen wird, um die zu rendernden Objekte zwischen „Online-PDB-Dateien“ und „MegaMol-Daten“ umzuschalten.
- **initObject**: wählt die zu rendernden Objekte und setzt ein Flag, der die erfolgreiche Initialisierung meldet. Bis zum nächsten Laden von Objekt-Daten muss keine weitere Objekt-Initialisierung durchgeführt werden. Somit spart der Prozess bei großen Datensätzen viel Aufwand.
- **initMolecule** und **initPDB**: initialisieren die von MoleculeConnector() empfangenen Atom-Daten aus MegaMol bzw. die durch PDBLoader() eingelesene Online-PDB-Datei. Die Eigenschaften von Atomen wie Positionen (Atommittelpunkt), Radien und Farben werden jeweils in ein Vertexbuffer-Objekt geschrieben (siehe Listing 5.3) und mit einem Vertex-Shader-Handle verbunden. Außerdem wird hier noch der Skalierungsfaktor bezüglich der Bounding-Box berechnet. Dieser Faktor wird später bei der Transformationsberechnung verwendet.

Listing 5.3 Vertexbuffer-Objekt

```
ByteBuffer bb = ByteBuffer.allocateDirect(vertex_array.length * 4);
bb.order(ByteOrder.nativeOrder());
Framebuffer vbo = bb.asFloatBuffer();
vbo.put(vertex_array);
vbo.position(0);
```

Methoden, die beim Rendering helfen:

- **getShaderCode** und **loadShader**: getShaderCode sucht die Shader-Datei, lädt den Inhalt und sendet den Code als Zeichenkette (`String`) zurück. LoadShader kompiliert den Shader-Code und sendet ihn zur OpenGL ES API.
- **createFBO**: erstellt ein Framebuffer-Objekt. Mehr über Framebuffer ist in Abschnitte 4.3 zu finden.

Shader-Dateien

Shader-Dateien werden üblicherweise in einem Ordner »raw« unter »res« gespeichert. Jeder Render-Modus benötigt zwei Shader-Dateien: Vertex-Shader und Fragment-Shader. In der MegaMol-Applikation werden jedes Frame zwei Renderings durchgeführt: ein *off-screen*-Rendering für den Tiefentest und ein *on-screen*-Rendering für das Endergebnis auf dem Bildschirm. Dieser Teil wird in Kapitel 6.2 ausführlich beschrieben.

5.1.4 PDBLoader + Color

Die Klasse »PDBLoader« lädt eine PDB-Datei von einer Online-Datenbank [BWF+00] und setzt die entsprechenden Elemente anhand einer Farbtabelle (*Colortable*) auf bestimmte Farben. Nach dem Einlesen einer PDB-Datei stößt PDBLoader ein Flag an, der der Klasse »Renderer« den vollständigen Zugriff auf die Molekül-Daten erlaubt.

Beide Klasse bilden zusammen eine vereinfachte Version des PDBLoaders von MegaMol nach.

5.1.5 CameraConnector

Die Klasse »CameraConnector« baut eine Socket-Verbindung mit MegaMol auf und sendet und empfängt Kamera-Parameter.

Die Methode Connection() läuft als ein Thread, der eine Socket-Verbindung zwischen MegaMol und Tablet herstellt. Nach dem Aktivieren der Verbindung empfängt Receive() die initialen Kamera-Parameter aus MegaMol, um die Tablet-Kamera mit der MegaMol-Kamera zu synchronisiern. Diese Synchronisation findet nur ein einziges Mal statt und das Tablet übernimmt danach die Kontrolle über die MegaMol-Kamera durch den Aufruf

der public Methode send(). Obwohl die MegaMol-Kamera noch durch eine angeschlossene Maus bewegt werden kann, wird diese wieder mit der Tablet-Kamera synchronisiert, sobald MegaMol einen neues Parameter-Daten-Array vom Tablet empfängt.

5.1.6 MoleculeConnector

Die Klasse »MoleculeConnector« empfängt PDB-Daten vom MegaMol-Server. Dieser Vorgang unterteilt sich in zwei Schritte: Zuerst wird die Atomanzahl des Molekül-Datensatzes empfangen. Mit dieser Zahl wird ein Byte- und ein Float-Array initialisiert. Danach wird der empfangene Daten-Stream in das Byte-Array gespeichert, das Format wird in Float-Zahl umgewandelt und in das Float-Array übertragen.

Nachdem die Daten vollständig bearbeitet wurden, wird ein Flag `newData` auf true gesetzt, um die Applikation über neue Daten zu informieren. Das Float-Array wird am Ende an »Renderer« weitergeleitet und das neue Rendering beginnt mit der Initialisierung des Objekts. Abbildung 5.6 zeigt das Ergebnis der dynamischen Visualisierung.

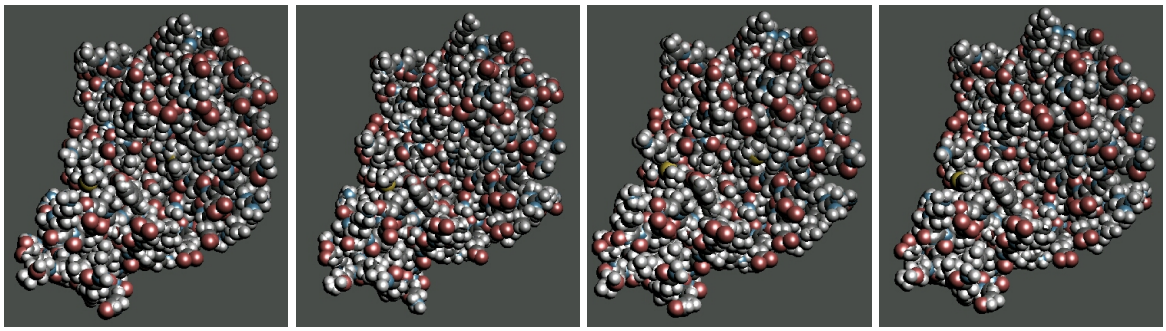


Abbildung 5.6: Visualisierung mehrerer Zeitpunkte der aus MegaMol übertragenen dynamischen Datensätze.

Für Internet-Anwendungen muss in der Manifest-Datei folgende Deklaration ergänzt werden:

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
```

5.2 Neu in MegaMol

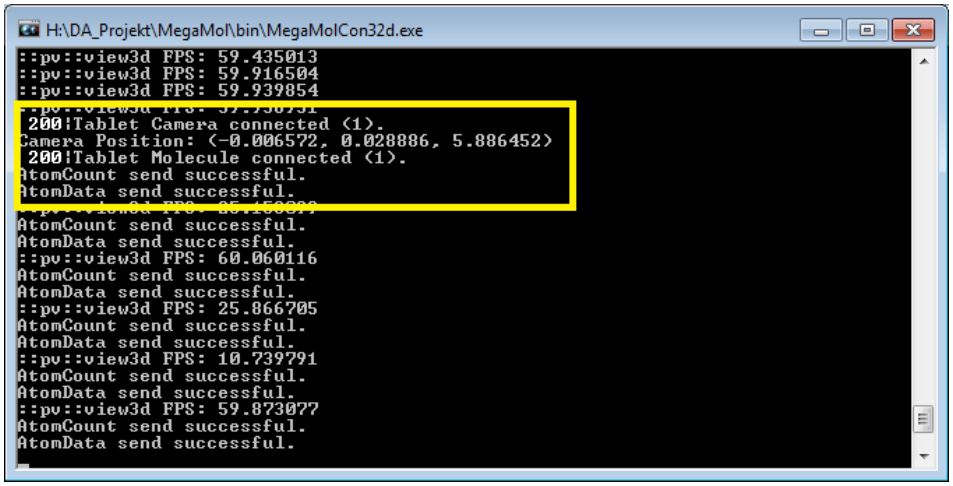
Damit MegaMol in der Lage dazu ist, mit dem Tablet über Socket-Verbindungen zu kommunizieren, sind einige neue Klassen notwendig, welche die Netzwerkverbindung übernehmen. Weil das Tablet hauptsächlich die Molekül-Daten von MegaMol visualisiert, die Kamera von MegaMol steuert und MegaMol-Parameter modifiziert, muss das Tablet über insgesamt drei unterschiedliche Schnittstellen mit MegaMol kommunizieren.

5.2.1 Teilaufgabe Kamerasteuerung

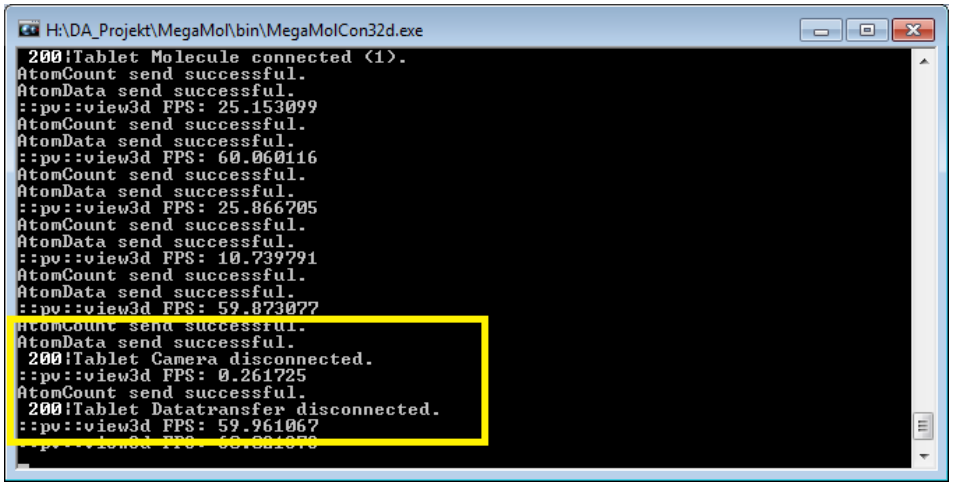
Diese Aufgabe betrifft zwei Klassen: »View3DTablet« und »TabletCameraConnector«.

»**TabletCameraConnector**« ist ein Connector, der einen PC mit MegaMol als Server fungieren lässt. Nachdem die Verbindung erstellt wurde, sendet der Server zuerst einen Kamera-Parameter an das Tablet, um die Kameras von MegaMol auf dem PC und dem Tablet zu synchronisieren. Danach wird die MegaMol-Kamera ausschließlich durch das Tablet gesteuert. (Steuerung mit Maus ist zwar erlaubt, aber sobald der Server neue Signale vom Tablet empfängt, lässt er die MegaMol-Kamera mit der Tablet-Kamera synchronisieren.)

»**View3DTablet**« ist eine Vererbung von View3D (d.h. View wird so dargestellt wie in View3D) und empfängt Kamera-Parameter des Tablets, falls neue Werte gesendet wurden.



(a) Socket-Verbindungen erstellt



(b) Socket-Verbindungen getrennt

Abbildung 5.7: MegaMol Verbindungsinfomationen.

5.2.2 Teilaufgabe Molekül-Datenübertragung

Diese Aufgabe betrifft zwei Klassen: »TabletMolecureConnector« und »SimpleMoleculeRenderer«

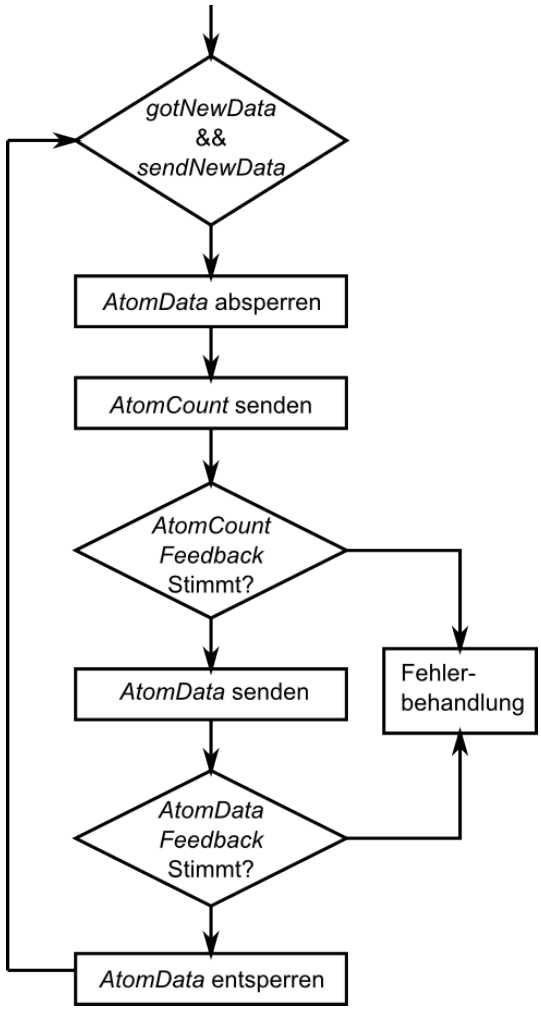


Abbildung 5.8: Übertragung von MegaMol-Daten

»TabletMolecureConnector« stellt wie »TabletCameraConnector« eine Socket-Verbindung her, allerdings wird nur diese Socket-Verbindung benutzt, um die von MegaMol geladenen PDB-Daten in einem eigenen Format an die Applikation zu senden. Der Prozess wird so lange wiederholt, bis die Verbindung getrennt wird. Dies wird durch eine `While`-Schleife mit Anfrage der Verfügbarkeit der Socket-Verbindung realisiert. Die Datenübertragung innerhalb der `While`-Schleife ist in Abbildung 5.8 skizziert.

In der ersten Verzweigungsraute wird überprüft, ob der Connector neue Daten senden muss (*gotNewData*) und ob die Applikation bereit ist, neue Daten zu empfangen (*sendNewData*).

Sind die Anfangsbedingungen erfüllt, beginnt der Sende-Prozess. Zuerst muss das Array mit dem aktuellen Frame des Moleküls als gesperrt markiert werden, sonst erhält die Applikation falsche Daten, wenn das Frame während dem Absenden mit Daten des nächsten Frames überschrieben wird.

Wenn das Daten-Array abgesichert ist, wird die Gesamtzahl der Atome zum Tablet gesendet. Der Server wartet daraufhin auf das Feedback der App. Ist die Zahl der Atome korrekt, werden die gesamten Atom-Daten im Byte-Array-Format übermittelt. Falls nicht, kommt es zur Fehlerbehandlung. Nach dem Senden des Byte-Arrays wartet der Server ebenfalls auf das Feedback der Applikation. Ist alles in Ordnung, kann das Daten-Array entsperrt werden. Wenn nicht, startet die Fehlerbehandlung wieder.

»SimpleMoleculeRenderer« ist eine bereits vorhandene Klasse, die Informationen von Atomen aus PDB-Daten einliest und Moleküle rendert. Diese Klasse wurde wie folgt ergänzt:

- ein Float-Array, das die Position, Radius und Farbe aller Atome beschreibt. Das Array wird auf Wunsch vom System zum TabletMoleculeConnector gesendet.
- ein binärer Wert, der angibt, ob sich die Atom-Daten (meistens nur Positionen) geändert haben.
- ein binärer Wert, der als Feedback vom Connector ist. Der zeigt an, ob der Connector bereit ist, neue Daten zu empfangen.
- ein Zeitzähler, der die Dauer misst, welche die Sendung des letzten Frames in Anspruch genommen hat. Ist dieser Wert ausreichend, dürfen neue Daten an den TabletMoleculeConnector gesendet werden, ansonsten wartet MegaMol. Dies ist nützlich, falls die Netzwerkverbindung nicht ausreichend schnell für große Datensätze ist. Während der Wartezeit ist die Aktualisierung des Float-Arrays erlaubt, damit das Tablet immer die aktuellsten Daten anzeigen kann.

5.2.3 Teilaufgabe Parametermodifizierung

Diese Aufgabe betrifft zwei Klassen: »TabletParamsConnector« und »SimpleMoleculeRenderer«.

»TabletParamsConnector« arbeitet prinzipiell identisch zu »TabletCameraConnector«. Die Klasse sendet und empfängt Parameter über eine Socket-Verbindung vom MegaMol-Control-Panel. Weil sich die Parameter nicht oft ändern, wird die Verbindung jedes Mal neu erstellt, sobald der Parameter-Editor in der Applikation geöffnet ist, und getrennt, wenn er deaktiviert ist. Somit wird überflüssige Netzwerkkommunikation gespart.

»SimpleMoleculeRenderer« wird wie im letzten Abschnitt erwähnt leicht ergänzt, um die Parameter an »TabletParamsConnector« zu übertragen. Außerdem überprüft die Klasse, ob die Verbindung der Parameter-Übertragung noch aktiv ist. Falls die Verbindung schon geschlossen wurde, bereitet »SimpleMoleculeRenderer« einen neuen Socket für die nächste Verbindung vor.

Abbildung 5.9 zeigt das Ergebnis der Modifikation des Paramters „Farbton“. Der auf dem Tablet bestimmte Wert wird an MegaMol gesendet und bringt Änderung an der Visualisierung in MegaMol (links). Die Änderung wird durch »TabletMolecureConnector« zurück an das Tablet gesendet und die Daten werden auf dem Tablet gerendert (rechts).

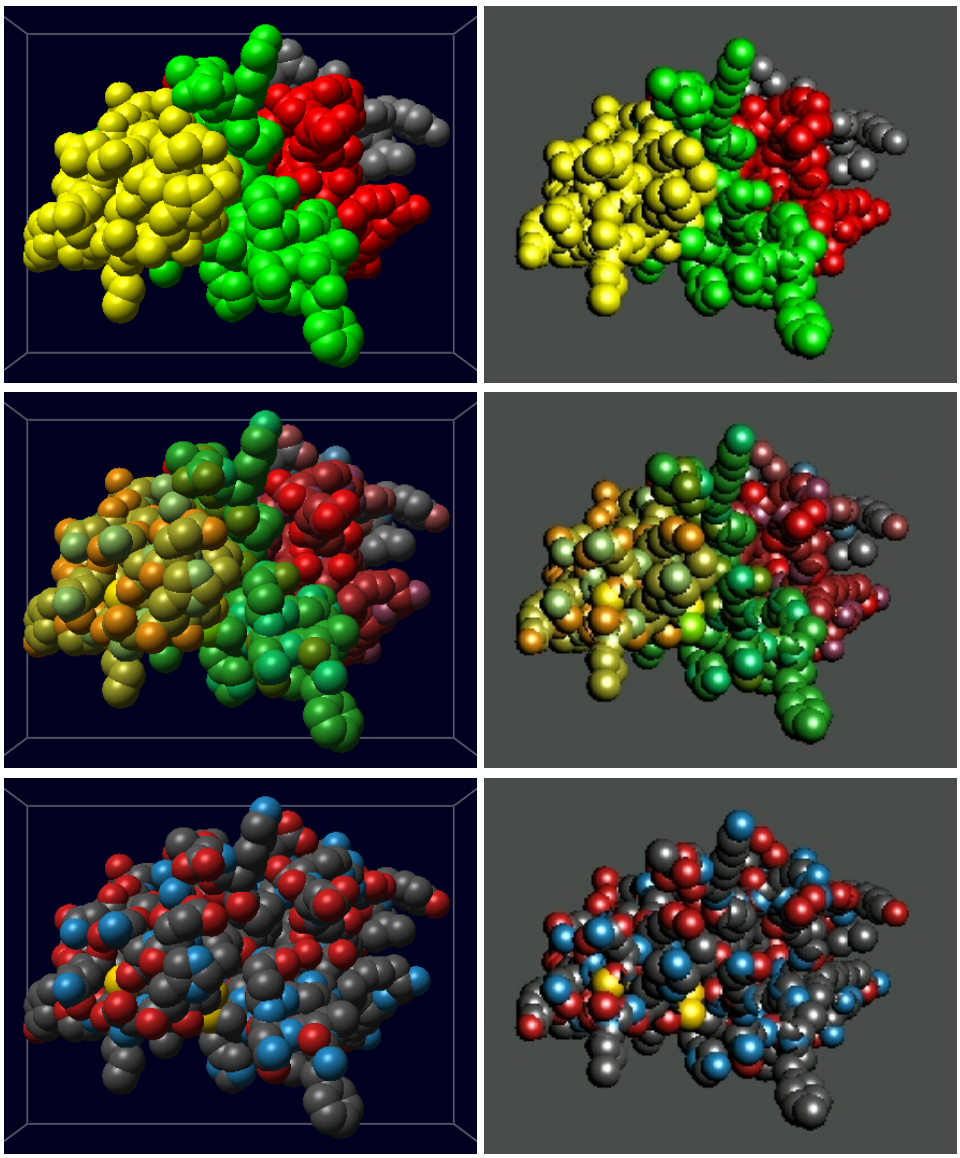


Abbildung 5.9: Parameter-Modifikation mit Tablet. Links: Ergebnisse in MegaMol. Rechts: Ergebnisse auf dem Tablet.

6 Schwierigkeiten und Lösungen

Schwierigkeiten bei der Applikationsentwicklung sind meistens in den Einschränkungen von Seiten des Tablets zu suchen: der limitierte Arbeitsspeicher und das vereinfachte OpenGL ES im Vergleich zu OpenGL bieten weniger Spielraum. Um die Molekül-Daten von MegaMol zu empfangen und korrekt zu visualisieren sind daher auch Umweg von Nöten, unter Umständen auch Tricks.

6.1 Kugel-Rendering

In der Applikation werden alle Atome eines Moleküls in Kugelform dargestellt und üblicherweise simuliert man Kugel-Oberflächen durch eine Vielzahl von Dreiecken. Vor diesem Hintergrund müssen umfangreiche geometrische Modelle berechnet werden, welche dann für jedes Atom unter Umständen hunderte Dreiecke umfassen, um den Eindruck von Glattheit zu vermitteln. Der Prozess muss für alle Atome eines Moleküls eventuell tausendfach wiederholt werden. Möchte man auch beim Heranzoomen die Illusion einer glatten Oberflächen aufrechterhalten, bedarf es einer Verfeinerung der Geometrie – d.h. es müssen weitere Dreiecke gezeichnet werden und die Struktur muss neu berechnet werden. All das übertrifft zusammengefasst bereits die Leistung eines Tablets bei Weitem, daher müssen Alternativen gesucht werden.

Außer Dreiecken bietet OpenGL ES als Primitive noch Punkte und Linien an. Zwar kann man eine Atom-Kugel durch einen Punkt ersetzen, da ein Punkt in OpenGL ES als ein abgerundetes Quadrat mit einem Zentrum (`gl_PointCoord`) und Radius (`gl_PointSize`) dargestellt wird, jedoch besitzt `gl_PointSize` einen Grenzwert. Wird dieser Wert überschritten, kommt es zu Problemen wie Aliasing oder der ungewollten Verzerrung des Punktes zu einem Quadrat.

Eine elegante Lösung[Lar11][TCM06] für dieses Problem ist hierbei einen 3D-Impostor im Shader-Rendering zu verwenden: statt einer 3D-Kugel verwendet man einen 2D-Kreis, der sich jederzeit orthogonal zur Kamera ausrichtet, wie Abbildung 6.1 zeigt.

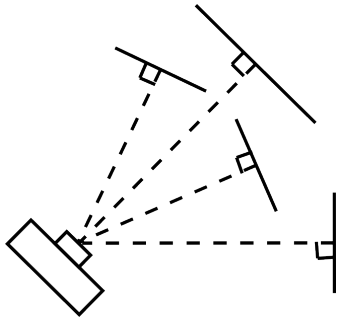


Abbildung 6.1: 3D-Impostor

Der erste Schritte ist, mit Hilfe von Mittelpunkt und Radius eines Atoms einen glatten orthogonalen 2D-Kreis zu erzeugen. Dazu definiert man ein Quadrat, welches exakt auf dem gewünschten Mittelpunkt zentriert ist. In

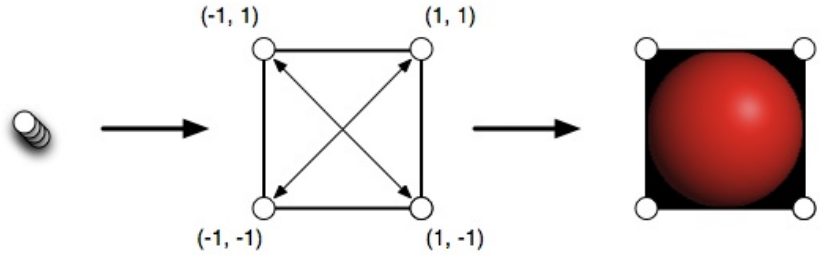


Abbildung 6.2: Erzeugung einer Einheitskugel mittels 3D-Impostor (Quelle:[Lar11])

OpenGL ES wird dies unter Verwendung zweier Dreiecke realisiert. Im Anschluss werden nur die Pixel auf dem Birdschirm gezeichnet, die sich innerhalb des Kreises befinden. Auf diese Art und Weise erfordert jedes Atom lediglich das Rendern von zwei Dreiecken, unabhängig von der Atom-Größe oder der Zoomstufe.

Der nächste Schritte bringt den 2D-Kreis-Ebene dazu sich immer um die Kamera zu drehen. Mit Hilfe Zweier Vektoren, die die Kamera-Richtung nach oben und die Kamera-Blickrichtung beschreiben, kann die Drehung der Ebene leicht bestimmt werden. Da die Kamera auch ein Objekt im Weltkoordinatensystem ist, können die zwei Vektoren direkt aus der inversen Model-View-Matrix ausgelesen werden.

Listing 6.1 zeigt wie man den 3D-Impostor durch OpenGL ES Shader erzeugen kann. Hier entspricht das „`inputImpostorSpaceCoordinate`“ einer Ecke des Quadrats. Insgesamt gibt es für ein Atom 6 Vertex-Berechnungen.

Listing 6.1 Shader-Code für 3D-Impostor

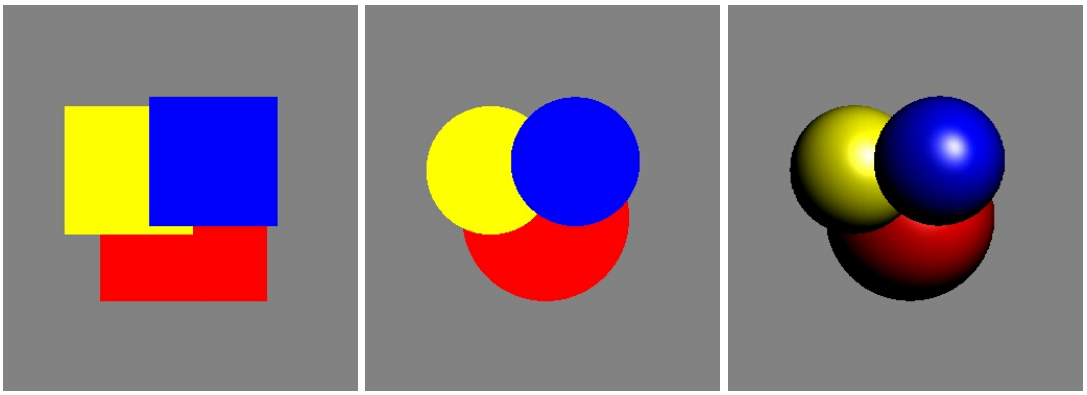
```
Vertex Shader Code:
        ...
        vec3 circle = camUp * inputImpostorSpaceCoordinate.y +
                              camRight * inputImpostorSpaceCoordinate.x;

        transformedPosition = atomPosition + vec4(circle, 0.0) * atomRadius ;
        transformedPosition = uMVPMatrix * transformedPosition;

        gl_Position = transformedPosition;
        ...

Fragment Shader Code:
        ...
        if (distanceFromCenter > 1.0) {
        discard;
        }
        ...
```

Das Ergebnis ist dargestellt in Abbildung 6.3(a). Hier werden lediglich drei sich miteinander schneidende Kugeln zur Veranschaulichung verwendet. Sichtbar sind drei Vierecke, welche sich gegenseitig laut Entfernung von ihrem jeweiligen Mittelpunkt zur Kamera hin bedecken. Bei einer Kreisform ist nur eine Überprüfung im Fragment-Shader erforderlich, um festzustellen, ob sich der Pixel innerhalb dieses Kreis befindet, siehe Fragment Shader in Listing 6.1 mit dem Ergebnis in Abbildung 6.3(b). Die Testzeile im Fragment-Shader kann später durch einen Test im Raycasting ersetzt werden.



(a) Quadrat-Ebene **(b)** Kreis-Ebene **(c)** Raycasting

Abbildung 6.3: Kugel-Rendering

Bis jetzt werden die Kugeln mit orthogonaler Projektion gerendert. Um die perspektivische Verzerrung zu gewinnen, wird die Raycasting-Technik [Gum03] [KSE04] [RE05] in Fragment-Shader verwendet, siehe Listing 6.2.

Unter Raycasting versteht man, die Sehstrahlen aus der Kamera zu allen Bildschirmpixeln zu verfolgen und ihre Schnittpunkte mit dem zu rendernden Objekt zu bestimmen, um damit die Volumendaten zu visualisieren. Gibt es keinen Schnittpunkt, dann wird dieser Pixel nicht gerendert bzw. nur mit der Standard-Hintergrundfarbe gezeichnet. Wenn der Strahl einen oder mehrere Schnittpunkte mit den Objekten besitzt, wird nur der (zur Kamera) vorderste Punkt als sichtbar notiert und gerendert (sieh Abbildung 6.4).

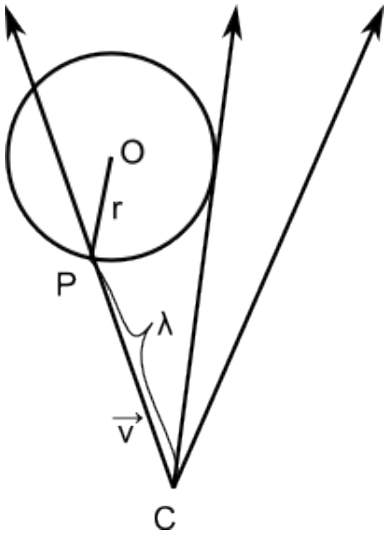


Abbildung 6.4: Raycasting

[RE05] stellte eine Methode vor, wie der Raycasting-Algorithmus im Fragment-Shader realisiert wird. Hier wird zunächst das zu rendernde Fragment mit Hilfe der Inversen der Model-View-Projektions-Matrix (`uMVPInvMatrix` in Listing 6.2) in ein lokales Koordinatensystem mit Kugelmittelpunkt als Ursprung umgerechnet. Auf dieser Basis kann der Sehstrahl-Vektor bestimmt werden.

Listing 6.2 Raycasting und Beleuchtung in Fragment-Shader-Code

```
    ...
    vec3 fragPosNDCS = vec3((gl_FragCoord.x / windowSizeDCS.x) * 2.0 - 1.0,
                        (gl_FragCoord.y / windowSizeDCS.y) * 2.0 - 1.0,
                        2.0*gl_FragCoord.z-1.0);

    vec3 intersectionPointGCS = vec3(0.0);

    // ray casting
    vec4 fragPosGCS = uMVPInvMatrix*vec4(fragPosNDCS, 1.0);
    fragPosGCS /= fragPosGCS.w;
    fragPosGCS.xyz -= atomCenter;
    vec4 viewDirGCS = vec4(normalize(fragPosGCS.xyz - cameraPosGlyphCS.xyz), 1.0);

    float rad = radius;
    float a = dot(viewDirGCS.xyz, viewDirGCS.xyz);
    float b = 2.0 * dot (viewDirGCS.xyz, cameraPosGlyphCS.xyz);
    float c = dot(cameraPosGlyphCS.xyz, cameraPosGlyphCS.xyz) - rad * rad;
    float discr = b * b - 4.0 * a * c;

    if (discr < 0.0) {
           discard;
    }

    float lambda = (-b - sqrt(discr)) / (2.0 * a);

    if (lambda < 0.0) {
      lambda = (-b + sqrt(discr)) / (2.0 * a);
    }

    intersectionPointGCS = -lambda * viewDirGCS.xyz + cameraPosGlyphCS.xyz;

    // blinn-phong
    vec3 V = -normalize(viewDirGCS.xyz);
    vec3 N = normalize(intersectionPointGCS);
    vec3 L = normalize(lightPosGlyphCS - intersectionPointGCS);

    vec3 h = normalize(V + L);
    float kd = max(0.0, dot(L, N));
    float ka = 0.0;
    float ks = 0.9 * pow(max(dot(N, h), 0.0), 30.0);

gl_FragColor = vec4(vec3(kd + ka) * color + ks * lightColor, 1.0);
    ...
```

Anschließend wird berechnet, ob es entlang diesem Sehstrahl Schnittpunkte zum Kugel-Objekt gibt. Ist P ein Punkt auf dem Sehstrahl und C die Kameraposition, dann wird folgende Formel erfüllt [Gum03]:

$$P(\lambda) = \lambda\vec{v} + C,$$

wobei $\vec{v}$ der nomalisierte Sehstrahl-Vektor ist und λ die Länge des Strahls. Angenommen, $\tilde{P}$ ist ein Schnittpunkt mit der Kugel, dann ist $\tilde{\lambda}$ genau dann die Distanz von Kamera bis zum Schnittpunkt, wenn $\left|\overrightarrow{O\tilde{P}}\right| = r$ ist. r ist hier die Länge des Radius. Ersetzt man $\left|\overrightarrow{O\tilde{P}}\right|$ durch r, so lässt sich $\tilde{\lambda}$ berechnen:

$$\tilde{\lambda} = \frac{-2\vec{v} \times \left|\overrightarrow{OC}\right| \pm \sqrt{(2\vec{v} \times \left|\overrightarrow{OC}\right|)^2 - 4\left|\vec{v}\right|^2 (\left|\overrightarrow{OC}\right|^2 - r^2)}}{2\left|\vec{v}\right|^2}$$

Wenn eine solche Lösung existiert, dann findet man einen Schnittpunkt. Der Schnittpunkt mit dem kleineren positiven $\tilde{\lambda}$ ist näher zu Kamera und wird mit der Blinn-Phong-Beleuchtung gerendert.

6.2 Tiefentest

Wie im vorigen Abschnitt beschrieben, werden 3D-Objekte durch 2D-Impostoren repräsentiert. Dies erspart zwar die Berechnung der polygonalen Geometrie, bringt jedoch auch das Problem des korrekten Renderings von sich überlappenden Objekten. Da die Kugeln lediglich aus einer flachen Ebene bestehen, können sie sich nicht auf die gewünschte, geometrische Weise „schneiden“. Daher kann das korrekte Rendering nicht mit dem gewöhnlichen Tiefentest der GPU gelöst werden.

Kommt die OpenGL-Shading-Language zum Einsatz, kann `gl_FragDepth` im Fragment-Shader dazu verwendet werden, um einen extern berechneten Tiefenwert des zu rendernden Fragments zu ermitteln. Mit Hilfe dieses vorgegebenen Tiefenwerts „weiß“ die GPU, welches Objekt das vorderste in diesem Fragment ist. Die Idee wurde aufgegriffen von [TCM06].

Unglücklicherweise wurde `gl_FragDepth` in OpenGL ES 2.0 gestrichen, weshalb eine Alternative gewählt werden muss, um die überlappenden Ränder korrekt zu zeichnen.

Alternative zur gl_FragDepth

Da das vorangehende Beispiel zum Kugel-Rendering bereits vorliegt, könnte man theoretisch dieses Beispiel weiter ausführen und dessen Lösung in der Arbeit implementieren.

In der iOS-Applikation von [Lar11] wird Folgendes vorgenommen: Das Framebuffer-Objekt wird für die Berechnung einer Textur benutzt, die so groß ist wie der Bildschirm. Die Textur wird auf eine solche Art und Weise erzeugt, dass der Farbwert dem Tiefenwert auf jedem

Fragment entspricht. Um die Reihenfolge der überlappenden Objekte zu unterscheiden, wird die Variable `GL_MIN_EXT` in der OpenGL-Extension benutzt, um das näheste Objekt zu identifizieren. Im Anschluss darauf findet das zweite Rendering statt und die eben berechnete Textur wird als Tiefenwerte-Map verwendet, um zu entscheiden, ob ein Fragment am nähesten zur Kamera ist. Es wird nur dann gerendet, wenn ihr Tiefenwert mit dem Wert in der Textur übereinstimmt, ansonsten wird das Fragment verworfen, weil es noch von anderen Objekten verdeckt wird.

Die Implementierung der beschriebenen Methode führt leider nicht zum Erfolg, weil Android `GL_MIN_EXT` nicht unterstützt. Da `GL_MIN_EXT` jedoch eine Variable der OpenGL-ES-Extension »EXT_blend_minmax« ist, fördert es die Idee, direkt in den offiziellen Extensions von Khronos [Reg] eine Ersatzmöglichkeiten zu suchen.

In den Extensions findet sich `gl_FragDepthEXT`, ein Ersatz für `gl_FragDepth`. Die Variable befindet sich in der Extension `GL_EXT_frag_depth`. Diese Extension fügt die eingebaute Variable `gl_FragDepthEXT` dem Fragment-Shader hinzu und ermöglicht so die Verwendung eines Tiefenwerts im Fragment-Shader.

Darüberhinaus existiert eine weitere Extension `GL_OES_depth_texture`. Sie definiert ein zusätzliches Texturformat, in der der Tiefenwert gespeichert wird. Dies ist theoretisch eine weitere Möglichkeit der Implementierung der Methode von [Lar11].

Erschwerenderweise kommt hinzu, dass keine der beiden Extensions von Android bzw. vom Tegra-Prozessor unterstützt wird.

In diesem Fall bleibt nur übrig, die Tegra-Extensions zu durchsuchen, welche den Entwicklern zur Verfügung stehen. Hierbei wird man schließlich bei »GL_NV_shader_framebuffer_fetch« fündig. Damit kann das Problem des Tiefentests über Umwege gelöst werden.

Um diese Extension (und auch jegliche andere Extension) benutzen zu können, muss man sie in der Shader-Dateien aktivieren. Die folgende Deklaration muss vor der ersten Verwendung im Fragment-Shader stattfinden. Sowohl "`enable`" als auch "`require`" sind erlaubt.

```
#extension GL_NV_shader_framebuffer_fetch : require
```

Die Extension »GL_NV_shader_framebuffer_fetch« definiert eine read-only eingebaute Variable `gl_LastFragColor`, in der der letzte Output des Fragments gespeichert wird. Weitere Informationen findet man im Referenzbuch [NVI11], S. 24.

Kugel-Rendering mit korrektem Tiefenwert

Das Rendering lässt sich in zwei Phasen einteilen: ein *Off-Screen*-Rendering im Framebuffer-Objekt, das die Tiefenwerte zu einer 2D-Textur umwandelt; ein *On-Screen*-Rendering, das das Objekt durch Raycasting und Beleuchtung auf den Bildschirm rendert.

In Kapitel 6.1 ist bereits dargestellt, wie Ray-Casting anhand eines 2D-Impostors eine 3D-Oberfläche rendert. Im Zwischenschritt wird die Position jedes sichtbaren Punktes

eines Objekts berechnet. Diese Position kann bei der Berechnung des richtigen Fragment-Tiefenwerts direkt verwendet werden. Listing 6.3 zeigt den Fragment-Shader-Code.

Die erste Zeile beschreibt, dass die Punktposition durch Ray-Casting berechnet wird. Darauf folgt die Bestimmung der Entfernung des Punktes bis zur Kamera. Der Wert wird auf den Bereich [0.0, 1.0] normiert und jedes Mal mit der vorherigen in dieses Fragment geschriebenen Farbe, auf die durch den Aufruf `gl_LastFragColor` zugegriffen werden kann, verglichen. Bei dem Vergleich wird der kleinere Wert behalten, denn kleiner bedeutet in diesem Fall näher zur Kamera. Das jeweilige Ergebnis wird für den Vergleich mit weiteren Entfernungswerten gespeichert. Als Endergebnis liegt ein Bild mit ausschließlich Grauwerten vor – je dunkler, desto kleiner ist der Tiefenwert eines Fragments und desto näher befindet es sich an der Kamera. Es wird als 2D-Textur von der Größe des Bildschirms gespeichert.

Listing 6.3 1. Rendering: Tiefenwertberechnung auf Basis von Ray-Casting

```
...
intersectionPointGCS = -lambda * viewDirGCS.xyz + fragPosGCS.xyz;

float depth = dot(uMVPTransMatrix[2], vec4(intersectionPointGCS + atomCenter, 1.0));
float depthW = dot(uMVPTransMatrix[3], vec4(intersectionPointGCS + atomCenter, 1.0));
depth = ((depth / depthW) + 1.0) * 0.5;

gl_FragColor = vec4(vec3(depth), 1.0);

if( gl_LastFragColor.r < gl_FragColor.r && gl_LastFragColor.r < 0.99) {
       gl_FragColor = gl_LastFragColor;
}
...
```

Der Begriff und die Nutzung des Framebuffer-Objekts wurden bereits in Abschnitt 4.3 vorgestellt. Es wird verwendet, um die Tiefentextur Off-Screen zu rendern. Off-Screen bedeutet, das Ergebnis – in diesen Fall die Tiefentextur – wird nicht auf dem Bildschirm gezeichnet, sondern nur in das Framebuffer-Objekt geschrieben. Lässt man dieses Ergebnis trotzdem On-Screen zeichnen, bekommt man ein Bild wie in Abbildung 6.5(b) dargestellt. Hier ist der Tiefenwert allerdings schwer zu erkennen, da die Objekte nah beieinander liegen.

Ein weiteres Beispiel der Visualisierung von Molekül-Daten sieht man in Abbildung 6.6(b). Hier erkennt man eine deutliche Tiefentextur.

Außer den üblichen Einstellungen zur Nutzung des Framebuffer-Objekts muss bei der Tiefenwertberechnung auf folgende Punkte Acht gegeben werden:

- Der GPU-Tiefentest (`GL_DEPTH_TEST`) muss ausgeschaltet sein, siehe erste Zeile in Listing 6.4, ansonsten wird die Z-Position der 2D-Fläche für den Tiefentest verwendet. Ist `GL_DEPTH_TEST` ausgeschaltet, wird es in der Reihenfolge der Objekte gerendert. `GL_DEPTH_TEST` darf nicht mehr aktiviert werden.

- Die Hintergrundfarbe muss direkt nach Aktivierung des Framebuffer-Objekts auf Weiß (1.0, 1.0, 1.0) gesetzt werden, siehe Listing 6.4. Tiefenwert 1.0 bedeutet maximale Entfernung. Die gewünschte Hintergrundfarbe kann später beim On-Screen-Rendering zurückgesetzt werden.

Listing 6.4 Voraussetzung für eine erfolgreiche Tiefenwertberechnung

```
        ...
        GLES20.glDisable(GLES20.GL_DEPTH_TEST);
        ...
        //---- Beginn off-screen-Rendering mit Framebuffer-Objekt ----
        GLES20.glClearColor(1.0f, 1.0f, 1.0f, 0.0f);
        GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT);
        ...
```

Im Anschluß findet das *On-Screen*-Rendering statt. Dabei wird im Fragment-Shader ein „Custom_Depth_Test“ definiert und die im Framebuffer-Objekt gespeicherte Tiefentextur wird als Tiefenwert-Map verwendet. Der Tiefenwert jedes zu rendernden Fragments wird wie beim *Off-Screen*-Rendering bestimmt und mit der Tiefentextur verglichen. Ist der Tiefenwert größer, findet das Rendering nicht statt, da es an der Stelle noch ein anderes Objekt gibt, das sich näher an der Kamera befindet. Nur wenn der Tiefenwert (fast) identisch zur Tiefentextur ist, wird dieses Fragment mit dem gewünschten Farbwert gerendert.

Listing 6.5 2. Rendering: Tiefentest mit Hilfe der Tiefentextur

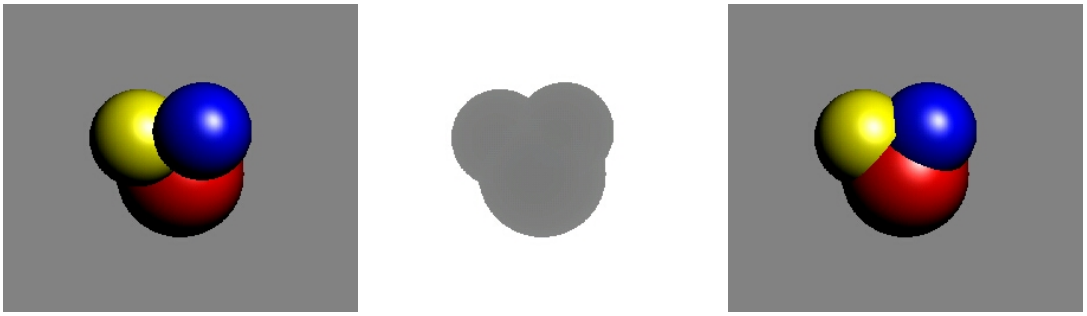
```
        uniform sampler2D depthBufferTex;
        ...
#define CUSTOM_DEPTH_TEST

void main(){
        ...
        float depth = dot(uMVPTransMatrix[2], vec4(intersectionPointGCS + atomCenter, 1.0));
        float depthW = dot(uMVPTransMatrix[3], vec4(intersectionPointGCS + atomCenter, 1.0));
        depth = ((depth / depthW) + 1.0) * 0.5;

#ifdef CUSTOM_DEPTH_TEST
        float lastDepth = texture2D(depthBufferTex, vec2((gl_FragCoord.x + 0.5) /
            windowSizeDCS.x, (gl_FragCoord.y + 0.5) / windowSizeDCS.y)).x;
        if( abs( lastDepth - depth) > 0.01 ) {
                discard;
        }
#endif
};
```

Listing 6.5 veranschaulicht den Ablauf eines Tiefentests. In der ersten Zeile wird die Tiefentextur mit dem Fragment-Shader verbunden und kann abgefragt werden. Die Tiefenwertberechnung für das zu rendernde Fragment ist gleich wie in Listing 6.3, daher ist der Tiefenwert auch vergleichbar mit der Tiefentextur. Der letzte Block ist der „Custom_Depth_Test“, wodurch das Endergebnis in Abbildung 6.5(c) zu sehen ist.

Jeder Frame der Applikation wird in den beschriebenen zwei Phasen gerendert, sichtbar ist allerdings immer nur das zweite Rendering-Ergebnis.



(a) ohne Tiefentest **(b)** Tiefenwert in Graustufen **(c)** mit Tiefentest

Abbildung 6.5: Kugel-Rendering mit Tiefentest

Dadurch wird der Tiefentest durchgeführt, allerdings ist dieser mit einem gewissen Fehler behaftet, welcher mit der Rechengenauigkeit zusammenhängt. Objekte mit sehr ähnlichem Tiefenwert werden nicht ausreichend genau miteinander verglichen, wie Abbildung 6.5(c) zeigt. Genauer gesagt liegt das Problem an der Bit-Länge der Float-Zahl von der verwendeten Hardware und lässt sich deshalb nicht ohne Weiteres lösen.

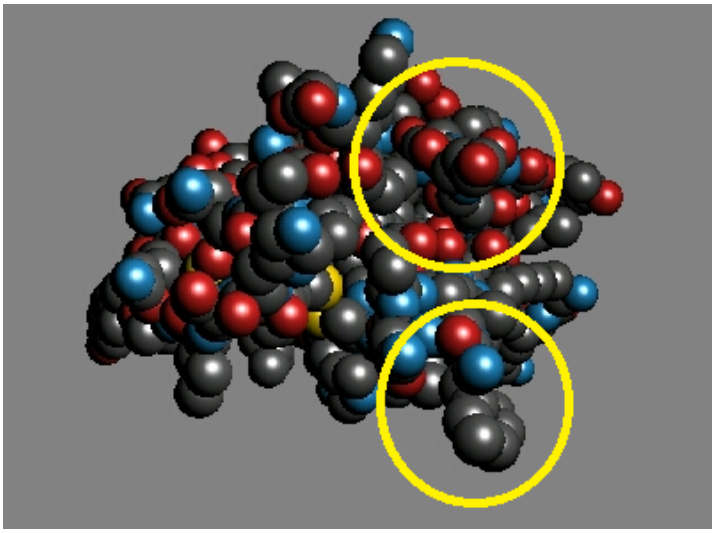
6.3 Socket-Verbindung

Die Kommunikation durch Socket-Verbindungen ist keine neue Technik, trotzdem gibt es hier viele Feinheiten, die schnell Probleme verursachen können. Bei einer Weiterentwicklung des Programms sollte man auch die folgenden Aspekte berücksichtigen.

6.3.1 Netzwerkverbindung als Thread

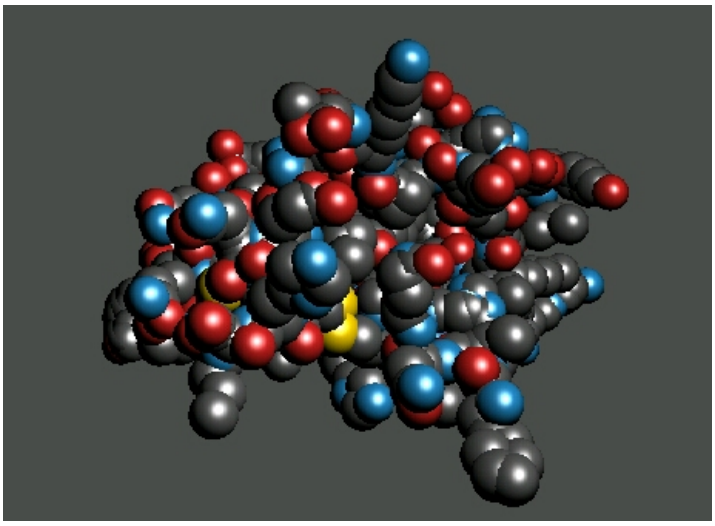
Ein Thread ist eine Folge von Anweisungen, die abgeschlossen sind und nacheinander ausgeführt werden[Gar11]. Eine CPU kann sich zu jedem Zeitpunkt nur mit einem einzelnen Thread befassen. Mehrere Threads müssen sich die CPU-Zeit teilen und die Aufgaben werden zeitversetzt abgearbeitet, was den Eindruck erweckt, dass sie „gleichzeitig“ bearbeitet werden.

Da immer mehr Smartphones, allen voran Tablets, mehrere Prozessor-Kerne besitzen, ist es nun kein Problem mehr, mehrere Threads gleichzeitig laufen zu lassen. Die Aufgaben in unterschiedlichen Threads blockieren sich daher nicht gegenseitig. Dies bietet einen immensen Vorteil bei denjenigen Abläufen, die viel Zeit beanspruchen, aber andere Prozesse nicht beeinflussen sollen, wie z.B. das Herunterladen von Dateien.



(a) ohne Tiefentest

(b) Tiefenwert in Graustufen



(c) mit Tiefentest

Abbildung 6.6: Molekül-Rendering mit Tiefenwerten.

Ab Android-Version 3 wird eine Netzwerkverbindung innerhalb einer Applikation nur über einen eigenen Thread erlaubt. Ziel ist es, Prozessblockierung wegen Netzwerkverbindungen oder Datenübertragungen zu vermeiden.

6.3.2 Byte-Reihenfolge und Byte-Array

Diese beiden Punkte sind von großer Wichtigkeit. Einer betrifft die Byte-Reihenfolge beider Verbindungsseiten, der andere, dass Float-Arrays bei Datenübertragungen nur schwierig zu verwenden sind und daher in einen Byte-Array umgewandelt werden.

Bei der Byte-Reihenfolge unterscheidet man zwischen Big-Endian und Little-Endian. In der Netzwerktechnik und Java kommt Big-Endian zum Einsatz, während C++ das Little-Endian-Schema verwendet. Für Integer gibt es `ntohl()` und `htonl()` zur Umwandlung zwischen Big-Endian und Little-Endian, bei Float ist es hingegen komplizierter. Hilfreich ist hier die C-Struktur Union.

Will man einen langen Float-Array senden, kann man dies nicht mehr mit einzelnen Floats umsetzen. Hier bevorzugt man einen Byte-Array. Die Umwandlung zwischen Float und Byte kann durch Nutzung einer Union-Struktur geschickt gelöst werden.

6.3.3 Paketlänge

Die Datenpaketgröße ist abhängig von Protokoll und Betriebssystem, aber auch von der Hardware. Obwohl das TCP-Protokoll eine hohe Grenze an Datenpakete setzt, sind Daten aufgrund eines Ethernet-Frames pro Paket auf ca. 1500 Bytes beschränkt. Für die Übertragung der Kamera-Parameter ist diese Länge ausreichend, allerdings nicht mehr für Daten von tausenden Atomen, die aus Position, Radius und Farbe bestehen. Hier wird der gesamte Bytebuffer automatisch vom Router in mehrere Datenpakete aufgeteilt. Abhängig von der Netzwerkgeschwindigkeit kommen die Datenpakete meistens nicht schnell genug an, bevor das Client-Programm die empfangenen Daten an weitere Bearbeitungsschritte sendet. Die nicht ausreichend schnell gelieferten Datenpakete entfallen dann und das Rendering ist nicht korrekt.

Der Client muss daher so lange warten, bis alle Daten in den Puffer gelesen sind. Die Länge des Streams ist wegen der Summe der Atom fest und kann vom Client daher vorausberechnet werden. Die Pufferung über eine definierte Datenlänge kann mit einer Schleife realisiert werden.

7 Ausblick

Da sich Mobilplattformen in der Informatik stetig zunehmender Beliebtheit erfreuen und die Hardware in kurzer Zeit große Leistungssprünge erfährt, steigen damit automatisch auch die softwareseitigen Anforderungen und die Komplexität der Methoden. Bereits während der Entwicklung der aktuellen Applikation kam es zu einigen unerwarteten Problemstellungen, die neue Lösungsansätze erforderten. Obwohl die gesteckten Ziele größtenteils erreicht werden konnten, bedeutet dies nicht, dass es keine Verbesserungsmöglichkeiten gibt. Eine mögliche Richtung für eine zukünftige Entwicklung der Applikation wird in den nachfolgenden Punkten grob umrissen.

- Grenzen des Tablets evaluieren und bessere Kriterien für den Datenaustausch

  Ein Tablet unterliegt im Vergleich zu einem Desktop-PC hinsichtlicher der verbauten Hardware starken Beschränkungen. Aus diesem Grund kann es passieren, dass es bei größeren Datensätzen nicht schnell genug rendern kann. Es wäre daher sinnvoll die Leistungsgrenze des Tablets genau auszuloten, um festzustellen welche Molekülgröße machbar ist. Die Frage ist also, wie viele Atome das Tablet unter den gestellten Anforderungen noch ausreichend schnell rendern kann. Besonders bei dynamischen Daten ist es kritisch, wie lange die Berechnung eines bewegenden Frames benötigt. Hier spielt in ähnlicher Weise auch die Netzqualität eine wichtige Rolle, denn die Übertragungsgeschwindigkeit hat einen direkten Einfluß auf die Visualisierung.

  Kennt man die Leistungsgrenze, könnte man die Visualisierung darauf (dynamisch) anpassen. Eine weitere Alternative zum direkten Rendering wäre, die Molekül-Daten auf einem leistungsfähigen Server rendern zu lassen und das Ergebnis als Bild/Textur an das Tablet zu übertragen.

- OpenGL ES 3.0 für mehr Genauigkeit

  In Kapitel 6.2 ist zu erkennen, dass es beim Tiefentest auf dem Tablet zu leichten Genauigkeitsproblem kommt. Das liegt sehr wahrscheinlich an der Genauigkeit der von OpenGL ES unterstützten Float-Zahlen. Das vor Kurzem angekündigte OpenGL ES 3.0 soll 32-Bit-Float unterstützen und damit die Rechengenauigkeit deutlich erhöhen. Da OpenGL ES Version 3.0 kompatibel zu Version 2.0 ist, ist der Umstieg auf eine höhere Version zu empfehlen. Außerdem kann in OpenGL ES 3.0 `gl_FragDepth` geschrieben werden, wodurch der Tiefentest automatisch durchgeführt wird. Allerdings wird die Tegra-3-Hardware vermutlich kein OpenGL ES 3.0 unterstützen.

- Weitere Visualisierungsmethoden

Das Desktop-Programm von MegaMol bietet verschiedene Proteinvisualisierungen an, wie z.B. Stick, Ball-and-Stick, SAS, Cartoon und Oberflächendarstellungen. Es wäre deshalb sinnvoll, diese Visualisierungen ebenfalls in der MegaMol-Applikation zu implementieren. Allerdings müssen auch hier die Beschränkung eines Tablets berücksichtigt werden.

- GUI für Parameter-Editor

  Der in der Preference eingebaut MegaMol-Parameter-Editor besitzt in der aktuellen Form nur 1 Variable. Dies gilt als erster Test eines Mechanismus zur Parameter-Modifikation, da der gewünschte Editor letztendlich eine eigene, benutzerfreundliche GUI besitzen soll. Diese Teilaufgabe konnte aus Zeitgründen nicht vollständig umgesetzt werden. Einige Ideen umfassen einen transparenten Editor, damit die Visualisierung im Hintergrund weiterlaufen kann. Hier kann eine eigens gestaltete Preference eingesetzt werden, oder sogar die internen Speicherung verwendet werden und mit einer eigenen Datei, statt „SharedPreferences“. Da die Parameter unterschiedliche Datenstrukturen besitzen, sollte jede Größe einzeln gesendet und emfangen werden, anstatt alle Parameter in einen Bytearray umzuwandeln.

- Menü zu Actionbar

  Seit Android-Plattform 3.0 (API level 11) gibt es die API „ActionBar“, welche Menüelemente direkt in den Action-Bar setzt. Dadurch kann der Benutzer die Funktionen in der Menüleiste direkt im Action-Bar auswählen, anstatt zuerst das Menü-Icon anzuklicken und dann in der Liste zu suchen. Der Action-Bar ist nicht nur eine Alternative zum Menü, er bietet zudem auch mehr Interaktivität an. Die aktuelle Android-Entwicklung zielt darauf ab, das Menü Stück für Stück durch den sogenannten Action-Bar zu ersetzen. Daher bietet sich dies als mögliche Idee für eine Weiterentwicklung der MegaMol-Applikation an.

8 Zusammenfassung

Diese Diplomarbeit handelt von der Entwicklung einer Android-Applikation, welche die Visualisierungssoftware MegaMol per Socketverbindung fernsteuert, aber auch Molekülda-tensätze eigenständig visualisieren kann. Die Applikation bietet darüberhinaus auch die Molekülvisualisierung an, ohne eine Verbindung zu MegaMol. Die dafür benötigten Schnitt-stellen in MegaMol, welche mit der Applikation interagieren, wurden ebenfalls ergänzt.

Die Entwicklung der Android-Applikation erfolgte in OpenGL ES 2.0 in Verbindung mit Java auf Eclipse und dem Android-SDK; die Erweiterung des MegaMol-Programms wurde in C/C++ vorgenommen. Die praktische Evaluation der Applikation erfolgte auf einem Tablet-PC, dem ASUS Transformer Prime TF201 mit dem Betriebssystem Android Plattform 4.0.3, API Level 15.

In den häufigsten Fällen hängen die Schwierigkeiten in der Umsetzung mit der Einschrän-kung von Tablet-Hardware und OpenGL ES zusammen. Als besondere Hürde sei hier das Kugel-Rendering erwähnt, welches durch ein 2D-Impostor optimiert bzw. überhaupt erst ermöglicht wurde. Das Problem des fehlenden Tiefen-Buffers wurde mit Hilfe von Off-Screen-Rendering im Framebuffer-Objekt und der Tegra-Extension für OpenGL ES 2.0 gelöst. Aus diesem Grund ist die Applikation spezifisch für die Tegra-Tablets.

Da die Popularität von Applikationen auf Mobilgeräten in den letzten Jahren drastisch angestiegen ist, schreitet auch die Entwicklung in Hard- und Software schnell voran und wird stetig verbessert. Die häufige Anpassung an den jeweils neusten Stand der Technik ist zwar mit Aufwand verbunden, aber gerade im Bereich der Computervisualisierung bieten sich dadurch bisher unerreichte Einsatzmöglichkeiten.

Literaturverzeichnis

[Ble12] C. Bleske. *Java für Android: Native Android-Apps programmieren*. Franzis, 2012. (Zitiert auf den Seiten 13, 15 und 19)

[BWF+00] H. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. Bhat, H. Weissig, I. Shindyalov, P. Bourne. The Protein Data Bank, 2000. http://www.pdb.org. (Zitiert auf den Seiten 10, 32, 33 und 40)

[dev] Android Developers. http://developer.android.com/. (Zitiert auf den Seiten 29 und 33)

[Dua10] The Benefits of Multiple CPU Cores in Mobile Devices (Whitepaper), 2010. http://www.nvidia.com/content/PDF/tegra_white_papers/Benefits-of-Multi-core-CPUs-in-Mobile-Devices_Ver1.2.pdf. (Zitiert auf Seite 11)

[Gar11] M. Gargenta. *Einführung in die Android Entwicklung*. O'Reilly, 2011. (Zitiert auf den Seiten 13, 15, 19 und 55)

[Gum03] S. Gumhold. Splatting Illuminated Ellipsoids with Depth Correction. In *VMV*. 2003. (Zitiert auf den Seiten 49 und 51)

[Khr12] The Khronos Group Inc. *OpenGL ES Version 3.0*, 2012. http://www.khronos.org/registry/gles/specs/3.0/es_spec_3.0.0.pdf. (Zitiert auf Seite 26)

[KSE04] T. Klein, S. Stegmaier, T. Ertl. Hardware-accelerated Reconstruction of Polygonal Isosurface Representations on Unstructured Grids. In *Proceedings of Pacific Graphics '04*, S. 186–195. 2004. (Zitiert auf Seite 49)

[Lar11] B. Larson. Enhancing Molecules using OpenGL ES 2.0, 2011. http://www.sunsetlakesoftware.com/2011/05/08/enhancing-molecules-using-opengl-es-20. (Zitiert auf den Seiten 47, 48, 51 und 52)

[MF12] R. Marucchi-Foino. *Game and Graphics Programming for iOS and Android with OpenGL ES 2.0*. John Wiley & Sons, Inc, 2012. (Zitiert auf den Seiten 27 und 28)

[MGS08] A. Munshi, D. Ginsburg, D. Shreiner. *The OpenGL ES 2.0 programming guide*. Addison-Wesley, 2008. (Zitiert auf den Seiten 23, 24, 25 und 27)

[NVI11] NVIDIA Corporation. *OpenGL ES 2.0 Development for the Tegra Platform*, 2011. http://developer.download.nvidia.com/assets/mobile/files/tegra_gles2_development.pdf. (Zitiert auf den Seiten 11 und 52)

[PAM+08] K. Pulli, T. Aarnio, V. Miettinen, K. Roimela, J. Vaarala. *Mobile 3D Graphics with OpenGL ES and M3G*. Morgan Kaufmann Publishers, 2008. (Zitiert auf den Seiten 23 und 24)

[Qua11] The Benefits of Quad Core CPUs in Mobile Devices (Whitepaper), 2011. http://www.nvidia.com/content/PDF/tegra_white_papers/tegra-whitepaper-0911a.pdf. (Zitiert auf Seite 11)

[REo5] G. Reina, T. Ertl. Hardware-Accelerated Glyphs for Mono- and Dipoles in Molecular Dynamics Visualization. In *EuroVis*, S. 177–182. Eurographics Association, 2005. (Zitiert auf Seite 49)

[Reg] Khronos OpenGL ES API Registry. http://www.khronos.org/registry/gles/. (Zitiert auf Seite 52)

[TCMo6] M. Tarini, P. Cignoni, C. Montani. Ambient Occlusion and Edge Cueing to Enhance Real Time Molecular Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 2006. (Zitiert auf den Seiten 47 und 51)

Alle URLs wurden zuletzt am 17. 08. 2012 geprüft.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Jing Sheng)