Michael Brempel

LipSync in Echtzeit-3D-Anwendungen (Schwerpunkt Computerspiele)

Theoretische und praktische Ausarbeitung eines Automatisierungsworkflows

Inhaltsverzeichnis

Vorwort

Nicht nur der libanesische Schriftsteller und Maler Chalil Djubran[1] bezeichnete die Lippen als Boten des Wortes. Die Lippenbewegungen und ihre entsprechenden Mundbewegungen sind mit ausschlaggebend für die bis heute am meisten genutzte Kommunikationsform des Menschen, des Sprechens. Das synchrone Einhergehen des gesprochenen Wortes mit den dazugehörigen Lippenbewegungen des Sprechers sind seit Jahrhunderten ein alltäglicher Bestandteil unseres Lebens.

Seit dem Beginn der digitalen Medien mit ihren umfangreichen Unterhaltungs- und Informationsangeboten strebt man nach einer möglichst realistischen Darstellung des Realen oder Pseudorealen.

Am geschichtlichen Werdegang der Computerspiele lässt sich dieser Trend sehr gut verfolgen. Während zur Pionierzeit der Computerspiele in den frühen 70er Jahren noch einfarbige Linien und Punkte ein Computerspiel wie „Spacewar“[2] oder „Pong“[3] ausmachten, so wird heutzutage großer Wert darauf gelegt, die visuellen Eindrücke wie Charaktere, Animationsbewegungen und Spieleumgebungen so realistisch wie möglich darzustellen. Unter anderem bedient man sich in aktuellen Spielen wie „Mafia“[4], „Halflife 2“[5], „Stalker“[6] u.a. sogar aufwändiger Techniken wie 3D-Scanning und MotionCapture-Systemen, um einen hohen Grad an Realismus zu erreichen.

Ein weiterer Baustein zum Erreichen dieses Zieles ist demnach auch das Übertragen des uns alltäglichen lippensynchronen Sprechens auf den 3D-Avatar eines Computerspiels.

Heutigen Anforderungen an Technik, Workflow und Resultat angepasst, behandelt diese Diplomarbeit das Gebiet der Lippensynchronisation in 3D-Echtzeit-Anwendungen in Theorie und Praxis, so dass wir am Ende eine Möglichkeit kennengelernt haben, diesen Baustein zu erstellen und zu verwenden.

Ich wünsche Ihnen viel Spass und Erfolg!

Michael Brempel

Furtwangen im November 2004

1 Khalil Gibran (Chalil Djubran), 1883-1931, „Der Geist“
2 Steve Russell, Massachusetts Institute of Technology (MIT), 1962
3 Magnavox, Atari, 1972
4 Illusion Softworks, Take2, 2002
5 Valve, Sierra, Vivendi Universal, 2004
6 GSC World Computing, THQ, 2005 geplant

Danksagung

An dieser Stelle möchte ich mich bei allen Leuten bedanken, die mich beim Schreiben der Diplomarbeit unterstützt haben. Dieser Dank gilt in erster Linie Herrn Prof. Schrödinger und Herrn Prof. Dr. Friedmann, die mich als betreuende Professoren während der Diplomarbeit begleitet haben. Besonderen Dank gilt auch Frank Brempel, der mich mit seinen programmier- und audiotechnischen Kenntnissen mehrere Male mit neuen Anregungen und Verbesserungsvorschlägen versorgt hat.

Kontakt

Die Diplomarbeit entstand im Wintersemester 2004/2005 an der Fachhochschule Furtwangen[7] im Studienbereich Medieninformatik des Fachbereichs Digitale Medien[8]. Bei Fragen und Unklarheiten können Sie sich gerne über meine Email-Adresse mbrempel@gmx.de oder über meine Internetseite http://www.mbrempel.de mit mir in Verbindung setzen.

7 Homepage der FH Furtwangen: www.fh-furtwangen.de
8 Homepage des Fachbereichs: www.dm.fh-furtwangen.de

1 Einleitung

1.1 Ziel und Schwerpunkt der Diplomarbeit

Ziel dieser Arbeit ist es, ein funktionsfähiges System zu entwickeln, das Entwicklern von Computerspielen hilft, Lippensynchronisation in ihren Anwendungen zu implementieren. Das System wird in dieser Arbeit theoretisch erläutert und anschließend praktische Umsetzung in einer kleinen Beispielanwendung finden.

Obwohl das System primär für Computerspiele vorgesehen ist, soll es auch gut auf andere Anwendungen übertragbar sein.

Das zu entwickelnde System soll in echtzeitkritischen Programmen im Zusammenspiel mit anderen Modulen lauffähig sein.

Es sollen einige Möglichkeiten der Lippensynchronisation kurz dargestellt und daraus ein System entwickelt werden, das ein gutes Mittelmaß an individueller Freiheit im gestalterischen Bereich sowie eine gute Effiziens bei der Umsetzung erzielen soll.

Konkret soll das System aus digitalisierten Audio-Sprachdaten die entsprechende Mundstellung des virtuellen Sprechers ermitteln und auf einen 3D-Charakter übertragen. Das Resultat der Lippensynchronisation soll unabhängig vom Sprecher beim Betrachter einen glaubwürdigen Eindruck hinterlassen.

1.2 Was nicht behandelt wird

Die Diplomarbeit umfasst viele unterschiedliche Gebiete, die für sich gesehen äußerst komplex sind. Viele Bereiche werden deshalb nur angerissen und nicht erschöpfend behandelt. Hauptaugenmerk liegt im Zusammenspiel der Teilbereiche.

Beispielsweise wird in dieser Arbeit keine detaillierte Phonetik behandelt werden, obwohl das Thema durchaus ein wichtiger Bestandteil ist. Auch werden mathematische Vorgänge wie beispielsweise die für das System wichtige Fourieranalyse nicht näher betrachtet. Selbstverständlich wird ihre Aufgabe im Gesamtkonzept erklärt, jedoch nicht algorithmisch.

Jeder Teilbereich bietet sicherlich weitere Lösungswege. Diese Arbeit veranschaulicht zunächst das Grundkonzept und beschreitet anschließend einen möglichen Weg. Wenn nötig werde ich gegebenfalls auf andere Möglichkeiten hinweisen ohne jedoch näher darauf einzugehen.

Der praktische Teil der Arbeit beschäftigt sich mit der beispielhaften Umsetzung des Konzeptes. Dieser Bereich ist sehr umfangreich an C++-Quellcode. Tiefgehende enginetechnische Erläuterungen werde ich in der Diplomarbeit nicht tätigen sondern auch hier lediglich auf die Funktionalität und Zusammenhänge eingehen.

1.3 Anwendungsgebiete

Für das zu entwickelnde System gibt es zahlreiche Anwendungsgebiete. Mit leichter Modifikation können sich auch noch weitere Anwendungsmöglichkeiten ergeben.

Der Lipsync-Workflow ist zunächst in Echtzeitanwendungen verwendbar, die direkt aus einem Eingabesignal, z.B. per Mikrofon oder Audiofile, einen Charakter die Lippen bewegen lassen sollen. Hierzu zählen primär Echtzeit-Computerspiele und bei unserer Ausrichtung auf 3D-

Charaktere hauptsächlich 3D-Games. Der modulare Aufbau macht es jedoch mögliche das System leicht verändert in 2D-Games einzusetzen.

In den Echtzeitbereich fallen auch alle Möglichkeiten von 3D-Avataren, die man in verschiedenen Formen zu sehen bekommt. Beispielsweise könnten sprechende interaktive Hilfsagenten mit einem LipSync-System ausgestattet werden. Anwendungsgebiete wären in solchen Fällen das Internet, PC-Anwendungen, Hilfsterminals u.a.. Auch bei einer Voice-Chat- oder Telefonvisualisierung mittels eines Avatars wäre Lippensynchronisation denkbar.

In interaktiven TV-Shows, die beispielsweise über Zuschauertelefonate agieren, ließe sich über eine Lippenvisualisierung mittels Avatar nachdenken.

Das LipSync-System lässt sich auch mit leichten Veränderungen als Plugin z.B. für 3D-Softwareprodukte verwirklichen. So könnte der Workflow in ein aufnehmendes System integriert werden, mit dem sich Computersequenzen erstellen lassen, die dann weiterbearbeitet und beispielsweise in Filmen oder Animationssequenzen Verwendung finden können.

Durch die freien Regeldefinitionen der Ein- und Ausgabemethoden können noch einige weitere Anwendungsgebiete erschlossen werden. Das Audioverarbeitungsmodul kann beispielsweise als selbständiger, konfigurierbarer Audio-Controller mit beliebigen Ausgaberegeln verknüpft werden. Diese Anwendungsgebiete sind sehr spannend und experimentell, haben dann jedoch kaum noch etwas mit dem eigentlichen Ziel dieser Arbeit zu tun und erfordern teilweise umfangreichere Veränderungen der Schnittstellen, so dass an dieser Stelle nicht weiter darauf eingegangen wird.

1.4 Momentaner Entwicklungsstand/Techniken

Schon in älteren 2D-Computerspielen gab es ansatzweise Mundanimationen zu Textdialogen. Da zu dieser Zeit der Speicherplatz jedoch für Audiofiles zu klein war stellte sich die Frage nach synchroner Mundanimationen erst gar nicht.

Seit der Marktfähigkeit der CDRom als Datenträger in den späten 90er Jahren wurden Computerspiele multimedial aufwändiger. Statt einiger weniger Megabytes standen auf einmal einige Hundert Megabytes zur Verfügung. So konnten die bisherigen Textdialoge mit Audiodaten unterlegt oder ersetzt werden.

Der Aufschwung der 3D-Technologie begann mit „Doom“[9] im Jahre 1993. Heutzutage setzen fast alle Spieleentwickler auf Echtzeit-3D-Grafik. Die Freiheiten, die die Manipulation der Rohdaten gegenüber den vorgerenderten 2D-Sprite-Grafiken bietet, sind für die meisten Spieleentwickler erste Wahl. Durch Hardware beschleunigt sind selbst komplexe Berechnungen in Echtzeit möglich. Morphing und Animationsüberblendungen wären in 2D undenkbar oder nur sehr schwer realisierbar gewesen. In 3D hingegen sind sie nun durch Vertexverschiebungen[10] und Meshmanipulation[11] an den zu rendernden Rohdaten verhältnismäßig einfach umzusetzen. Und gerade diese Möglichkeit benötigt man für Facial-Animation bzw. für die Mundbewegungen.

Audiotechnisch gesehen laufen alle Soundausgaben direkt über die Hardware der Soundkarte und belasten die CPU somit fast gar nicht. Es sei denn es handelt sich um komprimierte Audiofiles, die zuvor vom Prozessor decodiert werden müssen.

Heutige Spielecomputer sind in der Regel mit ausreichend schnellem Speicher und guten Prozessoren ausgestattet, die modernen Spieleanforderungen genügen.

9 ID Software, 1993
10 Vertex: Gitternetzpunkt; kleinste Einheit eines 3D-Körpers
11 Mesh: Gitternetz, das ein 3D-Objekt beschreibt

Im Verhältnis zu aufwändigen Wasser- und Physiksimulationen ist ein LipSync-Modul prozessorlastig vernachlässigbar.

Einige Computerspieleentwickler haben deshalb auch schon begonnen Lippenbewegungen in ihre Computerspiele einzubauen. Aufgrund des hohen Aufwandes bei der Erstellung beschränken sich die meisten jedoch auf die Cutscenes[12], wohingegen im Spiel selbst meist keine Mundbewegungen zu sehen sind.

Es gibt verschiedene Techniken für Mundanimationen bzw. für synchrone Mundanimationen. Folgend nun habe ich einige Vorgehensweisen aufgelistet, wie sie in Computerspielen Verwendung finden.

- Mundanimationen werden synchron zum Audiofile voranimiert und im Spiel zusammen abgespielt (siehe Abbildung 1). Dies ist ein sehr aufwändiges und datenintensives Verfahren, wenn man bedenkt, dass moderne Computerspiele teilweise mehrere Stunden an Sprachfiles beinhalten. Andererseits ist es auch das Verfahren, das dem Animator die größten künstlerischen Freiheiten gibt und bei dem man sicher sein kann, dass das Ergebnis so aussieht, wie man es sich erwünscht. Außerdem können neben Mundanimationen auch weitere Facial-Animations eingebaut werden. Diese Technik ist weit verbreitet im Computerspielebereich, und wird häufig für Cutscenes eingesetzt.

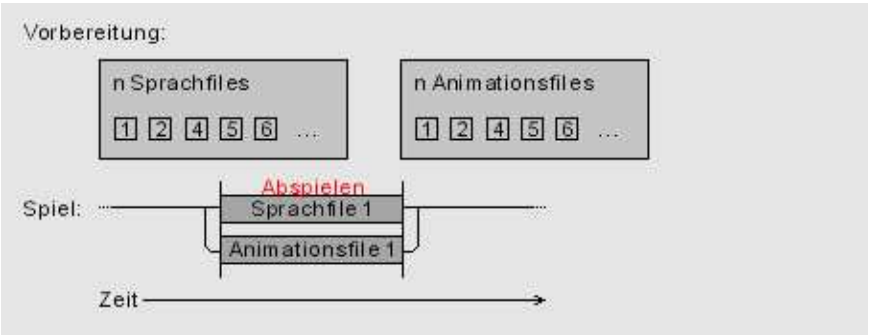

Abbildung 1: Voranimierte Mundbewegungen und synchrones Abspielen

Bei der Generierung der Animationsfiles gibt es Tools, Plugins und andere Möglichkeiten, die den Aufwand reduzieren können. Beispielsweise gibt es Phonem- und Texterkennungsprogramme, die mittels unterschiedlicher Analyseverfahren Animationen aus Text- oder Audiodateien generieren. Ein Beispiel hierfür wäre das Plugin für 3D Studio Max „Voice-O-Matic“ von Digimation[13]. Eine andere Möglichkeit sind Motion-Capturing-Verfahren, die direkt beim Einsprechen der Audiofiles beim Sprecher vorgenommen werden. Aus den daraus gewonnenen Daten können dann die Mund- und Gesichtsanimationen erstellt werden.

- Eine weitere Möglichkeit ist die Erstellung eines generischen Animationspools (siehe Abbildung 2) . Dabei werden verschiedene Mundanimationen erstellt, die möglichst universell einsetzbar sind. Im Spiel wird dann zufällig die eine oder andere Animation zusammen mit dem Audiofile abgespielt. Ist das Audiofile länger als die Animation, so wird danach eine weitere zufällige Animation gestartet. Wenn das Audiofile vor der Animation endet wird die Animation einfach ausgeblendet. Weiterentwickelt können auch einfache Regeln definiert

12 Cutscene: Zwischensequenzen, die die Geschichte des Spiels vorantreiben
13 Homepage von Digimation: www.digimation.com

werden. Auch lautstärkebasierte Abspielsysteme sind möglich.

Dieses System ist nicht lippensynchron und deshalb nur für den visuellen Eindruck geeignet. Dafür ist es einfach zu implementieren und mit relativ geringem Aufwand umsetzbar.

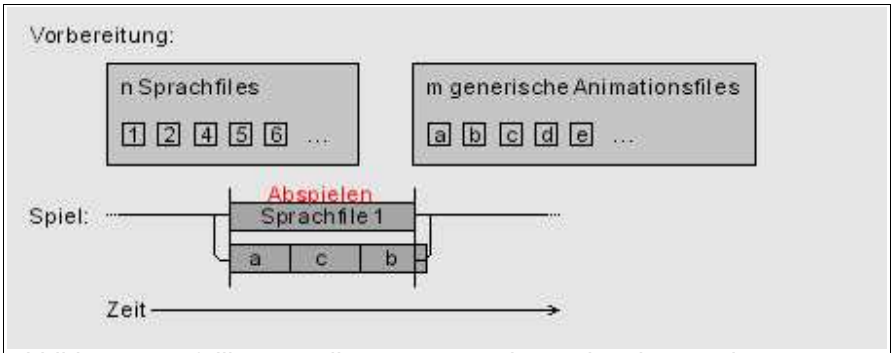

Abbildung 2: Zufällige Mundbewegungen über Animationspool

- Vom Datenaufwand gering und im Prinzip komplett automatisierbar ist das Verfahren, bei dem die Mundanimationen erst im Spiel generiert werden (siehe Abbildung 3). Hierfür müssen zuvor Regeln aufgestellt werden, nach denen die Animationen vor dem Abspielen des Audiofiles erstellt werden. Die Informationen zur Generierung der Animationsdaten werden beispielsweise aus zuvor erstellten Morphdaten und den erhaltenen Informationen aus der Sprachfileanalyse gewonnen. Der Vorteil hierbei ist, dass das komplette Audiofile analysiert werden kann und man deshalb zusätzlich Erkennungsalgorithmen anwenden kann die auf Worterkennung basieren, wie man es von Spracherkennungsprogrammen wie z.B. „ViaVoice“ von IBM[14] her kennt. Problematisch hierbei ist, dass man vorher nicht weiß, wie lange die Generierung der Animationsdaten vor allem bei längeren Audiofiles dauert und wie genau sie aussehen.

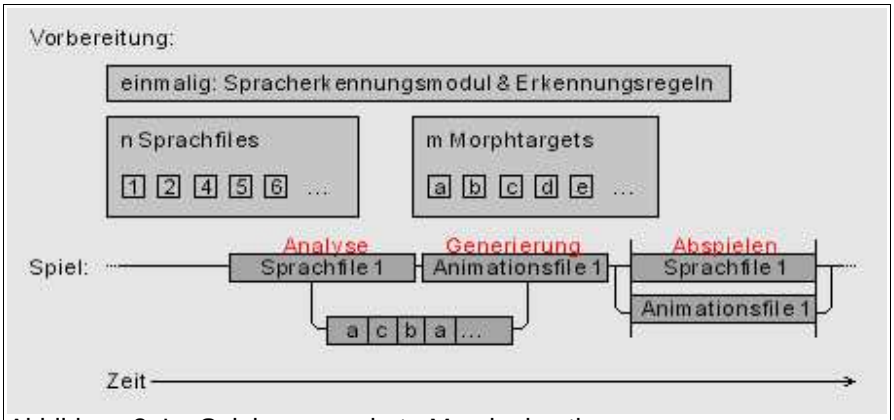

Abbildung 3: Im Spiel vorgenerierte Mundanimationen

- Eine leicht abgeänderte Variante der im Spiel generierten Animationsdaten ist es, die Daten in Echtzeit während der Laufzeit zu erstellen (siehe Abbildung 4). Das Audiofile wird hierbei direkt beim Abspielen stückchenweise analysiert und eine Phonemerkennung durchgeführt.

14 Homepage von IBM: www.ibm.com

Auch bei diesem Verfahren müssen zuvor die Mundpositionen und Verwendungsregeln definiert werden. Nachteilig ist, dass man sich allein auf die Phonemerkennungsalgorithmen stützen muss und dass man ebenfalls kaum eine direkte Kontrolle über das Aussehen des Resultates hat. Vorteilig ist der sehr geringe Daten- und Arbeitsaufwand. Bei einer guten Phonemerkennung ist das sicherlich das am besten geeignete System zur Automatisierung synchroner Lippenbewegungen.

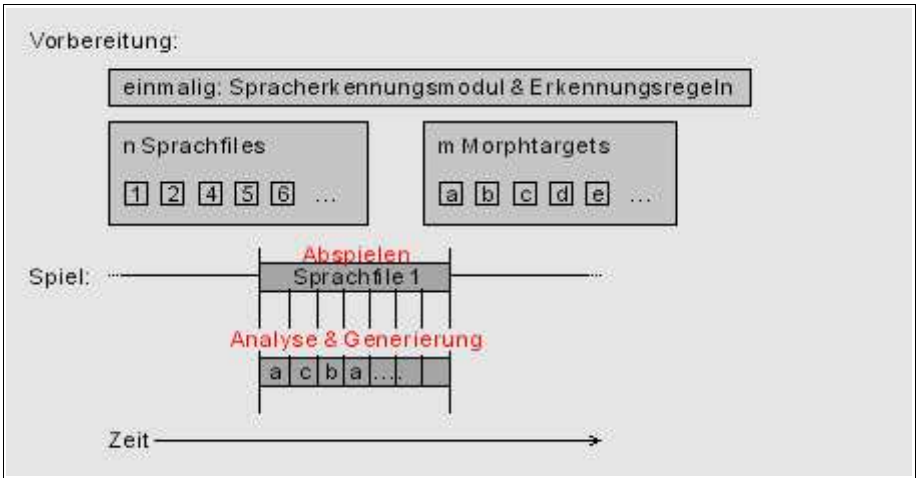

Abbildung 4: Im Spiel in Echtzeit generierte Mundbewegung

1.5 Analyse aktueller Computerspiele

Die Analyse einiger Computerspiele ergab, dass je nach Computerspielgenre mehr oder weniger Wert auf Mundanimation und synchrone Mundbewegungen gelegt wird. Erste Voraussetzung ist, dass das Spiel Echtzeit-3D-Grafik verwendet. Dann kann prinzipiell zwischen Cutscene- und Ingame[15]-Qualität unterschieden werden. Nahezu alle Spiele, die keine vorgerenderten Cutscenes einsetzen, sondern auch hier die eigene Realtime-Engine nutzen, verwenden an die Zwischensequenz angepasste Animationsbewegungen und Mundanimationen. Oftmals sind die Szenen spielfilmreif inszeniert wie in „Mafia“[16]. Dafür gibt es im Spiel selbst keine Mundbewegungen zu sehen.

15 Ingame: im Spiel; zur Laufzeit
16 Illusion Softworks, Take2, 2002

Abbildung 5: Ingame-Screenshot aus „Mafia“

Abbildung 6: Cutscene-Screenshot aus „Mafia“: Einführungsgespräch

Mundbewegungen im Computerspiel sind speziell im Rollenspiel- und Action-Adventure-Genre verbreiteter, da diese sprachlastiger sind wie beispielsweise die meist mit einem Text-Briefing auskommenden Ego-Shooter. Strategiespiele wie beispielsweise „Die Siedler – Die Erben des Königs“[17] verzichten komplett auf Mundbewegungen, da die Figuren meist nur sehr klein auf dem Bildschirm dargestellt werden. Vereinzelt werden Mundanimationen in größeren Darstellungen auf dem Interface verwendet wie bei „Warcraft 3“[18].

Abbildung 7: Ingame-Screenshot aus „Warcraft 3“: Sprechender Ork

Sehr interessant ist das Sprachsystem von „Gothic 2“[19], einem Rollenspiel von Piranha Bytes. Nahezu jedes Sprachfile aller Personen im Spiel besitzen synchronisierte Mundbewegungen. Das lässt vermuten, dass dort ein Automationssystem verwendet wird, das aus den Audiofiles die Animationen generiert. Ob das in Echtzeit im Spiel geschieht oder vorberechnet wird konnte ich leider nicht herausfinden. Vermutlich handelt es sich jedoch um ein ähnliches System wie wir später im praktischen Teil dieser Arbeit entwickeln werden. Zusätzlich zu den Mundbewegungen scheinen auch die Körperbewegungen vom Audiofile gesteuert zu sein.

17 Bluebyte, Ubisoft, 2004
18 Blizzard, Vivendi Universal, 2003
19 Piranha Bytes, Jowood, 2002

Abbildung 8: Ingame-Screenshot aus „Gothic 2“: Dialog mit der Stadtwache

Abbildung 9: Ingame-Screenshot aus „Gothic 2“: Ansprache eines Priesters

„Morrowind“[20], ein Rollenspiel von „Bethesda Softworks“, verwendet ein ähnliches System. Auch hier sind mehrere Stunden Dialoge vertont. Bei allen Sprachsequenzen bewegt der Charakter annähernd synchron seinen Mund. Es wird jedoch vermutlich lautstärke- oder zufallsbasiert nur zwischen zwei Zuständen hin- und hergemorpht.

Abbildung 10: Ingame-Screenshot aus „Morrowind“

Beispielvideos aus „Mafia“, „Warcraft 3“, „Gothic 2“ und „Morrowind“ sind auf der beiliegenden CDRom zu finden.

1.6 Zukunftsorientierter Ausblick

In naher Zukunft wird sich das Computerspiel immer mehr dem Spielfilm annähern. Die Grafiken und die Animationen werden zunehmend realistischer und vielfältiger. Auch die Budgets, mit denen Computerspiele namhafter Hersteller entwickelt werden, steigen enorm an. Cutscenes werden zum Teil nicht mehr vorgerendert sondern können direkt in Echtzeit von leistungsfähigen Engines berechnet werden. Realistische Lichtreflexionen, Wasserbewegungen, Physik- und Kleidungssimulation sind nur einige Begriffe, die zur Zeit höchste Aktualität im Computerspielgenre besitzen.

20 Bethesda Softworks, Ubisoft, 2002

Es wird nur eine Frage der Zeit sein, wann realistische Mundbewegungen zum Standardumfang einer Gameengine gehören. Hierbei wird man aufgrund des Umfanges und Aufwandes nicht auf vorgefertigte Animationen bauen, sondern sicherlich den flexibleren Weg über die Spracherkennung nehmen.

Wie im vorigen Kapitel erwähnt, besitzen heutzutage schon sehr viele Spiele Gesichts- und Mundanimationsfeatures mit unterschiedlichen Ansätzen. Für Close-Ups und Nahaufnahmen reichen diese Systeme jedoch noch nicht aus.

Hier kann man sich sicher sein, dass in Zukunft noch aufwändigere Gesichtsanimationen angestrebt werden, die auch für Nahaufnahmen tauglich sind, und die die Emotionen der Charaktere noch besser vermitteln. Aufgrund des hohen Aufwandes ist es sehr gut möglich, dass in Zukunft die Gesichtsanimationen nicht mehr hauptsächlich per Morphing realisiert werden sondern über das rechenintensivere Bonesystem[21], da sich das Skinning[22] leichter automatisieren lässt. Auch werden Facial-Animations in Trick- und Spielfilmen heutzutage fast ausschließlich mittels Bones realisiert.

In Bezug auf die Phonemerkennung kann man sich gut vorstellen, dass es bald vorgefertigte Module z.B. in Software Development Kits von Microsoft gibt, die einem diese Aufgabe komfortabel abnehmen. Erste Schritte sind diesbezüglich schon mit dem MS SpeechSDK gemacht worden. Beispielsweise könnte im für Spieleentwickler konzipierten DirectX die DirectSound-Komponente um diese Funktionalität erweitert werden. Nicht ganz abwegig erscheint dann auch eine Hardware-Lösung direkt auf Soundkarten zu implementieren.

Das Ausnutzen der Audiodaten könnte in der Zukunft neue Dimensionen erreichen. Zahlreiche Institute suchen an Lösungen, wie Emotionen sprachlich ausgedrückt werden. Wenn man in diesen Gebieten neue Kenntnisse erhält, könnten viele Facial-Animation- und LipSync-Probleme der Vergangenheit angehören. Man könnte revolutionäre Animationscontroller entwickeln, die komplett aus den Audiodaten beispielsweise den Gemütszustand eines Charakters erkennen, und diesen dann dementsprechend aussehen und agieren lassen.

Die Entwicklungen in diesem Bereich sind jung und es wird spannend, was die Zukunft bringen wird.

21 Bonesystem: Knochensystem mit dessen Hilfe ein Charakter animiert werden kann
22 Skinning: Verbindung zwischen Mesh und Bonesystem herstellen

2 Gesamtkonzept-Übersicht

2.1 Technische Vorgehensweise

Mit dem Ziel den Aufwand bei der Erstellung der 3D-Daten möglichst gering zu halten und einen im Verhältnis von Aufwand und Resultat praktikablen Workflow zu erstellen, fällt die Entscheidung zu Gunsten der Umsetzung eines Echtzeit-Analyse-Workflows aus, wie er in Kapitel 1.4 beschrieben wurde.

Die Hauptersparnis liegt hier bei der Erstellung der Mundanimationen. Das bedeutet, dass der Artist keine kompletten Animationssequenzen erstellt, sondern diese vom Programm während der Laufzeit automatisch generiert werden. Der Artist gibt lediglich einige Mundstellungen vor, die vom System dann verwaltet und verwendet werden. Die Umsetzung des Systems erfolgt in der hardwarenahen Programmiersprache C++.

Folgend eine Übersicht über den Ablauf des Systems:

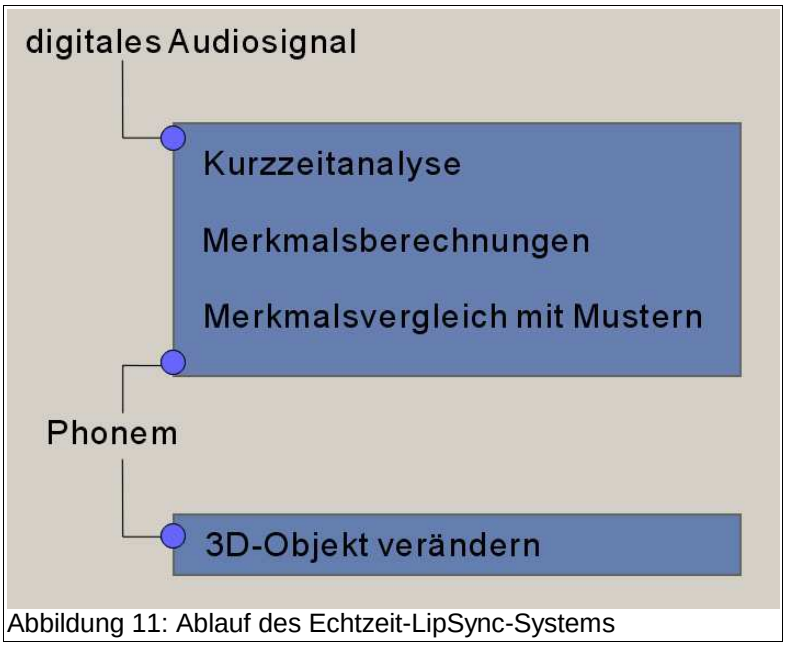

Abbildung 11: Ablauf des Echtzeit-LipSync-Systems

Es gibt demnach verschiedene Bereiche, die für die Umsetzung eines solchen Systems wichtig sind.

Wir benötigen zunächst Hintergrundwissen über den Aufbau eines digitalen Audiosignales, sowie die Besonderheiten von Sprachsignalen. Für die Spracherkennung benötigen wir Grundlagenwissen über allgemeine Phonetik und über Methoden der Audiosignalbearbeitung und Filterung.

Für die 3D-Komponenten sind genaue Kenntnisse über den Aufbau eines 3D-Objektes und die Methoden zur Veränderung eines 3D-Objektes nötig.

Im Bereich der praktischen Umsetzung benötigen wir Kenntnisse im Umgang mit der Programmiersprache C++ sowie den verwendeten Libraries Direct3D und DirectSound.

Da wir für unser Beispielprogramm auch die Arbeiten des Artists übernehmen werden, sind Kenntnisse im Umgang mit einem 3D-Programm wichtig.

Um eine gute Schnittstelle zwischen 3D-Programm und Beispielanwendung herstellen zu können sind Kenntnisse über eventuell unterstützte Pluginschnittstellen oder Scriptingmöglichkeiten der

3D-Software wichtig.

2.2 Voraussetzungen

Da das System für zukünftige und aktuelle Computerspiele und Anwendungen konzipiert ist, wird dementsprechend auch aktuelle Hardware des PC's[23] benötigt.

Eine wichtige Rolle spielen hier vor allem eine gute Grafikkarte mit DirectX-Hardwareunterstützung, möglichst viel schnellem Grafikkartenspeicher und einem schnellen Bus, wie beispielsweise einen 4x oder 8xAGP-Bus für eine schnelle Grafikdatenübertragung.

Zum Abspielen der Audiodaten benötigen wir am besten eine Soundblaster-kompatible Soundkarte.

Die Phonemanalyse und die Objektberechnung übernimmt der Prozessor. Die Daten zur Berechnung liegen in der Regel im RAM[24]-Speicher. Aus diesem Grund ist es gut, wenn man eine CPU[25] mit schneller Taktung besitzt und möglichst viel schnellen RAM-Speicher, wie beispielsweise DDRAM[26].

Softwaretechnisch ist das Beispielprogramm aufgrund des Nutzens von DirectX9 an MS WindowsXP[27] gebunden.

Aktuelle PC-Systeme sind in der Regel schnell genug und besitzen die nötige Grundausstattung für das hier entwickelte System.

2.3 Kurzbeschreibung der Workflow-Module

Wie aus Abbildung 11 schon zu entnehmen ist sind die wichtigsten Module des Systems ein Modul, das die Phonembearbeitung übernimmt und ein Modul, das die Objektbearbeitung ausführt. An das Phonembearbeitungsmodul wird ein kurzes Audiosignal geliefert, das dann einer Phonemanalyse unterzogen wird. Das übergebene Audiosignal beinhaltet die Audiodaten des zuletzt oder momentan abgespielten Audiosignals. Die Größe des Signals liegt idealerweise aufgrund der Echtzeitreaktionszeit bei etwa 10-40ms. Die Phonemanalyse gibt eine ID zurück, anhand jener der Programmierer das erkannte Phonem erhält. In der Phonemanalyse muss eine Mustererkennung mit Phonem-Mustervorlagen stattfinden. Das bedeutet, dass zuvor Muster für die zu erkennenden Phoneme und Merkmalserkennungsregeln definiert sein müssen.

An das Objektbearbeitungsmodul wird das Phonem übergeben. Anhand des Phonems kann das Modul die vorzunehmenden Veränderungen berechnen und weiß, wie es das Objekt verändern muss. Die Veränderungsregeln müssen zuvor definiert worden sein. Dies ist die verbleibende Aufgabe für den Artist im Artist-Workflow. (siehe Kapitel 5.4)

Die Aufgaben des Artist lassen sich wie folgt zusammenfassen:

- Objekt erstellen
- Veränderungsregeln je definiertem Phonem erstellen

23 PC: Personal Computer
24 RAM: Random Access Memory
25 CPU: Central Processing Unit
26 DDRAM: Double Data Random Access Memory
27 Microsoft Windows: Betriebssystem der Firma Microsoft http://www.microsoft.com

- Objekt exportieren
- Veränderungsregeln exportieren

Die Veränderungsregeln bestehen aus den unterschiedlichen Mundstellungen der Phoneme. Der Artist braucht dementsprechend keine Mundanimationen mehr zu erstellen sondern definiert nur die möglichen Mundzustände. Dies bringt eine enorme Zeitersparnis. Der Artist benötigt bei der Erstellung lediglich ein 3D-Programm, einen Exporter für die Objektdaten und einen Exporter für die Mundstellungen.

Die Aufgaben des Programmierers werden im Verhältnis zur Reduzierung der Arbeit des Artist nicht wesentlich erhöht. Er muss einmal die Funktionalität des Systems einbauen und in der weiteren Arbeit gibt es auch für ihn keinen Mehraufwand. Als Vorbereitungsaufwand sind hier zunächst folgende zu erwähnen:

- Phonemanalyse implementieren
- Phonemerkennungsregeln definieren
- Veränderungsregeln verwalten
- Objektbearbeitung anhand der Phonemdaten implementieren

Sind diese Implementierungen vorgenommen ist der einzige Unterschied, dass der Programmierer statt des Abspielens einer Mundanimationsdatei die Phonemanalyse startet und das Ergebnis an das Objektbearbeitungsmodul weiterreicht.

3 Audiosignal-Analyse

3.1 Grundlagen der Spracherkennung

Die Spracherkennungssysteme haben seit dem Beginn im Jahre 1984 in den letzten Jahren große Fortschritte gemacht. Automatische Diktierprogramme von IBM oder Philips gehören heutzutage schon zur Standardsoftware-Palette.

Trotzdem ist es Computersystemen nur möglich Sprache nach vorgegebenen Mustern zu erkennen bzw. zu kategorisieren. Wo der Mensch im Gesamtkontext einen gewissen interpretatorischen Freiraum hat, folgt die computergestützte Spracherkennung immer denselben vorgegebenen Regeln. Durch selbstlernende Systeme wie Neuronale Netze wird versucht das menschliche Lernverhalten nachzuahmen und dadurch im Laufe der Zeit die Fehlerquote bei der Erkennung zu minimieren.

Der Kernbereich der Spracherkennung konzentriert sich auf die Worterkennung. Aus den eingesprochenen Audiodaten werden anhand vorgegebener Muster aus einem zuvor definierten Lexikon die Worte zugeordnet. Sie dient hauptsächlich der Texterfassung und Sprachsteuerung. Spracherkennung findet heutzutage beispielsweise beim Telefon-Banking oder bei computergesteuerten Telefon-Auskünften statt.

Man unterscheidet bei der Spracherkennung grundsätzlich zwischen zwei Systemen. Zunächst die diskrete Spracherkennung. Bei ihr muss jedes zu erkennende Wort einzeln gesprochen werden. Zwischen den Worten müssen deutliche vom Erkennungssystem als Stille wahrnehmbare Pausen erfolgen. Dafür ist es für das Erkennungsprogramm wesentlich einfacher die Worte auseinanderzuhalten weshalb die Fehlerquote dementsprechend drastisch reduziert wird. Unter dem Begriff der kontinuierlichen Spracherkennung versteht man Spracherkennungssysteme, bei denen der Benutzer den natürlichen kontinuierlichen Sprachfluss beibehalten kann. Das Erkennungssystem versucht selbst die einzelnen Worte aus dem Sprachfluss zu separieren. Der Erkennungsvorgang ist bei der kontinuierlichen Spracherkennung um einiges komplexer und für den Computer ist es wesentlich schwieriger einzelne Worte zu separieren und korrekt zuzuordnen.

Eine Sprecherunabhängigkeit der Systeme wird meist durch intensives Training erreicht bzw. durch eine große Datenbank an Vergleichsmustern.

Da die Worterkennungssysteme jedoch nicht echtzeitfähig für Lippensysnchronisation sind, müssen noch kleinere Elemente des Wortes analysiert werden. Die kleinste Einheit der Sprache ist das Phonem. Laut Harry R. Ihm[28] beträgt die Dauer eines Phonems ca. 10-40ms.

Die Phonetik ist ein sehr weitreichendes Forschungsfeld. Fakultäten versuchen Phoneme zu kategorisieren (siehe Abbildung12[29]), spezielle Merkmale zuzuordnen und sie so für die computergestützte Spracherkennung etwas greifbarer zu machen.

28 Harry R. Ihm: "Das grosse Spracherkennungsbuch" (1999: Linguatec Sprachtechnologien)
29 Abbildung aus http://medi.uni-oldenburg.de/download/docs/lehre/kollm_phystechmed_akustik/aku4.pdf (04.01.2005)

	vokalisch	konsonantisch	kompakt	dunkel	nasal	abrupt	gespannt	stimmhaft	scharf
b (Bad)	-	+	-	+	-	+	-	+	-
d (du)	-	+	-	-	-	+	-	+	-
f (Fee)	-	+	-	+	-	-	+	-	+
g (gut)	-	+	+	+	-	+	-	+	-
h (Haar)	-	-	+	+	-	-	+	-	-
k (Kai)	-	+	+	+	-	+	+	-	-
l (lag)	+	+	-	-	-	-	-	+	-
m (Mal)	-	+	-	+	+	+	-	+	-
n (nun)	-	+	-	-	+	+	-	+	-
p (Pein)	-	+	-	+	-	+	+	-	-
r (raus)	+	+	-	-	-	+	-	+	-
s (das)	-	+	-	-	-	-	+	-	+
ʃ (Scheu)	-	+	+	-	-	-	+	-	+
t (Tal)	-	+	-	+	-	-	-	+	-
v (Vase)	-	+	-	+	-	-	-	+	-
x (Dach)	-	+	+	+	-	-	-	-	-
z (Sinn)	-	+	-	-	-	-	-	+	+
j (Jod)	-	+	+	-	-	-	-	+	-

Abbildung 12: phonetische Katalogisierung der deutschen Konsonanten

Um eine Spracherkennung in Echtzeit vorzunehmen bedient man sich sogenannter Kurzzeitspektren. Das Eingangssignal wird dabei unabhängig von der Länge der Worte in kurzen Millisekunden-Abschnitten analysiert und anhand von Merkmalsvektoren einem Phonem zugeordnet.

Um diese Merkmalsvektoren zu erhalten gibt es unterschiedliche Verfahren:

- einfache Mekmalsmodellierung: Nach unterschiedlichen Analyseschritten bzw. vor und nach der Spektralzerlegung des Audiosignales werden spezielle Merkmale ermittelt wie beispielsweise die Energie des Signals, das Vorhandensein spezieller Frequenzen etc. Anhand dieser Merkmale wird dann durch Vergleich mit den Referenzmustern das Phonem zugeordnet.

- Mel-Cepstrum-Koeffizienten: Hierbei werden anhand weiterer Cosinustransformationen zusätzliche Merkmale gewonnen, die unabhängig vom Sprecher und der Lautstärke sind.

- Lineare Prädiktionsanalyse: Aus dem Eingangssignal werden über Approximation und Fehlerminimierung zwei Merkmalsvektoren, die sogenannten Prädiktionskoeffizienten, gewonnen.

Bei der Merkmalsfindung können schon gewonnene Erkenntnisse zu Rate gezogen werden. Beispielsweise gibt es Formanttabellen (siehe Abbildung13[30]), die zeigen, bei welchem Vokal welche Frequenz immer vorhanden ist. Formanten sind Frequenzbänder, die bei bestimmten Lauten unabhängig von der Grundfrequenz immer mitschwingen.

30 Abbildung von www.sengpielaudio.com (04.01.2005)

Formant-Bereiche deutscher Vokale:

Vokal	Hauptbereich um ca.	Nebenbereich um ca.
U	320 Hz	(800 Hz)
O	500 Hz	(1000 Hz)
å	700 Hz	(1150 Hz)
A	1000 Hz	(1400 Hz)
ö	1500 Hz	500 Hz
ü	1650 Hz	320 Hz
ä	1800 Hz	700 Hz
E	2300 Hz	500 Hz
I	3200 Hz	320 Hz

Abbildung 13: Formanttabelle

Auch gibt es Forschungen, die versuchen Phoneme direkt der Mund-, Zungen- und Kehlkopfstellung zuzuordnen. Abbildung14[31] zeigt das artikulatorische Vokaldreieck, an dem die Formanten F1 analog zur Mundöffnung und der Formant F2 analog zur Unterkieferstellung zu sehen ist.

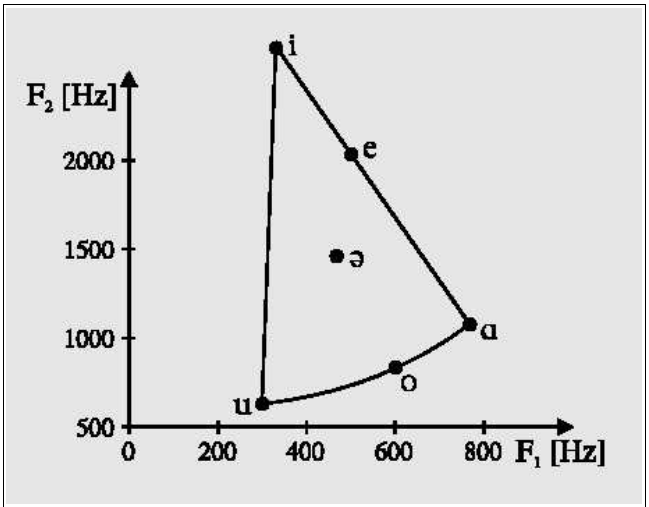

Abbildung 14: Artikulatorisches Vokaldreieck

3.2 Spracherkennung in Echtzeitsystemen

Für Echtzeitsysteme gilt es einige Einschränkungen und Bedingungen zu beachten. Die Antwortzeit von in Echtzeit agierenden Systemen ist nicht genau definiert. Allgemein spricht man jedoch von Echtzeitsystemen, wenn die Antwort fast ohne Zeitverzögerung erfolgt. In unserem Fall würde das bedeuten, dass die Mundbewegungen quasi ohne zeitliche Latenz zum Audiosignal erfolgen muss. Das heißt, dass eine Spracherkennung mittels Wortanalyse nicht in Frage kommen kann. Aus diesem Grund werden in Echtzeitsystemen immer kleine Segmente des Eingangssignals analysiert, beispielsweise in 20-40ms-Segmenten.

31 Abbildung aus http://medi.uni-oldenburg.de/download/docs/lehre/kollm_phystechmed_akustik/aku4.pdf (04.01.2005)

Um den Rechenaufwand zu minimieren kann die Anzahl der zu erkennenden Phoneme auf die wirklich notwendigen reduziert werden. Die Notwendigkeit ermittelt sich durch Analyse der Mundstellungen der einzelnen Phoneme. Beispielsweise sind die Mundstellungen von /o/ und /u/ annähernd ähnlich und können deshalb zusammengefasst werden. Das vermindert den Datenaufwand, spart Speicherplatz und zusätzlich den Arbeitsaufwand in der Produktionsphase. Weniger Erkennungsegeln reduzieren die Prozessorbelastung des Spracherkennungsmoduls der Applikation.

In Minimalanforderung können die Mundstellungen in Echtzeitsystemen auf folgende reduziert werden:

1. offener Mund: /a/, /ae/...
2. breiter Mund: /e/, /i/...
3. runder spitzer Mund: /o/, /u/...
4. geschlossener Mund: /m/, /p/,...
5. leicht geöffneter Mund: /s/, /r/, /l/,...

Dementsprechend können sich die Regeln bei der Phonemerkennung auf die oben genannten konzentrieren.

Der Auflösung sind dementsprechend fast keine Grenzen gesetzt. Hochauflösende Systeme verwenden heutzutage bis etwa 40 verschiedene Mundstellungen.

Vorteilhaft ist es, möglichst universelle Regeln zu definieren, die unabhängig von der Tonhöhe des Sprechers sind. Ansonsten müsste man vor der eigentlichen Analyse anhand der Grundfrequenz des Eingangssignales ermitteln, ob es sich verallgemeinert um die Sprache eines Kindes, einer Frau oder eines Mannes handelt und dementsprechend unterschiedliche Analyseregeln anwenden.

Da das Phonemerkennungsmodul normalerweise ein Phonem als Wert zurückgibt, müssen weiche Animationsübergänge vom Animationscontroller übernommen werden.

In Echtzeitanwendungen werden individuelle Einstellungsmöglichkeiten immer wichtiger. Je nach Leistungsfähigkeit des Systems sollten Details und Berechnungen vereinfacht oder abgeschaltet werden können. Diese Möglichkeit ist bei Echtzeit-Sprachanalyse-Systemen gegeben. Man könnte beispielsweise bei einer Maximum-Details-Einstellung eine volle Sprachanalyse durchführen und alle vorgegebenen Phoneme einbeziehen und verrechnen. Für Medium-Details verwendet man nur noch ein lautstärkebasiertes System und bei Minimum-Details könnte man die Spracherkennung auf nur noch zwei Mundzustände reduzieren.

Es gibt einige für Echtzeitanwendungen vorgesehene kommerzielle Spracherkennungs- bzw LipSync-Libraries. Zwei Beispiele wären „IMS V2A Realtime MDK“ von IMS3D[32] oder „Realtime LipSync-SDK“ von Annosoft[33].

3.3 Das Sonagramm

Das Sonagramm stellt ein Eingangssignal, wie z.B. ein Sprachsignal, in einem Zeit-Frequenz-Intensitäts-Diagramm dar.

Mit Hilfe der Sonagrammdarstellung können Merkmale für bestimmte Laute erkannt werden oder

32 IMS 3D-Homepage: www.ims3d.com
33 Annosoft-Homepage: www.annosoft.com

umgekehrt auch Laute anhand ihrer spezifischen Merkmale bestimmt werden.

In einem Sonagramm werden vorhandene Frequenzen auf einer Zeitachse kenntlich gemacht. Die Y-Koordinatenachse beschreibt die Frequenz in Hz und die X-Achse die Zeit in ms. Die Intensität der Frequenz wird über die Helligkeit codiert. Die drei akustischen Dimensionen sind Zeit-Frequenz-Intensität.

Das Sonagramm in Abbildung 15 zeigt verschieden gesprochene Laute und ihre Frequenzzerlegung. Alle 10ms werden die vorhandenen Frequenzen des Eingangssignales über eine Fouriertransformation ermittelt und dargestellt. Die Helligkeit des Pixels gibt die Intensität der Frequenzamplitude an. Im unterem Beispiel sind die Amplituden vor der Ausgabe normalisiert, so dass die lauteste Frequenz den Helligkeitswert RGB(255,255,255) aufweist. Frequenzen, die den Helligkeitswert RGB(0,0,0) aufweisen sind nicht vorhanden oder werden von dem im Beispiel vorgeschalteten Denoise-Filter entfernt.

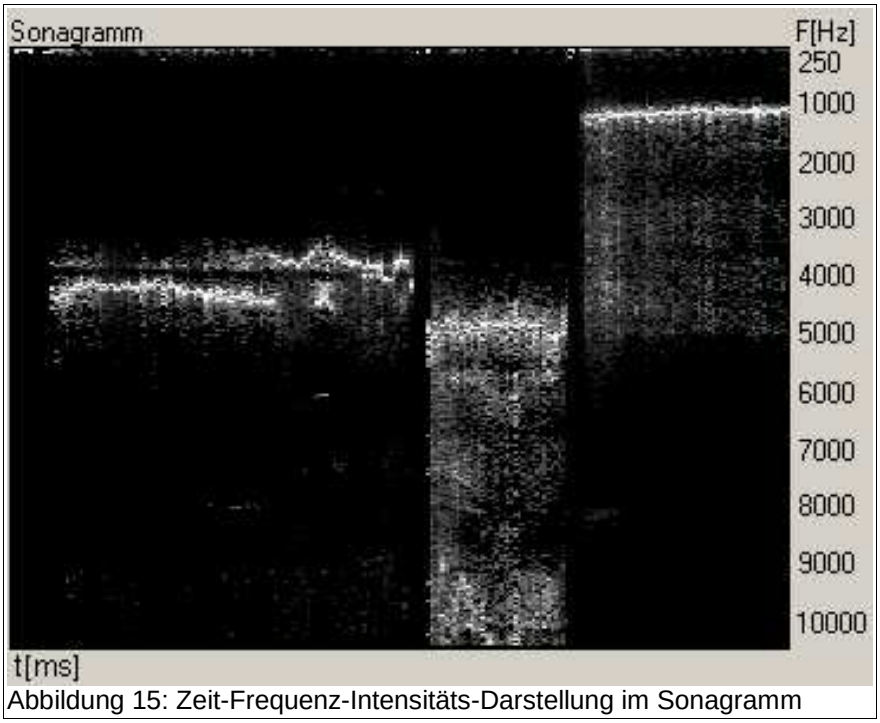

Abbildung 15: Zeit-Frequenz-Intensitäts-Darstellung im Sonagramm

Anhand der Darstellung können geübte Sonagramm-Leser einzelne Laute abgrenzen und bestimmen, ob es sich dabei um Plosive, Nasale, Vokale oder Frikative handelt. Außerdem können bei hochauflösenden Sonagrammen die Harmonischen abgelesen werden und die Grundfrequenz ermittelt werden. Betrachtet man die Frequenzentwicklung über eine größere Zeitspanne lassen sich waagerechte Frequenzbänder erkennen, die fachsprachlich auch als Formanten bezeichnet werden.

Aus einem Sonagramm lassen sich so noch weitere sehr interessante Informationen über das Eingangssignal und insbesondere auch das Sprachverhalten gewinnen. Dies geschieht jedoch meist nur im Nachhinein und im Zusammenhang mit einem größeren Zeitfenster, so dass diese Analyseverfahren meist für die Verwendung in Echtzeit ungeeignet sind.

Für uns dient das Sonagramm in erster Linie dazu, Regeln für die Erkennung von Lauten zu finden und zu definieren, wie wir sie für die Phonemanalyse im praktischen Teil benötigen. Dabei hilft uns das im praktischen Teil entwickelte Programm „PhonemeAnalyser“, das uns unter anderem eine

Echtzeit-Sonagrammdarstellung des Inputsignals zur Verfügung stellt. Auf die Funktionen des Programms wird in Kapitel 5.3.1.3 genauer eingegangen.

Weiterführende Informationen zum Lesen von Sonagrammen sind am Ende der Arbeit unter den Literaturhinweisen und Internetlinks zu finden.

3.4 Lautunterscheidung und Erkennungsmerkmale

Je nach Phonem werden unterschiedliche Frequenzen verschieden stark in Schwingung versetzt. Über ein Sonagramm kann man diese Unterschiede im Frequenzbereich der verschiedenen Laute und Phoneme erkennen.

Abbildung 16 zeigt ein Sonagramm aus dem „PhonemeAnalyser“ bei eingesprochenem /a/, /e/, /i/, /o/ und /u/.

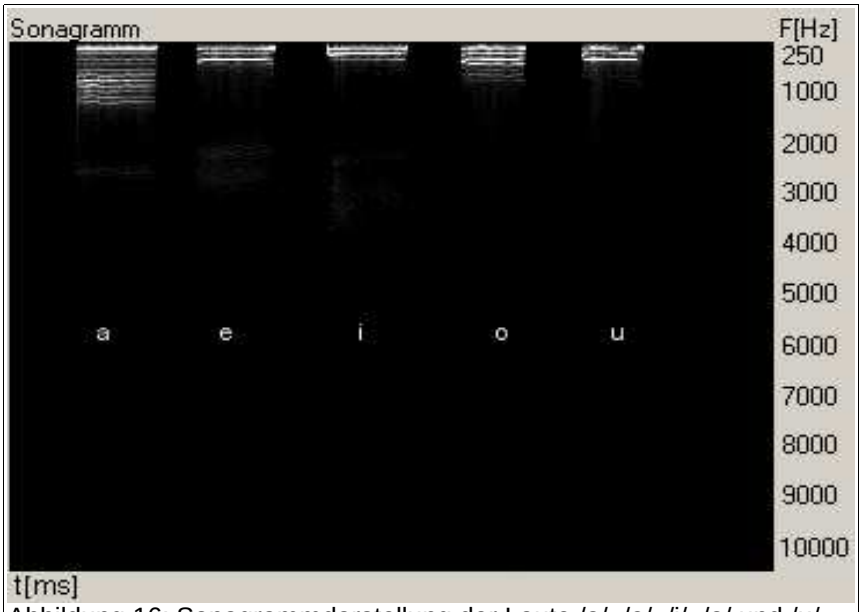

Abbildung 16: Sonagrammdarstellung der Laute /a/, /e/, /i/, /o/ und /u/

Die Laute wurden normal deutlich, in einem von Störgeräuschen weitgehend abgeschirmten Raum aufgenommen. Die Unterschiede in den einzelnen Frequenzbereichen sind dabei deutlich zu erkennen. Die Intensitäten wurden vor der Darstellung normalisiert, so dass die intensivste Frequenz durch weiße Frequenzbänder dargestellt werden.

Es fällt auf, dass die größten Unterschiede in den Frequenzen um 250-2500Hz auszumachen sind. Ein /a/ kann durch das Vorhandensein von Frequenzen im Bereich von ca 1200 Hz eindeutig von einem /o/ unterschieden werden. Es ist jedoch auch zu erkennen, dass ein /e/ nur anhand der Frequenzen schwierig von einem /u/ zu unterscheiden ist. Anhand der Sonagramme wird jedem Phonem ein Muster zugeordnet, das später bei der Phonemerkennung zur Identifizierung beitragen soll.

Die folgende Abbildung 17 zeigt den Frequenzbereich einiger gesprochener Zischlaute. Hierbei fällt vor allem das hohe Vorkommen von Mitten- und Höhenfrequenzen auf.

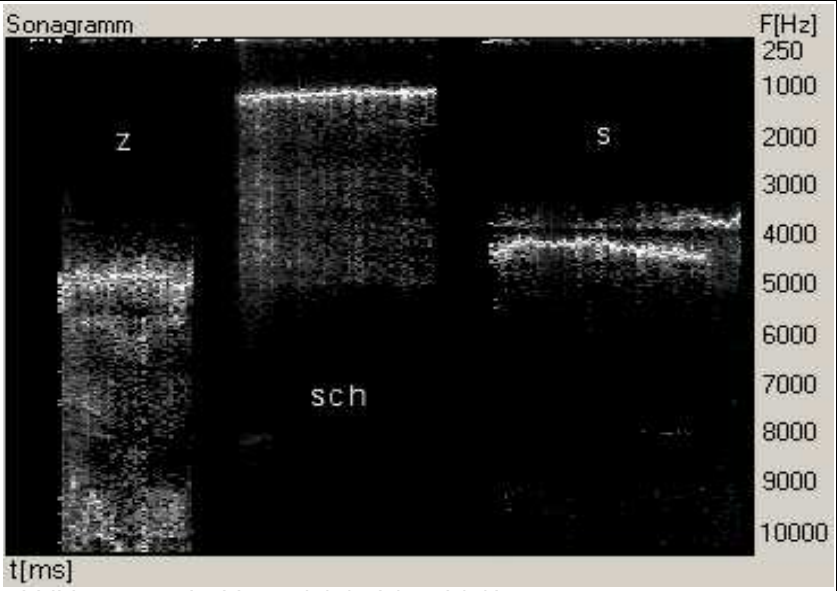

Abbildung 17: Zischlaute /z/, /sch/ und /s/ im Sonagramm

Im Gegensatz dazu besitzen Laute wie /m/ und /n/, die bei geschlossenem Mund artikuliert werden, eher tieffrequente Töne (siehe Abbildung 18).

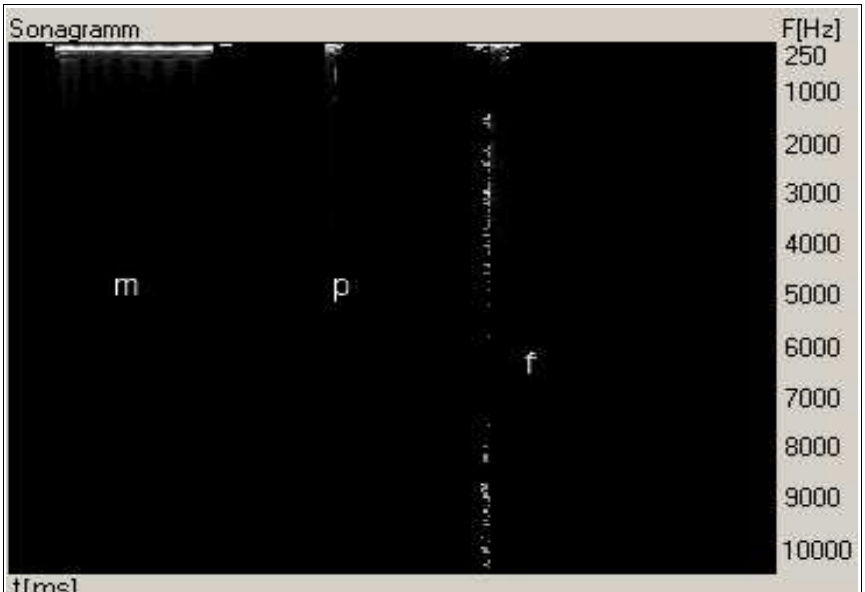

Abbildung 18: Nasal /m/, Poplaut /p/ und Frikativ /f/

Plosive wie der Poplaut /p/ bestehen eigentlich nur aus sehr wenigen sehr tiefen Frequenzen. Frikative wie das /f/ nutzen quasi das komplette dargestellte Frequenzspektrum.

Aus den hieraus gewonnenen Erkenntnissen können rudimentäre Muster zur Erkennung der Phoneme erstellt werden. Diese können dann im Nachhinein durch weitere Einsprechvorgänge verfeinert werden.

Außerdem können zur Unterscheidung noch weitere Merkmale hinzugezogen werden. Zur Erkennung der Stille bietet sich beispielsweise das Merkmal der kompletten Energie an, bei dem alle Amplitudenwerte vor der Normalisierung des Signals zusammengerechnet werden. Eventuell kann auch die Anzahl der Nulldurchgänge des untransformierten Audiosignals interessant sein oder die maximale Lautstärke.

3.5 Zusammenfassung Audiodaten-Verarbeitungspipeline

Ziel der Audiodaten-Verarbeitungspipeline (siehe Abbildung 19) ist es, aus einem Eingangssignal in Echtzeit so zuverlässig wie möglich das gerade gesprochene Phonem zu erkennen. Sie enthält alle Arbeitsschritte, die dafür notwendig sind. Je genauer die Audiodaten-Verarbeitungspipeline funktioniert, desto genauer ist auch das Ergebnis. Sie ist deshalb mit das wichtigste Qualitätskriterium des LipSync-Systems.

Zur Verarbeitung der Audiodaten benötigen wir diese ausschließlich in digitalisierter Form. Alle Verarbeitungsschritte geschehen digital.

Als Eingangssignal kann z.B. ein Audiofile dienen oder die über Mikrofon digitalisierten Audiodaten eines Sprechers.

Generell sollten die digitalisierten Audiodaten „sauber“ sein. Das bedeutet, dass sie möglichst knack- und rauschfrei sowie frei von Nebengeräuschen sein müssen. Ansonsten führt dies zu falschen Ergebnissen bei der Phonemanalyse. Je höher die Samplerate und die Bit-Auflösung, desto genauer, jedoch auch aufwendiger, ist die Fouriertransformation. Die heutzutage üblichen Standardwerte für Audio von 44.1 kHz bei 16 Bit sind für unsere Ziele schon überdimensioniert. Da der Frequenzbereich bis 10kHz für die Unterscheidung der Phoneme ausreicht, benötigen wir demzufolge ein Frequenzspektrum bis ca. 10kHz. Auch mit reduzierter Datenmenge von 22 kHz, wie sie z.B. häufig in Computerspielen verwendet wird, lässt sich demnach gut arbeiten. Das zu analysierende Signal sollte Mono bzw. einkanalig sein.

Das digitale Signal wird nun in kleinen Segmenten mittels einer FastFourier-Transformation spektral zerlegt. Als Ergebnis bekommen wir ein float-Array mit 512 Werten. Das Array deckt den halben analysierten Frequenzbereich ab. Bei 22 kHz Eingangssignal ist die Bandbreite der menschlichen Sprache, die bis ca. 10 kHz reicht, bis auf einige Obertöne noch vollständig vorhanden. Im Frequenzarray werden die Frequenzamplituden als float-Wert gespeichert.

Das Frequenzarray wird nun durch Filter noch etwas aufbereitet. Zunächst verwenden wir einen Denoise-Filter, der alle Frequenzen unterhalb eines Thresholds herausfiltert. Eventuell können noch weitere Filterungen wie z.B. Hoch- oder Tiefpassfilter zur Signalaufbereitung verwendet werden. Im zweiten Schritt wird die Lautstärke normalisiert, um die Mustererkennung der Phonemerkennung zu erleichtern. Hierbei wird die lauteste Frequenz ermittelt, diese auf den Maximalwert 1.0 angehoben und die restlichen Frequenzwerte entsprechend verstärkt. Als Ergebnis erhalten wir das gefilterte Frequenzarray.

Das gefilterte Frequenzarray dient uns nun zur Phonemerkennung. Die zuvor ermittelten und definierten Regeln für die zu erkennenden Phoneme liegen vor und es wird nun frequenzweise das Phonemarray mit den Phonemregeln verglichen. Das wahrscheinlichste Phonem ist dann schließlich das Ergebnis.

Außerdem können zwischen den einzelnen Schritten weitere Merkmale extrahiert werden, wie beispielsweise die Anzahl der Nulldurchgänge beim Eingangssignal oder die unnormalisierte Maximalenergie des Spektrums.

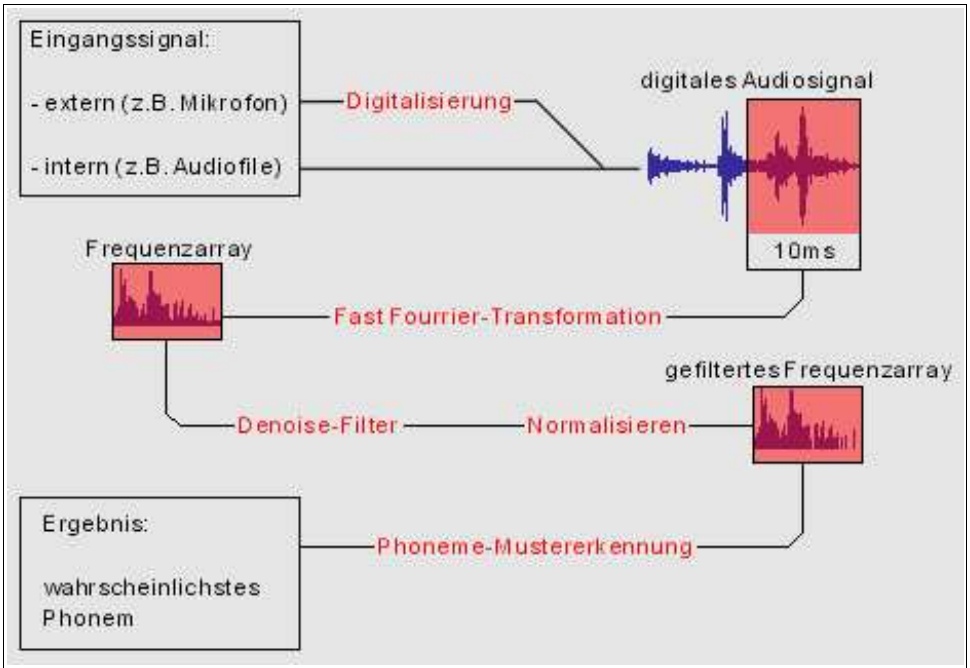

Abbildung 19: Diagramm der Audiodaten-Verarbeitungspipeline

3.6 Verbesserungsmöglichkeiten

Dieses Kapitel enthält Verbesserungsmöglichkeiten für das hier dargestellte Spracherkennungssystem, die jedoch in dieser Arbeit keine praktische Umsetzung finden. Vielmehr sollen es Anregungen für zukünftige Verbesserungen sein.

Die Qualität des Systems steigt mit der Genauigkeit und Zuverlässigkeit des erkannten Phonems. Aus diesem Grund müssen alle Verbesserungsmöglichkeiten auf dieses Ziel ausgerichtet sein. Dabei darf natürlich der Anspruch einer Echtzeitapplikation nicht vernachlässigt werden.

Erster Ansatzpunkt für Verbesserungen sind die Regeln der Mustererkennung. Je genauer die Erkennungsmerkmale definiert sind, desto höher ist auch die Trefferquote. Dasselbe gilt auch für die Anzahl der vorhandenen Vergleichsmuster. Je mehr Muster vorhanden sind, desto genauer kann ein Phonem bestimmt werden. Hierfür ist eine Merkmalsdatenbank, die durch Training von unterschiedlichen Testpersonen gefüllt wird, denkbar.

Eine zusätzliche Aufbereitung des Eingangssignals zur Verbesserung der Erkennung wäre möglich. Beispielsweise soll die Verwendung eines Preemphasize-Filters oder die Filterung des Spektrums anhand einer Mel-Filterdatenbank die Merkmale deutlicher hervorheben.

Verbesserungen sind auch bei der FFT[34] denkbar. Je genauer die FFT arbeitet, umso differenzierter lassen sich die einzelnen Frequenzen analysieren. In einem Frequenzarray, das beispielsweise statt 512 Frequenzen 1024 oder noch mehr enthält lässt sich eine detailliertere Analyse des Signals durchführen.

Eine größere Erkennungsdifferenzierung würde zu einer gesteigerten Trefferquote führen, da bisher vom System unbekannte Laute nicht sicher zugeordnet werden können. So könnte man beispielsweise in vereinfachten Systemen O und U zusammenfassen und in genaueren Systemen O

34 FFT: Fast Fourier Transformation

und U voneinander unterscheiden.

Über das Erstellen von Regelbäumen könnte einiges an Performance gewonnen werden. Beispielsweise könnten über Ausschlussverfahren schon bei der Analyse der ersten Frequenzen bestimmte Phoneme nicht mehr in Frage kommen. Die Verwendung von Bäumen ist vor allem in der Spielebranche bei performancelastigen Aufgaben wie Culling[35] sehr hilfreich.

Man könnte auch überlegen die Ergebnisse statistisch zu beeinflussen und so eine höhere Genauigkeit zu erhalten. Beispielsweise kommt in der deutschen Sprache das /e/ häufiger vor als das /o/[36] oder nach einem /a/ folgt zu höherer Wahrscheinlichkeit ein /i/.

Im direkten Bezug zu Mundbewegungen und Mundsynchronisation ließe sich noch genauer analysieren, ob nicht ein direkter Zusammenhang zwischen Mundform und den Frequenzen besteht. Wenn dies der Fall ist, so ließe sich der Zwischenschritt über die Phonemerkennung komplett übergehen. Allerdings ist es aufgrund der Komplexität des Sprechapparates fraglich, ob sich dieser Schritt so vereinfachen lässt. Jedoch gibt es Forschungsberichte in diesem Bereich und eine genauere Untersuchung wäre sicherlich lohnenswert.

35 Culling: Begriff aus dem 3D-Bereich. Vom Benutzer nicht sichtbare Geometrien werden erst gar nicht gezeichnet
36 http://129.247.105.233/rpleger/ebuss.cgi/EbussWikipedia2.htm (04.01.2005)

4 LipSync-Analyse

4.1 Grundlagen der Mundbewegungen

Lippenbewegungen begegnen uns täglich in Dialogen mit Mitmenschen oder beim Betrachten von TV. Die Sprache und der Drang zu kommunizieren sensibilisiert uns schon in frühen Jahren auf Sprache und die damit verbundenen Mund- und Lippenbewegungen. Mundbewegungen haben einen großen Einfluss auf die Gesichtsmimik.

Die Wichtigkeit der Lippenbewegungen für Gehörlose, die teilweise nur aus den Mundbewegungen den Kontext des Gesprochenen ermitteln können, lässt auf die hohe Bedeutung der Lippenbewegungen für die Kommunikation und somit auch für die Glaubhaftigkeit und den damit verbundenen Realismus schließen.

Trotz der Wichtigkeit von lippensynchronen Mundbewegungen für die Glaubhaftigkeit gibt es jedoch auch eine Akzeptanz für nicht hundertprozentige Lippensynchronität. Beispielhaft stehen hier vor allem nachsynchronisierte Spielfilme. Der Zuschauer ist hier zwar sehr anfällig auf schlecht synchronisierte Mundbewegungen, ab einer gewissen Schwelle stören asynchrone Bewegungen jedoch nicht mehr. Das bedeutet, dass ein gut synchronisierter Film nichts an Glaubwürdigkeit einbüßt, obwohl die Mundbewegungen der Akteure nicht zu den gesprochenen Texten passen.

Zu erklären ist dies beispielsweise durch die hohe Akzeptanz des Audiosignals, das bei normal Hörenden gegenüber dem Bild bei Sprachmitteilungen deutlich mehr Gewichtung hat. Die Mehrzahl der Menschen entnimmt die Information aus dem Audiosignal und nicht aus den Mundbewegungen.

Um nochmals auf das Beispiel des nachsynchronisierten Filmes zurückzukommen, lassen sich hier interessante Rückschlüsse für die Entwicklung unseres LipSync-Moduls ableiten. Dies bedeutet nämlich, dass eher die Bewegung während des gesprochenen Signals an sich wichtig ist, anstatt der phonemgenauen Mundbewegungen. Je genauer die Bewegungen passen desto besser. Jedoch hat laut dieser Erkenntnisse die Stille absolute Erkennungspriorität.

Neben dem Erkennen der Stille sind die möglichst realistischen Mundbewegungen äußerst wichtig. Das bedeutet, dass möglichst sprunglos zwischen den Mundstellungen gemorpht werden sollte. Es sollten demnach möglichst weiche Bewegungen angestrebt werden, ohne jedoch die Bewegungen zu langsam auszuführen. In einem kontinuierlich gesprochenen Text sind die Lippen meist in Aktion, ohne dabei hektisch zu wirken. Der Bewegungsverlauf sollte somit einem Kurvenverlauf entsprechen.

Außerdem ist es generell zu vermeiden zu starke Unterschiede zwischen den einzelnen Mundstellungen zu modellieren, da die Bewegungen sonst meist nicht homogen wirken.

Trotz der vielen Laute, die es in den unterschiedlichen Sprachen gibt, kann man die Mundbewegungen weitgehend kategorisieren und auf einige wichtige reduzieren. Näheres hierzu in Kapitel 4.3.

4.2 LipSync in Echtzeitsystemen

Um Lippensynchronisation in Echtzeitsystemen zu realisieren gibt es, wie in Kapitel 1.4 bereits erläutert, mehrere Möglichkeiten. Neben 3D-Echtzeit-Anwendungen gibt es natürlich auch noch 2D-Anwendungen. Auch hier ist es möglich Echtzeit-LipSync anzuwenden. Es stehen einem hier zwar nicht so zahlreiche Möglichkeiten der Animationssysteme zur Verfügung, jedoch kann über

Austauschen einzelner Bildbereiche wie z.B. der Mund oder Augenpartie ein ähnlicher Lippensynchronisationseffekt erreicht werden.

Im 3D-Bereich gibt es im Grunde drei Möglichkeiten Mundanimationen in Echtzeit zu realisieren. Zunächst das im Charakter-Animation-Bereich eigentlich nicht mehr zeitgemäße Verwenden von hierarchischen Animationen, dann das für Sekundäranimationen häufig verwendete Morph-System und schließlich die Deformation über Bonesysteme oder Dummies mittels Vertexgewichtung.

Hierarchische Animationssysteme werden sehr selten noch direkt bei Charakteranimationen eingesetzt. Hierarchische Animationen bestehen aus einzelnen Objekten, die hierarchisch miteinander verbunden sind. Wird ein Objekt bewegt oder rotiert so bewegen sich alle in der Hierarchie darunter befindenden Objekte ebenfalls mit. Ein Beispiel für eine Hierarchieanimation wäre Oberarm-Unterarm-Hand-Finger. Wird der Unterarm bewegt, so bewegen sich Hand und Finger ebenfalls. Dieses System ist schnell und nicht schwierig zu implementieren. Nachteilig sind die unschönen Übergänge zwischen den Objekten, so dass sie für die Charakteranimationen und vor allem für die Mundanimation nahezu ungeeignet sind.

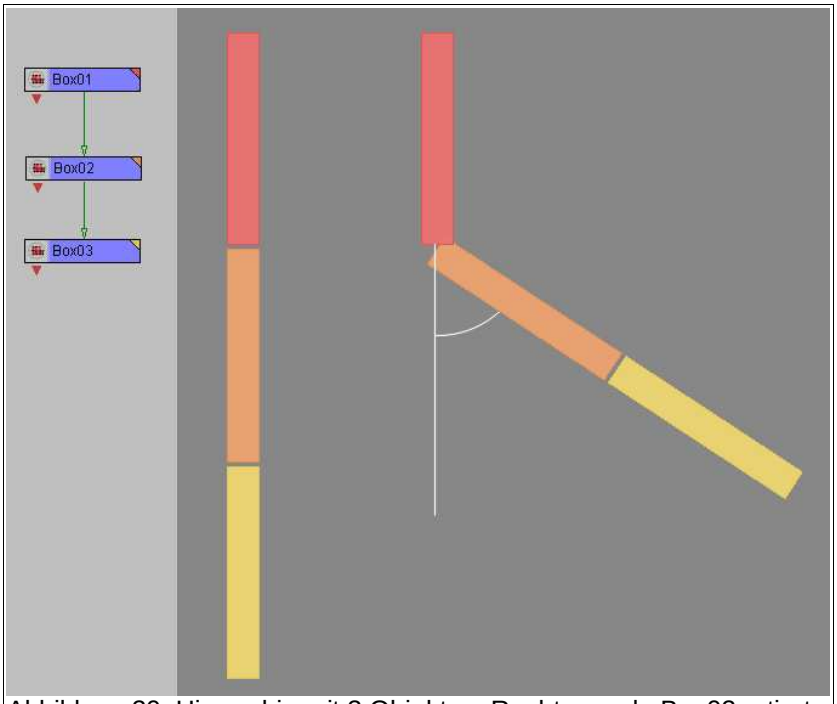

Abbildung 20: Hierarchie mit 3 Objekten. Rechts wurde Box02 rotiert

Bonesysteme werden im Echtzeitbereich in der Regel für die Primäranimationen wie grundsätzliche Körperbewegungen von Armen, Beinen, Kopf, etc. verwendet. Diese Bewegungen lassen sich mittels Morphing eigentlich nicht realisieren. Bonesysteme haben den Vorteil, dass sie flexibel sind und dem natürlichen Bewegungsapparat nachempfunden sind. Das Bonesystem wird mittels Hierarchie animiert. Über Rotation eines Bones wird das Mesh deformiert. Die Deformation erfolgt durch die gewichtete Verbindung der Vertices an den Bone. Da die Berechnungen äußerst komplex sind, sind die Bonesysteme in Echtzeitanwendungen meist noch beschränkt. Beispielsweise ist meist die Anzahl der Bones eines Bonesystems begrenzt und die Anzahl der Bonezuweisungen pro Vertex ebenfalls. Aus diesem Grund verwendet man heutzutage noch keine Bonesysteme für Sekundäranimationen wie Mundbewegungen etc.

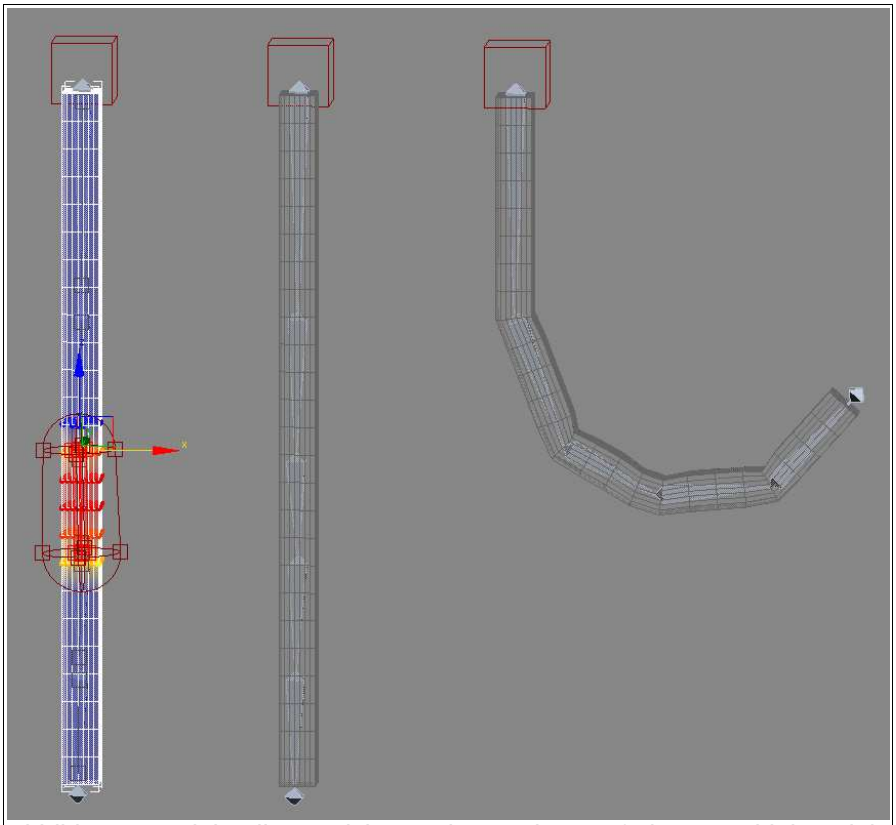

Abbildung 21: Links die Gewichtung der Vertices auf ein Boneobjekt. Mittig das Objekt mit seiner Bonestruktur und rechts das über Rotation der Bones deformierte Objekt

Schließlich bleibt noch das Morphsystem. Hier werden verschiedene Meshes des zu animierenden Objektes erstellt und prozentual zwischen ihnen hin und hergemorpht. Dies geschieht über lineare Verschiebung der Vertices anhand eines Verschiebungsvektors. Aus diesem Grund sind Rotationen mit diesem System nicht zu realisieren, außer man definiert mehrere Zwischenstufen. Da für die meisten Sekundäranimationen wie beispielsweise auch die Mundanimationen keine Rotationsbewegungen nötig sind, kann hier die Morphanimation genutzt werden. Die Morphanimationen sind sehr schnell, einfach zu implementieren und gut mit Bonesystemen kombinierbar. Auch lassen sich durch prozentuale Verwendung verschiedener Morphzustände unterschiedliche Verschiebungsvektoren errechnen. So entsteht eine große Vielfalt an Bewegungen.

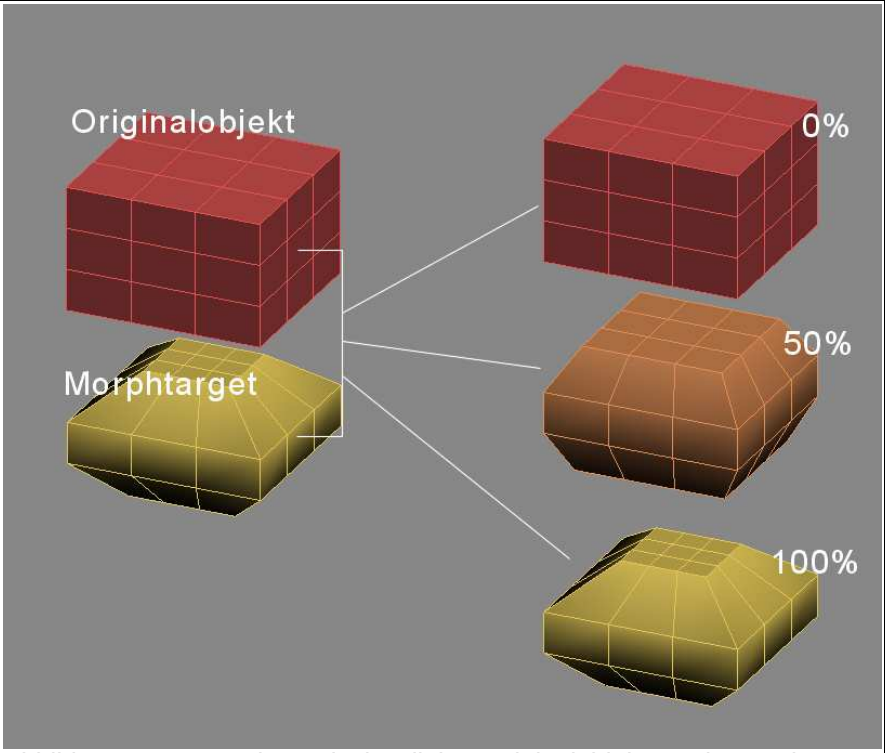

Abbildung 22: Morpher-Prinzip: links Originalobjekt und Morphtarget. Rechts die prozentualen Morphzustände des Originalobjektes

Das im Praxisteil verwendete Prinzip der Movestates entspricht dem Morphsystem. Eine genaue Beschreibung des Movestate-Prinzips erfolgt in Kapitel 5.3.2.2.

Um über die Animationssysteme in Echtzeit die entsprechend zum Audiofile passende Mundbewegung herzustellen benötigt man ein wie in Kapitel 3.5 beschriebenes Echtzeit-Spracherkennungs-System, das an den AnimationsController die entsprechenden Daten für die Mundbewegung liefert.

Einige Beispiele für Computerspiele, die Mundbewegungen und teilweise auch lippensynchrone Mundbewegungen verwenden sind in Kapitel 1.5 beschrieben.

4.3 Mundstellungen

Die verschiedenen Mundstellungen (Viseme) entstehen über die Bewegung des Unterkiefers und die Verformung der Lippen. Der Unterkiefer ist prinzipiell für die vertikale Bewegung verantwortlich und Lippen für die horizontale Bewegung. Seitlich betrachtet bewegt sich der Unterkiefer ebenfalls bei manchen Lauten wie beispielsweise dem /o/ nach vorne bzw bei Lauten wie dem /i/ leicht nach hinten. Ansonsten führt der Unterkiefer eher eine Rotationsbewegung am Kiefergelenk aus.

Annosoft reduziert die menschlichen Viseme auf 11 Zustände (siehe Abbildung23[37]).

37 Abbildung von www.annosoft.com (04.01.2005)

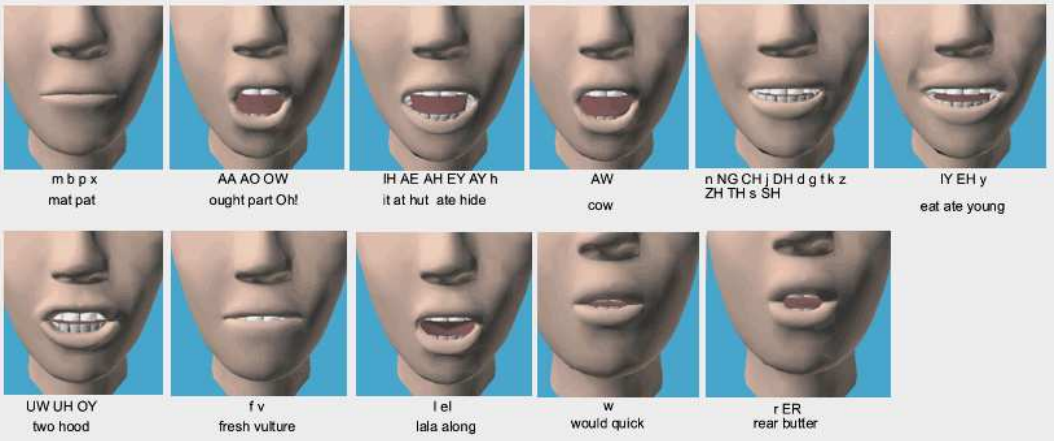

Abbildung 23: 11 Viseme nach Annosoft

Noch weiter vereinfacht können wir die Viseme auf minimal 5 reduzieren. Beispielsweise können /o/ und /u/ zusammengefasst werden sowie /e/ und /i/ und die verschiedenen Formen von /a/.

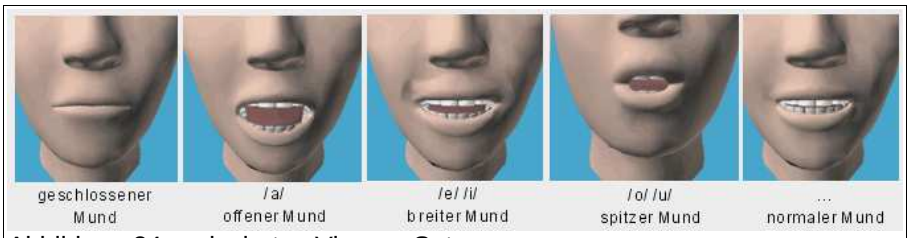

Abbildung 24: reduziertes Viseme-Set

Der geschlossene Mund wird für Phoneme wie das /p/, /b/, /m/,... verwendet, der offene Mund für /a/, der breite Mund für /e/, /i/ und eventuell für /s/, der spitze Mund für /o/, /u/ und eventuell für /sch/ und der normale Mund für den Rest. Da die meisten Sprachen von Vokalen dominiert werden, sollten sich auch aus dem reduzierten Viseme-Set gute sprachenunabhängige Mundbewegungen ableiten lassen.

4.4 Zusammenfassung LipSync-Daten-Pipeline

Das Erstellen der LipSync-Daten beschränkt sich bei Verwendung des Echtzeit-LipSync-Workflows auf die Erstellung der Zustände der Mundanimationen. In diesen Arbeitsschritten liegt die große Einsparung an Aufwand im Gegensatz zu anderen LipSync-Systemen.

Es müssen ein Grundmodell erstellt und exportiert werden sowie für jedes vom System zu erkennende Phonem ein Mundzustand. Der Aufwand kann noch weiter reduziert werden, indem ein Mundzustand für mehrere Phoneme verwendet wird. Beispielsweise könnte eine Phonemanalyse /e/ und /i/ zwar unterscheiden, jedoch könnten beide Phoneme aufgrund der Ähnlichkeit denselben Mundzustand verwenden. Die Anzahl der zu unterscheidenden Phoneme und Mundstellungen sollte demnach zuvor definiert sein.

Der Datenaufwand hält sich je nach verwendeten Datenformaten sehr gering. In der Regel wird das Hauptmodell in einer Modelldatei gesichert und die Mundstellungen zusammen in einer weiteren Datei. Da für die Mundstellungen nur die Verschiebungsvektoren benötigt werden braucht man

lediglich die Differenzdaten zum Grundmodell zu sichern, was den Datenaufwand nochmals erheblich reduziert. Entwickelt man ein eigenes Datenformat ließe sich überlegen, ob man sogar die Mundstellungen direkt in der Datei des Grundmodells mitspeichert. So erhält man eine Datei, die alle für lippensynchrone Mundanimationen benötigten 3D-Daten enthält.

Folgende Abbildung 25 enthält die Erstellungsschritte, die für die LipSync-Daten notwendig sind.

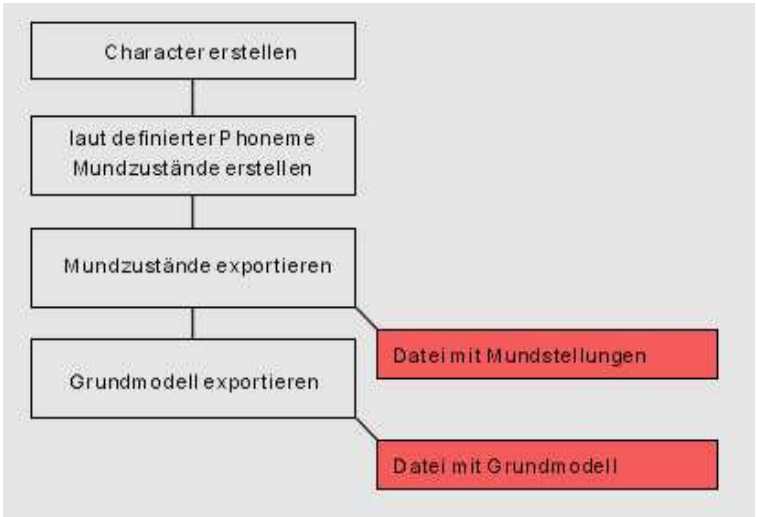

Abbildung 25: Erstellung der 3D-LipSync-Daten

Je nach verwendeten Datenformaten benötigen wir für die Anbindung von Softwarepaketen entsprechende Exporter, die die Daten im gewünschten Format aus der 3D-Anwendung heraus speichern.

Um eine Lippenanimation zu erhalten reicht es nicht, je nach Resultat der Phonemanalyse in den entsprechenden Mundzustand umzuschalten. Es ist sinnvoll, hier zwischen den Mundzuständen weich überzublenden. Diese Verrechnungsaufgabe übernimmt ein Mischmodul, der sogenannte Morpher.

Folgende Abbildung 26 verdeutlicht die Arbeit der Morphers.

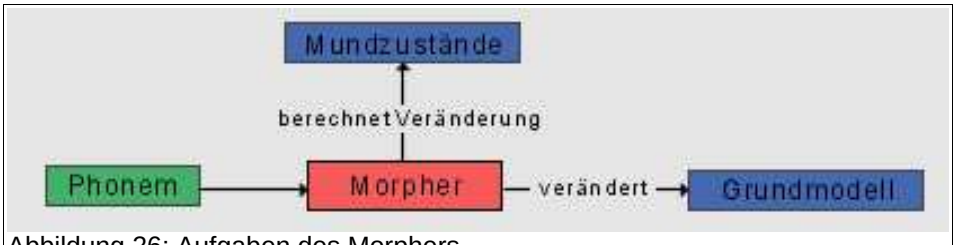

Abbildung 26: Aufgaben des Morphers

Im späteren Kapitel 5.3.2.2 wird noch genauer auf die Funktionalitäten eines Morphers eingegangen.

4.5 Verbesserungsmöglichkeiten

Wie zuvor enthält auch dieses Kapitel einige Verbesserungsmöglichkeiten für die hier vorgestellte LipSync-Daten-Pipeline, die jedoch in dieser Arbeit keine praktische Umsetzung finden. Vielmehr sollen es Anregungen für zukünftige Verbesserungen sein.

Eine genaue Phonemanalyse zusammen mit einer großen Anzahl verschiedener Mundstellungen verbessert das Resultat, erhöht jedoch auch den Aufwand bei der Datenerzeugung.

Das Ergebnis hängt auch maßgeblich von den erzeugten Mundstellungen ab. Hier ließe sich über gewisse Automatisierungsmöglichkeiten beim Erstellen der verschiedenen Mundstellungen in den einzelnen Softwarepaketen nachdenken. Beispielsweise wäre hier ein Automatisierungsvorgang unter Verwendung von Bonesystemen möglich. Dies würde sich jedoch nur bei einer großen Anzahl von Mundstellungen pro Charakter rentieren, da das Skinning[38] und Rigging[39] der Mundpartien eines Charakters auch einigen Aufwand erfordert.

Ansonsten sind die meisten Verbesserungsmöglichkeiten sicherlich in den Verrechnungsregeln des Morphers zu finden. Eine genaue Analyse der Mundbewegungen, bei welchen Phonemen der Mund schnell geöffnet wird und bei welchen eher langsamer und dergleichen könnten hier einberechnet werden. Außerdem ließen sich Varianten einzelner Mundstellungen durch Übersteuerung oder zufälliges Verschieben einzelner Mundpartien berechnen und verwenden, was zu einer noch höheren Varianz der Bewegungen führen würde.

Im Zusammenspiel mit der Audioanalyse ließen sich noch viele weitere interessante Informationen gewinnen und im Morpher verarbeiten. Beispielsweise könnte über die Lautstärke des Audiosignals die Intensität der Mundstellung verändert werden oder andere Audioparameter weitere Facial-Animations in den Morpher mischen.

38 Skinning: Begriff aus der 3D-Grafik: Verbinden des Bonesystems mit dem zu deformierenden Objekt (Vertexgewichtungen definieren)

39 Rigging: Begriff aus der 3D-Grafik: Erstellen eines Bonesystems

5 Beispielhafte praktische Umsetzung

5.1 Ziel der praktischen Umsetzung

Ziel der praktischen Umsetzung ist die Entwicklung einer Beispielanwendung, in der das zuvor im theoretischen Teil erarbeitete Wissen praktisch umgesetzt ist.

Die Beispielanwendung soll dementsprechend einen 3D-Charakter anhand von Audiodaten in Echtzeit möglichst lippensynchron sprechen lassen. Außerdem soll anhand des Programmes die Effiziens des Workflows praktisch getestet werden.

Die Anforderungen an das Programm beschränken sich auf folgende Bereiche:

1. 3D-Daten laden und darstellen (z.B. 3D-Charakter)
2. Audio-Daten laden und abspielen
3. Phonemanalyse der Audiodaten und synchrone Lippenbewegungen des 3D-Charakters ausführen
4. zusätzlich soll eine einfache Navigation innerhalb der Szene notwendig sein

5.2 Konzept

Die Konzeption der Beispielanwendung besteht zunächst in der Aufteilung der Funktionalitäten in einzelne Module. Als nächstes werden die Schnittstellen und Formate für den Datenaustausch definiert.

Folgende Abbildung 27 zeigt das Konzept der Beispielanwendung mit seinen Schnittstellen.

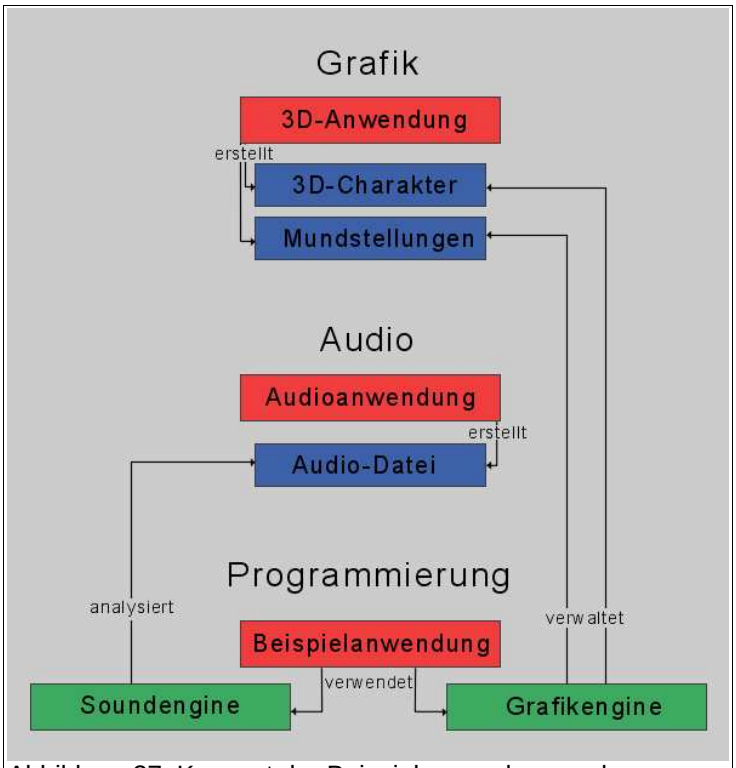

Abbildung 27: Konzept der Beispielanwendung und Schnittstellen

Zunächst modularisieren wir das Programm in den Audio- und den Grafikbereich. Das hat den Vorteil, dass beide Module weitgehend getrennt voneinander entwickelt werden können und später ohne größeren Aufwand auch ausgetauscht werden können.

Die Funktionalitäten beider Module werden letztendlich in der Beispielanwendung verwendet und miteinander verknüpft. So übernimmt das Soundmodul die Analyse der Audiodaten und liefert uns das gefundene Phonem. Anhand der zurückgelieferten Daten wird dann die Grafikengine mit der Aufgabe versehen die Lippen des geladenen 3D-Charakter in die entsprechende Mundstellung zu überführen.

Die nötigen Schnittstellen sind nach Abbildung 27 schnell gefunden. Zwischen Beispielanwendung und Sound- bzw. Grafikmodul definieren die Funktionsaufrufe und Returnwerte die Schnittstellen.

Als Schnittstellenformat zwischen 3D-Anwendung und Grafikengine bietet sich für die Modelldaten das X-File-Format an, da es einfach zu laden ist und prinzipiell aus jeder 3D-Anwendung exportiert werden kann. Es enthält alle für die Beispielanwendung relevanten 3D-Daten. Für die Mundstellungen wäre es unpraktikabel nochmals jeweils alle Modelldaten zu speichern. Aus diesem Grund werden wir ein eigenes Schnittstellenformat entwickeln, das schnell und einfach ist und die für uns wichtigen Daten beinhaltet. Es ist das *.3dm-Format, für das wir später einen Exporter schreiben werden. In Kapitel 5.3.2.2.3 wird das Fileformat genauer erklärt.

Als Schnittstellenformat im Audiobereich dient das gängige Wavefile-Format in unkomprimiertem Zustand und mit den Audioeigenschaften 44.1 kHz, 16 Bit und Mono.

5.3 Die Module

5.3.1 Soundengine

5.3.1.1 Übersicht

Die Aufgaben der Soundengine sind das Laden, Verwalten und Abspielen von Audiodaten. Außerdem bietet es sich an die Funktionalitäten der Spracherkennung in die Soundengine zu integrieren.

In der Beispielanwendung wird die von Frank Brempel entwickelte Soundengine SndMus verwendet, die ich um einige zusätzliche Funktionen für die Phonemerkennung ergänzt habe. Eine genaue Autorenzuweisung einzelner Funktionen gibt es in der Chm-Dokumentation auf der beiliegenden CDRom. Die Engine besteht aus einer Dll[40] und ist somit ideal für den modularen Aufbau geeignet.

Folgend nun eine kurze Auflistung und Beschreibung einiger für das Beispielprogramm wichtigen Funktionen der Soundengine. Der komplette Funktionsumfang der Soundengine SndMus ist in der Dokumentation zur Soundengine beschrieben.

bool InitSnd(HWND hWnd);

Initialisiert die Soundengine.

40 *.dll: Dynamic Link Library ist eine kompilierte Funktionsbibliothek

void ReleaseSnd();

Uninitialisiert die Soundengine.

SOUND* Load(char *file,bool buffer);

*Lädt eine Audiodatei (derzeit nur *.wav unterstützt) und erstellt ein Soundobjekt.*

Rückgabewert ist der Zeiger auf das erstellte Soundobjekt oder NULL.

void Delete(SOUND *sb);

Löscht eine SOUND-Struktur aus dem Speicher .

bool Play(SOUND *sb,bool flag);

Spielt einen Sound von der derzeitigen Playposition ab. Ist flag true so wird das Soundobjekt geloopt abgespielt.

Rückgabewert ist true oder bei Fehler false.

bool Stop(SOUND *sb);

Stoppt den SOUND und setzt die Abspielposition auf Anfang.

Rückgabewert ist true oder bei Fehler false.

bool Pause(SOUND *sb);

Stoppt SOUND, aber behält die Abspielposition bei .

Rückgabewert ist true oder bei Fehler false.

bool SetVolume(SOUND *sb,int db);

Setzt die Lautstärke für das Soundobjekt.

Rückgabewert ist true oder bei Fehler false.

float* GetLeftSpectrum(SOUND *sb);

Gibt das Frequenzspektrum des linken Kanals an der derzeitigen Abspielposition von SOUND zurück .

Rückgabewert ist Pointer auf ein 512er Array von Floatwerten zwischen 0 und 1, die das Frequenzspektrum beschreiben.

float* GetRightSpectrum(SOUND *sb);

Gibt das Frequenzspektrum des rechten Kanals an der derzeitigen Abspielposition von SOUND zurück .

Rückgabewert ist Pointer auf ein 512er Array von Floatwerten zwischen 0 und 1, die das Frequenzspektrum beschreiben.

void Denoise(float* spektrum,unsigned int buffersize,float treshold);

Setzt alle Frequenzwerte des Frequenzspektrums unterhalb von treshold auf 0.

void RCPass(float* spektrum,unsigned int buffersize,float startindex,float endindex);

Führt eine RC-Pass-Filterung (Tiefpass oder Hochpass) auf das Frequenzspektrum aus.

void Normalize(float* spektrum,unsigned int buffersize,float maxvalue);

Normalisiert die Werte des Frequenzspektrum, so dass stärkste Frequenz maxvalue entspricht.

SndMus baut auf DirectSound auf. Durch die Modularisierung ist es jedoch mit geringem Aufwand möglich andere Soundengines mit erweitertem Funktionsumfang oder komplexeren Sprachanalysefunktionen zu verwenden.

5.3.1.2 Phonemanalyse : GetPhonemeFromAnalyseBuffer()

Wie im vorherigen Kapitel schon erwähnt bietet es sich an, die Phonemanalyse ebenfalls in der Soundengine zu implementieren. Für den Programmierer ist es komfortabel, wenn er nur die entsprechende Funktion aufrufen muss und als Ergebnis das ermittelte Phonem erhält.

Deshalb implementieren wir in der Soundengine SndMus zusätzlich eine Phonemanalysefunktion und alle dafür notwendigen Filterfunktionen. Außerdem dürfen wir nicht den Abspielbuffer verändern, da wir ansonsten das Ausgangssignal verändern würden. Aus diesem Grund erstellen wir zusätzlich zum Abspielbuffer noch einen Analysebuffer, der die zuletzt abgespielten Audiodaten beinhaltet. Alle Analysen und Filterungen für die Phonemanalyse können somit problemlos ohne Veränderung des Ausgangssignals durchgeführt werden. Als Weiterentwicklung wäre hier möglich, den Analysebuffer nicht nur auf die schon abgespielten Audiodaten zu begrenzen, sondern den Buffer mit noch abzuspielenden und abgespielten Audiodaten zu befüllen. So könnten eventuelle Zeitverzögerungen weiter reduziert werden.

Die Erstellung und Befüllung des Analysebuffers übernimmt für Mono-Audiodaten folgende Funktion.

double* GetLeftAnalyseBuffer(SOUND *sb,unsigned int *size,bool normalize);

Erstellt und befüllt Analysebuffer. Gibt Zeiger auf Analysebuffer des linken Kanals des Soundbuffers zurück.

Die Phonemanalyse des Analysebuffers übernimmt in der Soundengine SndMus die Funktion GetPhonemeFromAnalyseBuffer. Folgend ein Auszug aus der SndMus-Dokumentation.

unsigned int GetPhonemeFromAnalyseBuffer(double* analysebuffer,unsigned int analysebuffersize,float* pOut_intensity,bool usestatistic);

Führt Phonemanalyse der Daten eines Analysebuffers durch. Ermittelt Phonem und seine Intensität.

Parameters:
double analysebuffer : Zeiger auf Analysebuffer*
unsigned int analysebuffersize : Größe des übergebenen Analysebuffers
float pOut_intensity : bereits allokierter Speicher für die Intensität des Phonems*
bool usestatistic : statistische Nachverrechnung an oder abschalten

Return Value:
unsigned int : Phonem
PHONEME_STILLE
PHONEME_UNIDENTIFIED
PHONEME_SZ
PHONEME_SCH
PHONEME_M
PHONEME_P
PHONEME_I
PHONEME_A
PHONEME_E
PHONEME_U
PHONEME_O

Remarks:
Bisher nur für 44100Hz, 16Bit, mono

Die implementierte Funktion, die die Phonemanalyse anhand des Analysebuffers übernimmt, ist folgend in vereinfachter Form zum Nachvollziehen der Analyseschritte dargestellt. Bei der Darstellung wurden lediglich Fehlerabfangroutinen, nicht relevante Kommentare und für die Erklärung irrelevante Codezeilen entfernt. Der ungekürzte Quellcode befindet sich in der Datei sndmus.cpp auf der Begleit-CDRom.

```
unsigned int GetPhonemeFromAnalyseBuffer(double* analysebuffer,unsigned int
analysebuffersize,float* pOut_intensity,bool usestatistic)
{       //momentan nur für 44100Hz, 16Bit, mono
        ...
//maximalpegel holen
        float maxpegel=GetMaxPegel(analysebuffer,analysebuffersize);
        ...
//zerocrossings ermitteln
        int zerocrossings=GetZeroCrossings(analysebuffer,analysebuffersize);
        ...
//analysebuffer quantisieren (unterabtasten)...44100Hz nicht nötig bei
//spracherkennung...sprachmerkmale bis ca 5kHz
//11kHz reichen aus -> analysebuffer hat nun nur noch eine größe von 512 werten
```

```
        Quantisize(analysebuffer,&analysebuffersize,4);
        ...
//fft -> spektrum hat die größe von analysebuffer/2 = 256 repräsentieren frequenzen von 0-5kHz
        float* localspektrum=GetSpectrumFromAnalyseBuffer(analysebuffer, analysebuffersize,
timesmooth, frequencysmooth);
        ...
//hochpassfilter (unter 150Hz-0)
        RCPass(localspektrum, analysebuffersize, hochpassstartvalue, hochpassendvalue);
        ...
//komplette energie ermitteln
        float energy=GetEnergy(localspektrum,0,analysebuffersize/2,ENERGY_SUMME);
        ...
//maximale energie ermitteln
        float maxenergy=GetEnergy(localspektrum,0,analysebuffersize/2,ENERGY_MAXSCL);
        ...
//grundrauschen herausfiltern
        Denoise(localspektrum,analysebuffersize,denoisetreshold);
        ...
//frequenzen auf 1.0 normalisieren
        Normalize(localspektrum,analysebuffersize,1);
        ...
//merkmalserkennungsvektoren holen
        float energy_n=GetEnergy(spektrum,0,analysebuffersize/2,ENERGY_SUMME);
        float m_szsch1=GetEnergy(spektrum,80,130,ENERGY_SUMME);
        float m_s1=GetEnergy(spektrum,175,240,ENERGY_SUMME);
        float m_p1=GetEnergy(spektrum,3,5,ENERGY_SUMME);
        float m_p2=GetEnergy(spektrum,200,220,ENERGY_MAX);
        float m_ae1=GetEnergy(spektrum,50,100,ENERGY_MAX);
        float m_a1=GetEnergy(spektrum,12,27,ENERGY_SUMME);
        float m_o1=GetEnergy(spektrum,8,18,ENERGY_SUMME);
        ...
//auswertung und returnwerte
        phoneme=PHONEME_UNIDENTIFIED;
        if(zerocrossings<defdata.stille_zerocrossings
                && energy<defdata.stille_energy
                && maxenergy<defdata.stille_maxenergy
```

```
        && m_szsch1<defdata.stille_m_szsch1)
{       //stille
        phoneme=PHONEME_STILLE;
}
else if(m_szsch1>defdata.szsch_m_szsch1)
{       //s/z/sch
        if(m_s1>defdata.sz_ms_1)
        {       //sz
                phoneme=PHONEME_SZ;
        }
        else
                phoneme=PHONEME_SCH;
}
else if(maxenergy>defdata.p_maxenergy
        && energy_n>defdata.p_energy_n
        && m_p1>defdata.p_m_p1
        && m_p2<defdata.p_m_p2)
{       //poplaut
        phoneme=PHONEME_P;
}
else if(m_ae1>defdata.ae_m_ae1)
{       //a,e
        if(m_a1<defdata.a_m_a1)
        {       //a
                phoneme=PHONEME_A;
        }
        else
        {       //e
                phoneme=PHONEME_E;
        }
}
else if(m_ae1<defdata.ouim_m_ae1)
{       //o,u,i,m
        if(m_o1>defdata.u_m_o1)
                phoneme=PHONEME_U;
        else if(m_o1>defdata.o_m_o1)
```

```
                    phoneme=PHONEME_O;
                else if(m_o1>defdata.i_m_o1)
                    phoneme=PHONEME_I;
                else
                    phoneme=PHONEME_M;
            }
            ...
            //statistische phonemzuweisung bei unidentified
            ...
            return phoneme;
        }
```

Grob unterteilt kann die Phonemanalyse in 6 Bereiche gegliedert werden.

1. Filterung des Analysebuffers im Ortsbereich (Quantisierung,...)
2. Filterung des Spektrums des gefilterten Analysebuffers (RCPass, Denoise,...)
3. Merkmalswerte aus Frequenzbereichen berechnen
4. Merkmalswerte anhand Merkmalsregeln auswerten
5. eventuelle statistische Verrechnung
6. Phonem zurückgeben

Tests, den Analysebuffer vor der Analyse über spezielle Fensterungstechniken aufzubereiten, lieferten keine sichtbaren Verbesserungen des Ergebnisses, so dass hier das normale Rechteckfenster Verwendung findet. Abbildung 28[41] zeigt einige mögliche Fensterungsfunktionen für den Analysebuffer, wie sie unter anderem in der Literatur zur Spracherkennung erwähnt werden. Die Größe des Analysebuffers in der Beispielanwendung beträgt 2048 Werte bei 44100Hz, was einem Zeitfenster von ca 46ms entspricht, auf den die Phonemanalyse durchgeführt wird.

41 Abbildung von www.legamedia.net/legamall/2002/02-06/0206_weiss_oliver_sprachanalyse-methoden_01.php (04.01.2005)

Fensterfunktionen	Verlauf	Funktion
Rechteck		wr[n] = 1
Bartlett (Dreieck)		wb[n] = 1 - |2(n-(N-1)/2)/(N-1)|
Hanning		wn[n] = 0.5 - 0.5 cos(2πn/(N-1))
Hamming		wm[n] = 0.54 - 0.46 cos(2πn/(N-1))
Blackman-Harris		wh[n] = 0.35875 - 0.48829 cos(2πn/(N-1)) + 0.14128 cos(4πn/(N-1)) - 0.01168 cos(6πn/(N-1))

Abbildung 28: verschiedene Fensterfunktionen

Da für die Sprachanalyse nur ein Spektrum bis ca 5kHz notwendig ist wird der Analysebuffer zunächst quantisiert. Die Unterabtastung mit nur jedem 4ten Werte führt dazu, dass hohe Frequenzen wegfallen und der zu analysierende Buffer sich verkleinert. Ein kleinerer Analysebuffer bedeutet wiederum weniger Berechnungsschritte in der Fourieranalyse und bringt somit einen Geschwindigkeitszuwachs.

Nach der Fouriertransformation des reduzierten Analysebuffers wird zunächst eine Hochpassfilterung auf tieffrequente Brummgeräusche vorgenommen. Anschließend wird eine Normalisierung des Frequenzspektrums auf den Maximalwert 1.0 durchgeführt, um unterschiedliche Lautstärken auszugleichen.

Im nächsten Schritt erfolgen die Merkmalswertberechnungen der einzelnen Frequenzbereiche und schließlich die Auswertung der gefundenen Merkmale und die Zuordnung zu einem der definierten Phoneme. Die Funktion unterscheidet zwischen folgenden Zuständen: Stille, Zischlaute /sz/, /sch/, Poplaut /p/, Vokale /a/, /e/, /i/, /o/, /u/ und dem Nasal /m/. Wurde keines dieser Laute erkannt so ist das Ergebnis nicht definiert bzw Unidentified.

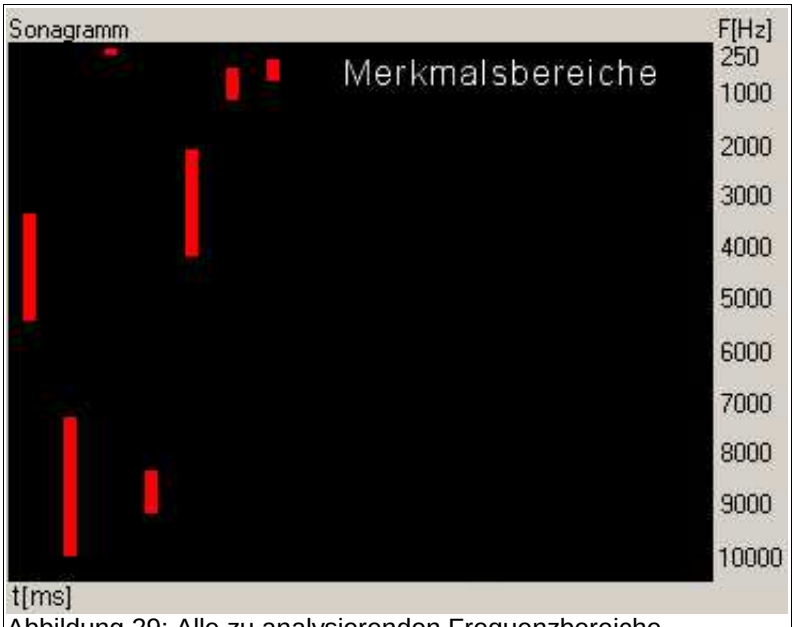

Abbildung 29: Alle zu analysierenden Frequenzbereiche

Anschließend ist noch eine statistische Korrektur des Phonems möglich. Beispielsweise wird dadurch über einen Timer eine zu schnelle Änderung der Phoneme verhindert und bei fehlgeschlagener Phonemerkennung ein Phonem anhand statistischer Wahrscheinlichkeit ermittelt.

Rückgabewert der Funktion ist schließlich das Phonem und eine aus der Lautstärke berechneter Intensitätswert.

5.3.1.3 Der PhonemeAnalyser

Das Tool PhonemeAnalyser wurde entwickelt, um die Merkmalsfindung und Merkmalsmodellierung der Phoneme zu vereinfachen. Es arbeitet eng mit der Phonemerkennungs-Funktion zusammen. Das bedeutet, dass beispielsweise die Maximum-, Minimum- und gemittelten Werte der zur Erkennung modellierten Variablen bequem über eine GUI ermittelt werden können. Die für die derzeitig implementierte Phonemerkennungsfunktion relevanten Variablen können über eine Combobox ausgewählt werden.

Das Tool diente außerdem maßgeblich bei der Entwicklung der GetLeftPhoneme-Funktion, indem diese hier prototypisch getestet und weiterentwickelt werden konnte. Da es sehr viele Möglichkeiten der Vorfilterung und Signalaufbereitung gibt, konnten diese vor der Implementierung in der Soundengine auf Tauglichkeit und Funktionalität überprüft werden. Dazu wurden die Filter zunächst implementiert und über die GUI zugänglich gemacht. So konnten die Auswirkungen direkt an den ausgegebenen Parametern und Visualisierungen verfolgt werden.

Außerdem war es unerlässlich für das Testen der Phonemerkennungsfunktion, da das Tool direkt Feedback über verschiedene Visualisierungen und Ausgabeparameter gibt. So wird beispielsweise das erkannte Phonem in einem Textfeld angezeigt oder das Spektrum des analysierten Audiosegmentes.

Abbildung 30 zeigt die grafische Oberfläche des PhonemeAnalysers.

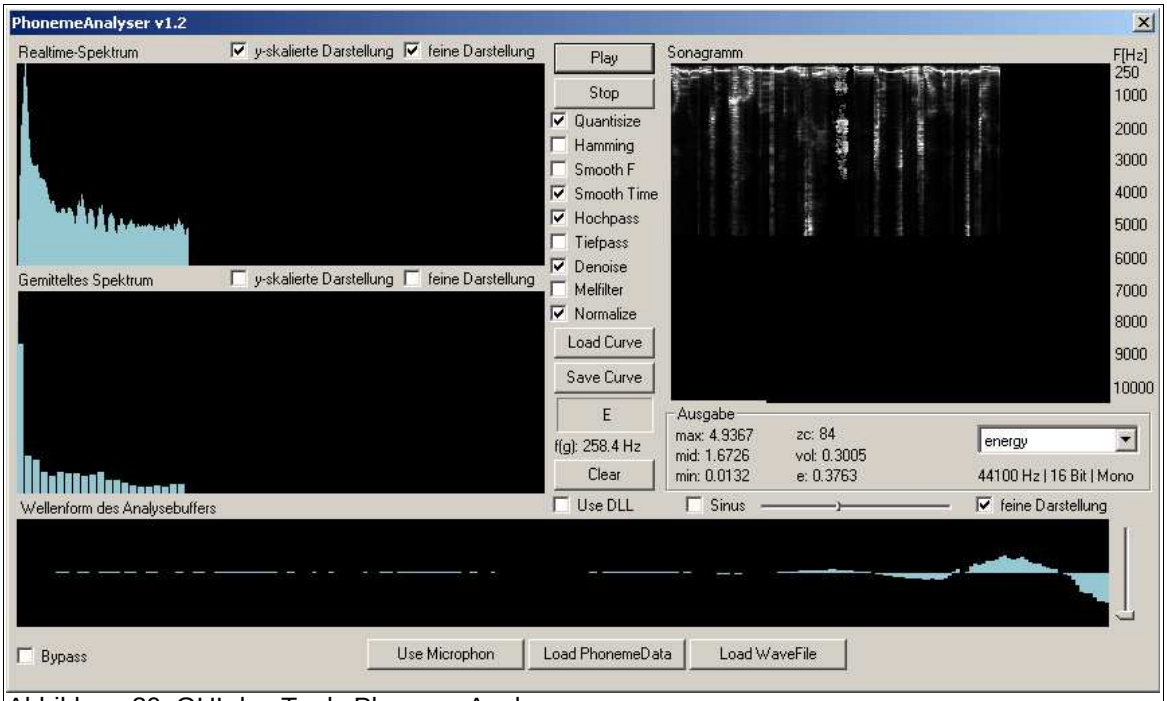

Abbildung 30: GUI des Tools PhonemeAnalyser

Die wichtigsten Eigenschaften des PhonemeAnalysers sind das Sonagramm und die Ausgabe der Maximum-, Minimum- und gemittelten Variablenwerte, die über die Combobox ausgewählt werden können. Zunächst wurden anhand verschiedener Sonagramme Bereiche für die wichtigsten Phoneme festgelegt (siehe dazu Kapitel 3.3 und 3.4). Jede daraufhin definierte Variable bzw. jedes Merkmal repräsentiert einen Spektrumbereich. Für jede Variable wurde daraufhin für die einzelnen Phoneme eingrenzende Wertebereiche ermittelt. Hierzu dienten die Maximum und Minimumwerte.

Da sich beliebige Wavefiles laden und analysieren lassen konnte die Merkmalsfindung auf unterschiedliche Sprecher und Tonhöhen ausgeweitet werden, so dass die Phonemerkennung schlussendlich weitgehend sprecherunabhängig erfolgt.

Zur Erklärung der in Abbildung 30 dargestellten Programmoberfläche folgt nun ein Ausschnitt aus der Dokumentation des PhonemeAnalysers.

> **Realtime-Spektrum** stellt das in Echtzeit ermittelte Spektrum des gewählten Audiosignales dar. X-Achse ist lineare Frequenzachse und Y-Achse ist lineare Intensitätsachse. Über die Auswahlboxen kann die Darstellung verfeinert und die Intensität skaliert werden.
>
> **Gemitteltes Spektrum** stellt das seit letztem Drücken des Clear-Buttons gemittelte Spektrum des gewählten Audiosignales dar. Die Mittlung beginnt bei Play/Record oder nach Klicken des Clear-Buttons. X-Achse ist lineare Frequenzen und Y-Achse ist lineare Intensität. Über die Auswahlboxen kann die Darstellung verfeinert und die Intensität skaliert werden.
>
> **Record/Play** startet das Abspielen des geladenen Wavefiles oder beginnt die Aufnahme.
>
> **Stop/Pause** beendet oder pausiert das Abspielen bzw die Aufnahme. (Pause nur im Abspielmodus möglich)
>
> In der Mitte der Programmoberfläche folgen nun einige Audiosignalbearbeitungsmöglichkeiten. Die Standardauswahl stellt die momentane

Standardfilterung für die Phonemerkennung dar.

Quantisize quantisiert den Audiobuffer von 44100Hz auf 22050Hz.

Hamming wendet ein Hammingwindow auf das Audiosignal an.

Smooth F ist ein Weichzeichner, der die Frequenzen untereinander weichzeichnet.

Smooth Time ist ein Weichzeichner, der das Frequenzspektrum über die Zeit weichzeichnet.

Hochpass führt eine Hochpassfilterung durch (ab ca 100 Hz)

Tiefpass führt eine Tiefpassfilterung durch (ab 10kHz)

Denoise entrauscht das Frequenzspektrum anhand eines (vordefinierten) Thresholds.

Melfilter führt eine Melfilterung des Frequenzspektrums durch.

Normalize normalisiert das Frequenzspektrum auf Maximalwert 1.

Über **Load Curve** und **Save Curve** kann das gemittelte Spektrum gespeichert und wieder geladen werden.

In dem darunter liegenden **Textfeld** wird das erkannte Phonem dargestellt.

f(g) gibt die ermittelte Grundfrequenz zurück.

Ist **UseDLL** aktiviert wird nicht die Phonemanalyse im Quellcode des Phonemeanalysers verwendet sondern die der Soundengine.

Der **Clear**-Button löscht alle Spektren, Arrays und Werte, so dass eine neue Analyse begonnen werden kann.

Das **Sonagramm** stellt den Frequenzverlauf in Abhängigkeit der Zeit dar. X-Achse ist die Zeitachse(ms) und Y-Achse die Frequenzachse.

Ausgabe gibt einige für die Analyse interessante Werte aus. **Max** ist der Maximalwert, **mid** der Mittelwert und **min** der Mindestwert der analysierten Variable. Der Wert, der hier berechnet und ausgegeben werden soll lässt sich mittels der rechts daneben befindlichen **Combobox** bestimmen.

Zc ist die Anzahl der ermittelten Nulldurchgänge, **vol** die Maximallautstärke und **e** die ermittelte Energie.

Wellenform des Analysebuffers zeigt die Wellenform des zu analysierenden Bufferteils nach allen Filterungen. Auch hier kann zwischen grober und **feiner Darstellung** gewechselt werden. Über **Sinus** und dem daneben befindlichen **Slider** lässt sich ein Sinussignal erzeugen.

Über den vertikalen Slider kann die Darstellung in y-Achse skaliert werden.

Bypass deaktiviert alle Filterungen.

UseMicrophone schaltet um auf Mikrofonbenutzung.

LoadPhonemeData lädt Phonemdaten zur Phonemberechnung wie z.B. die Stärke des Denoisefilters aus einem externen File und überschreibt die Standardeinstellungen.

LoadWaveFile lädt ein Wavefile zum abspielen und analysieren. Das Wavefile muss unkomprimiert, 44100Hz, 16Bit und ein Monosignal sein.

Das Programm und seine Dokumentation befindet sich auf der Begleit-CDRom.

5.3.1.4 Auslagerung der Erkennungsmerkmale

Da die Phonemerkennung teilweise unter verschiedenartigen Bedingungen erfolgen muss, ist es sinnvoll einige wichtige Steuerungsparameter der Phonemerkennung außerhalb des Quellcodes zugänglich und veränderbar zu machen.

Zum Beispiel gibt es Sprachfiles schlechter und guter Qualität. In einigen Situationen kann es deshalb sinnvoll sein, die im Laufe der Phonemerkennung vorgenommene Denoisefilterung zu verändern. Hier wäre eine automatische Kalibrierung anhand des Eingangssignals denkbar.

Die maßgebliche Steuerung der Phonemerkennung reguliert das Struct PhonemeData.

```
struct PhonemeData
{
        float timesmooth;
        float frequencysmooth;
        float hochpassstartvalue;
        float hochpassendvalue;
        float denoisetreshold;
        float stille_zerocrossings;
        float stille_energy;
        ...
};
```

Neben einigen anderen Einstellungswerten finden wir auch den Parameter '*denoisetreshold*'. Die derzeit verwendeten Werte bzw. die derzeitig verwendete Struktur kann über die Funktion '*GetCurrentPhonemeData()*' geholt werden. So könnten wir nun bequem den Wert des Thresholds des Denoisefilters den neuen Bedingungen anpassen.

Das komplette Struct kann auch als externe Datei vorliegen. So lassen sich Filterregeln für unterschiedliche Bedingungen in Dateien speichern und wieder abrufen. Das Laden der Filterregeln übernimmt die Funktion '*LoadPhonemeDataFromFile()*'.

Der Aufbau einer solchen Datei entspricht dem Aufbau der PhonemeData-Struktur. Zur besseren Lesbarkeit und Veränderbarkeit wurden jedoch einige Zeilenumbrüche eingefügt.

Folgend der Aufbau einer PhonemeData-Datei.

```
timesmooth,frequenzsmooth,hochpass_start,hochpass_end,denoise_treshold
stille_zeroings,stille_energy,stille_maxenergy,stille_m_szsch1
sz_sch1,sz_ms1
```

p_maxenergy,p_energy_n,p_m_p1,p_m_p2

ae_m_ae1,a_m_a1

ouim_m_ae1,u_m_o1,o_m_o1,i_m_o1

Zum Ermitteln dieser Werte und zum Erstellen einer an neue Bedingungen angepassten PhonemeData-Datei hilft das zuvor beschriebene Tool PhonemeAnalyser.

5.3.2 Grafikengine

5.3.2.1 Übersicht

Die Hauptaufgabe der Grafikengine besteht darin, 3D-Daten zu laden, zu verwalten und darzustellen. Außerdem sollen die 3D-Daten über verschiedene Funktionen manipuliert und verändert werden können.

Für die Beispielanwendung kommt die selbstentwickelte Grafikengine Gfx3D zum Einsatz. Sie arbeitet eng mit Direct3D zusammen. Das bedeutet, dass die komplette Darstellung, die Schnittstellen der Grafikformate und viele Funktionalitäten von Direct3D übernommen werden und in der Engine für den Benutzer vereinfacht zur Verfügung gestellt werden.

Aus diesem Grund bietet sich als 3D-Modellformat das DirectX-eigene *.x-Format an. Eine entsprechende Laderoutine von Direct3D ist in folgender Funktion gekapselt.

OBJECT* CreateXObject(char* xfile);

*Lädt ein *.X-File und gibt Zeiger auf das erstellte Objekt zurück.*

Gibt Zeiger auf Objekt zurück oder NULL bei Fehler.

Die interne Verwaltung der Daten wie Texturen, Materialien, Meshdaten und dergleichen geschieht teils über Direct3D-Klassen und teils über interne Listen. Dies hat den Vorteil, dass die Engine die Ressourcenverwaltung überwachen kann und somit Speicherlecks vermieden werden können.

Um ein Objekt darzustellen werden die von Direct3D vorhandenen Renderroutinen verwendet und in einer entsprechenden Render-Routine zusammengefasst. Die Renderroutine verwaltet die Aufrufe der Draw-Methoden der einzelnen Objekte anhand mehrerer Listen. Die Draw-Methoden schicken schließlich das Objekt mit seinen Material- und Textureigenschaften in die Renderpipeline von Direct3D, das letztendlich die Darstellung übernimmt.

void OBJECT::Draw();

Objekt darstellen (in die Direct3D-Renderpipeline schicken).

void Render();

Rendert komplette Szene.

Um ein Objekt verändern zu können müssen wir direkt auf die Vertexdaten des Objektes zugreifen können. Die Vertexdaten liegen in der Regel im Grafikkartenspeicher, weshalb ein direkter Zugriff auf diese Daten zu Performanceproblemen führt, da für jede Abfrage der Speicherbereich gesperrt werden muss. Aus diesem Grund sind alle Vertexdaten in einem Parallelarray in der Objektklasse abgelegt. Dies erhöht zwar den Speicherverbrauch etwas, spart jedoch Performance, da diese Daten direkt im RAM-Speicher liegen und somit ein schneller Zugriff möglich ist. Vor dem Rendervorgang wird dann, falls notwendig, in der Draw-Methode der komplette Vertexbuffer in den Grafikkartenspeicher kopiert. Zum Verändern der Vertexpositionen sind unter anderem folgende Funktionen implementiert.

OBJECT::UpdateVertexBuffer();

Kopiert eventuelle Veränderungen des Vertexarrays in den Vertexbuffer .

void OBJECT::SetVertexPosition(unsigned int vertexnumber, D3DXVECTOR3* posXYZ);

Setzt angegebenen Vertex im Vertexarray an neue Position (Wirkung wird erst durch UpdateVertexbuffer() sichtbar!).

void OBJECT::MoveVertex(unsigned int vertexnumber, D3DXVECTOR3* move);

Verschiebt angegebenen Vertex im Vertexarray anhand des Vektors (Wirkung wird erst durch UpdateVertexbuffer sichtbar!) .

unsigned int OBJECT::GetVertices(unsigned int*& pOut, D3DXVECTOR3* position, float radius);

Füllt Integerarray mit Vertexnummern, deren Vertices innerhalb der angegebenen Kugel liegen und gibt deren Anzahl zurück.

D3DXVECTOR3 OBJECT::GetVertexPosition(unsigned int vertexnumber);

Gibt Position des angegebenen Vertex zurück .

Die im nächsten Kapitel vorgestellten Movestates verwenden diese Vertexmanipulationsfunktionen um ein Objekt in seiner Form in einen anderen vorgegebenen Zustand zu überführen.

Eine Übersicht der derzeitigen Funktionalität der Gfx3D-Engine ist in der Dokumentation auf der beiliegenden CDRom zu finden.

5.3.2.2 Movestates

5.3.2.2.1 Konzept

Movestates dienen dazu, die verschiedenen Zustände eines 3D-Objektes zu beschreiben. Sie entsprechen vom Konzept her den aus 3D-Paketen bekannten Morphtargets oder Blendshapes.

Mit Hilfe der Movestates lässt sich demnach das Aussehen eines 3D-Objektes auf Vertexbasis verändern. Sie sind deshalb ideal zum Speichern der verschiedenen Mundstellungen geeignet.

Bedingung beim Erstellen der Movestates ist, dass sie alle aus Kopien des Hauptobjektes erstellt werden, so dass alle Objekte die gleiche Anzahl an Vertices mit gleichen Vertexnummern besitzen.

Ein Vorteil der Movestates ist, dass es nur die Differenzdaten (grüne Vektorlinien in Abbildung 31) der Morphtargets (rote Objekte in Abbildung 31) zum Ursprungsobjekt (blaues Objekt in Abbildung 31) in 3D-Vektorform speichert und somit der Datenaufwand im gespeicherten File, sowie später im geladenen Zustand in der 3D-Engine, sehr gering bleibt.

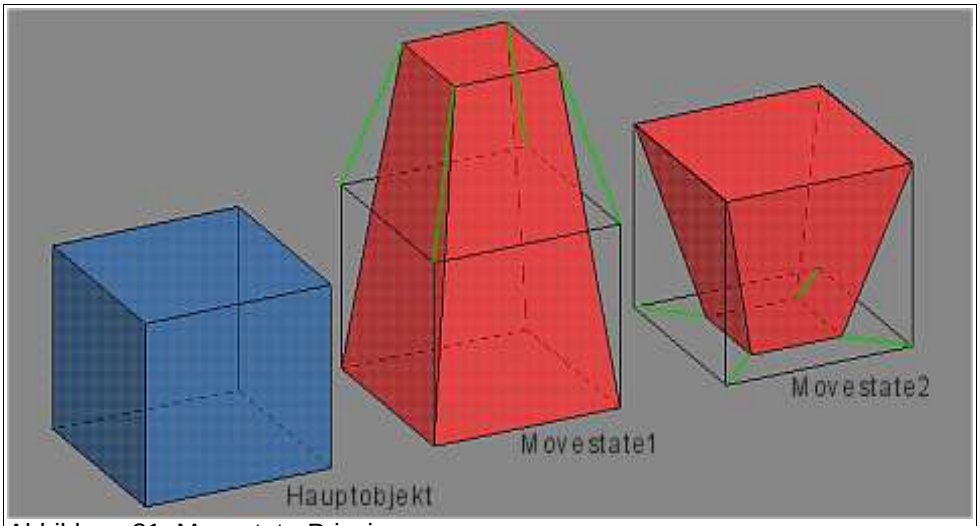

Abbildung 31: Movestate-Prinzip

Für alle sich verändernden Vertices des Objektes werden die Vertexnummer und der Verschiebungsvektor gespeichert.

Anhand dieser Informationen kann dann später in der Engine das geladene Ursprungsobjekt über das Verschieben der Vertices anhand der Movestate-Daten exakt in die gewollte Form überführt werden.

Ein weiterer Vorteil beim Arbeiten mit Verschiebungsvektoren ist, dass wir die Vektoren sehr leicht verkürzen und verlängern können. Sie sind also ideal für Animationen geeignet, bei denen man das Ursprungsobjekt in den definierten Movestate überblenden möchte. Außerdem sind auch Übertreibungen durch Verlängerung des Verschiebungvektors möglich.

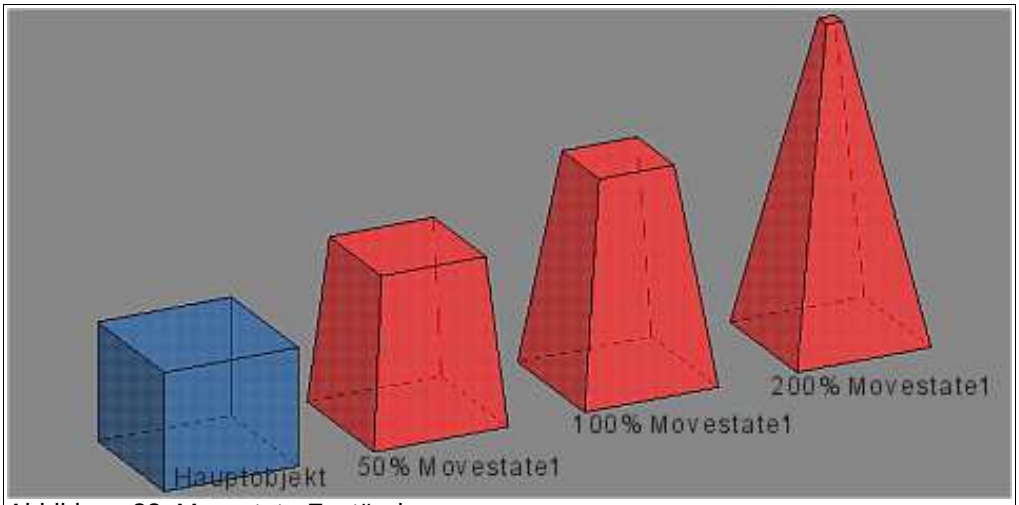

Abbildung 32: Movestate-Zustände

Nachteilig an dieser Form ist das Speichern der Vertexnummern, die wir ja für die Zuordnungen der Verschiebungsvektoren benötigen. Es ist nämlich nicht garantiert, dass die Vertexnummerierung beim Speichern des Hauptobjektes durch exporterbedingte Meshoptimierungen nicht verändert wird. Hierfür müssen wir beim Laden der Movestates die Vertexnummerierungen anhand der Positionen nochmals überprüfen und entsprechend umsortieren.

Das Problem kann man umgehen, indem man idealerweise ein eigenes 3D-Objektformat entwickelt.

Movestates sollten theoretisch auch zusammen mit Skelettanimationssystemen funktionieren. Um ein korrektes Verschieben der Vertices zu garantieren müssen die Movestates vor der Skelettdeformation stattfinden oder die Verschiebungsvektoren auf das lokale Achsensystem des zuständigen Bones umgerechnet werden.

5.3.2.2.2 3D Studio Max-Exporter

Um Movestates für unsere Beispielanwendung herzustellen verwenden wir das 3D-Programm 3D Studio Max R5[42]. Aus diesem Grund entwickeln wir unseren Movestate-Exporter für die Anbindung an 3D Studio. Aufgrund ihrer Einfachheit bei gleichzeitiger Leistungsfähigkeit verwenden wir die integrierte Scriptsprache MaxScript, in der sich die Aufgabe in wenigen Zeilen realisieren lässt.

Das Script soll mehrere Movestates nacheinander in ein File speichern. Jeder Movestate soll anhand eines Namens identifizierbar sein. Außerdem benötigen wir pro Movestate die Vertexnummern und die Verschiebungsvektoren der veränderten Vertices, die im von der Engine verwendeten LeftHanded-Koordinatensystem abgespeichert werden müssen.

Die Benennungen der Movestates werden einfach aus deren Objektnamen ermittelt.

Zum Ermitteln der Verschiebungsvektoren benötigen wir das Ursprungsobjekt. Aus diesem Grund bauen wir in die GUI unseres Exporters einen Pickbutton ein, mit dem man vor dem Exportvorgang das Hauptobjekt anwählen muss. Die zu exportierenden Movestates werden einfach über die aktuelle Objektauswahl definiert. Jedes Objekt wird durchgegangen und die lokalen Vertexpositionen mit den entsprechenden des Hauptobjektes verglichen. Liegt die Position oberhalb der Rundungsschwelle (hier 0.0001), so wird der Verschiebungsvektor berechnet und mit der Vertexnummer zwischengespeichert.

42 3D Softwarepaket von discreet, Homepage: www.discreet.com

Die so gewonnenen Daten werden anhand der 3dm-Dateiformat-Spezifikationen (siehe Kapitel 5.3.2.2.3) abgespeichert.

Zusätzlich bietet der Exporter die Funktionalität an, die Daten auch im Ascii-Format zu speichern. Dies kann für Überprüfungszwecke durchaus sinnvoll sein, da die Movestate-Daten binär gespeichert werden.

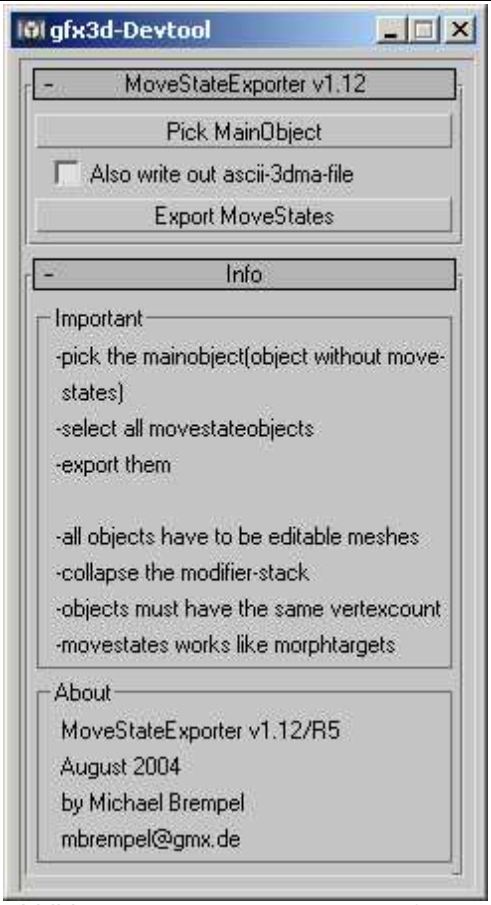

Abbildung 33: Movestate-Exporter für 3D Studio Max

Der Script-Quellcode des Movestate-Exporters befindet sich im Anhang und auf beiliegender CDRom.

5.3.2.2.3 Movestate-Fileformat (*.3dm und *.3dma)

Das Movestate-Fileformat *.3dm ist ein binäres selbstentwickeltes Format, das alle Informationen beinhaltet, um auf ein Objekt Movestates anzuwenden.

Die ersten 6 Byte beschreiben die FileID. Hier kann beispielsweise die aktuelle Version des 3dm-Files gespeichert werden. Außerdem kann erkannt werden, ob es sich beim geladenen File tatsächlich um ein 3dm-File handelt. Aktueller FileID-String ist „3dm01“.

Die nächsten 4 Byte ist die Anzahl der gespeicherten Movestates im unsigned int-Format.

Nun folgen die Movestates. Die ersten 4 Byte als unsigned int gibt die Länge des nun folgenden Namens des Movestates an. Dadurch ist es möglich einen quasi beliebig langen Namen zu verwenden. Dieser kann somit direkt in einen vorbereiteten Char-Array geladen werden. Darauf folgt wiederum als 4 Byte unsigned int die Anzahl der Vertices dieses Movestates. Entsprechend der Vertexanzahl können nun die Vertexnummern, die je in Folge ebenfalls als 4 Byte unsigned int gespeichert sind, an einem Stück in einen Array geladen werden. Genauso kann man mit den nun folgenden Verschiebungsvektoren verfahren. Diese liegen als 3x4 Byte-float-Werte als XYZ-Vektor vor.

Nach den Movestates werden noch alle sich verändernden Vertices aller Movestates gesichert. Es werden die Vertexnummern als 4 Byte unsigned int und ihre Vertexposition im Hauptobjekt als 3x4 Byte-float-Werte bzw als XYZ-3DVektor gespeichert. Dies ist notwendig, falls man die Vertexnummern beim Laden der Movestates neu zuweisen muss, falls diese durch den Objektexporter verändert werden.

***.3dm-Fileformat (3dm01)**		
6 Byte	char	FileID und Versionsnummer
4 Byte	Unsigned int	Anzahl der gespeicherten Movestates
4 Byte	Unsigned int	Namenslänge des Movestates
X Byte	char	Movestatename
4 Byte	Unsigned int	Anzahl der Vertices des Movestates
4 Byte	Unsigned int	Vertexnummern
...		Weitere Vertexnummern
12 Byte	3DVector	Verschiebungsvektor XYZ (float, float, float)
...		Weitere Verschiebungsvektoren
...		Weitere Movestates
4 Byte	Unsigned int	Anzahl Vertices aller Movestates
4 Byte	Unsigned int	Vertexnummern
...		Weitere Vertexnummern
12 Byte	3DVector	Vertexpositionen XYZ (float, float, float)
...		Weitere Vertexpositionen

Vorteile dieses Fileformats sind seine Flexibilität trotz des einfachen Aufbaus. Die Movestates können sehr schnell geladen werden, da die Vertexnummern und Verschiebungsvektoren direkt aus dem File in ein vorbereitetes Array kopiert werden können.

Das *.3dma-Fileformat ist eine leicht formatierte Ausgabe des binären *.3dm-Files als Ascii-File. Um das Lesen des Files zu vereinfachen sind zusätzlich einige Zeilenumbrüche und Lesehilfen eingebaut.

Der MovestateExporter ist derzeit für 3D Studio Max realisiert. Das Anbinden anderer Softwarepakete wie Maya[43], Cinema4D[44], Softimage[45], Lightwave[46] und andere ist jedoch kein

43 3D Softwarepaket von Alias, Homepage: www.alias.com
44 3D Softwarepaket von Maxon, Homepage: www.maxon.net
45 3D Softwarepaket von Avid, Homepage: www.avid.com
46 3D Softwarepaket von NewTek, Homepage: www.newtek.com

Problem. Man müsste lediglich den MovestateExporter auf das Softwarepaket anpassen um die Modelldaten und Movestates exportieren zu können.

5.3.2.2.4 Movestates in der Grafikengine

In der Engine werden Movestates als Modifier behandelt. Klassen und Funktionen befinden sich demzufolge in der modifiers.h und modifiers.cpp der Grafikengine.

Die Engine muss Movestates über das *.3dm-File laden können, die Movestates einem Objekt zuordnen können und entsprechend die Verwaltung über die Verwendung der Movestates übernehmen können.

Beim Laden muss die Möglichkeit bestehen, die Vertexnummern neu zuordnen zu lassen. Zur einfachen Handhabung benötigen wir Funktionen, mit denen wir einen Movestate zu einem gewissen Prozentsatz auf ein Objekt anwenden sowie alle Movestates schnell zurücksetzen können. Da mehrere Movestates gleichzeitig auf ein Objekt angewendet werden können ist es sinnvoll auch verschiedene Verrechnungen zur Verfügung zu stellen. In unserem Fall reichen vorerst vier aus. Einmal wirkt sich die Erhöhung eines Movestates nicht auf die anderen aus (wird durch das Makro GFX3D_NONE definiert), einmal können über die Verwendung von GFX3D_RESET alle anderen nicht als 'interactable' definierten Movestates automatisch auf 0% zurückgesetzt werden, wobei GFX3D_FORCERESET auch diese zurücksetzt, und zuletzt können mit GFX3D_INTERACT die anderen Movestates um verrechnete Prozentwerte vermindert werden.

Manche Movestates wie z.B. das Augenzwinkern, die unabhängig der anderen Movestates sind, können über das Flag 'interactable' von den anderen Movestates separiert werden. Das Ändern der anderen Movestates hat dann keinen Einfluss auf den als nicht interaktiv deklarierten Movestate.

Da einem Objekt für gewöhnlich ein ganzer Satz an Movestates zur Verfügung steht werden diese nochmals in einer MoveStateSet-Klasse zusammengefasst. Hier liegen die Movestates als Array vor und die Prozentwerte als Parallelarray. Der Anwender kommuniziert nur mit der MoveStateSet-Klasse. Das Aktualisieren der Vertexdaten des 3D-Objektes übernimmt die Objektklasse vor dem Rendervorgang.

Folgend eine kurze Beschreibung der wichtigsten Movestate-Funktionen. Der C++ Quellcode und die Headerdateien (hier speziell die modifiers.cpp und modifiers.h) sind auf der CDRom und teilweise im Anhang zu finden. Genauere Angaben zu den Funktionen können der entsprechenden Dokumentation auf der Begleit-CDRom entnommen werden.

MOVESTATESET* LoadMoveStates(char* filename,OBJECT* obj,bool reorganize);

*Movestates aus *.3dm-File laden, in einem Movestateset unterbringen und einem Objekt zuweisen. Außerdem bietet die Funktion die Möglichkeit die Vertexnummern neu generieren zu lassen.*

Rückgabewert ist ein Pointer auf das erstellte Movestateset oder NULL.

void Delete(MOVESTATESET* movestateset);

Löscht Movestateset mitsamt enthaltenen Movestates. Gibt verwendete Ressourcen frei.

void MOVESTATESET::SetInteractable(unsigned int index,bool interactable);

Als nicht 'interactable' definierte Movestates können nicht interaktiv von anderen Movestates verändert werden.

int MOVESTATESET::GetMoveStateByName(char* name);

Ermittelt Index des Movestates anhand des Movestatenamens.

Rückgabewert ist der Index oder -1. (Anmerkung: -1 Fehlercodes müssen selbst abgefangen werden, da die meisten Funktionen unsigned int als index erwarten)

void MOVESTATESET::SetPercentage(unsigned int index,float percentage,DWORD mode);

Gibt an, zu wieviel Prozent der über den Index angesprochene Movestate verwendet werden soll. Außerdem kann über mode der Verrechnungsmodus angegeben werden (GFX3D_NONE, GFX3D_RESET, GFX3D_FORCERESET oder GFX3D_INTERACT)

void MOVESTATESET::SetInteractable(unsigned int index,bool interactable);

Wenn der über den Index angesprochene Movestate 'interactable' ist kann er von anderen Movestates beeinflusst werden bzw wird bei ResetMoveStates() auf 0 zurückgesetzt.

Standardmäßig sind alle Movestates 'interactable'.

void MOVESTATESET::ResetMoveStates(bool forcereset);

Setzt alle nicht als 'interactable' deklarierten Movestates auf 0%. Stellt Ausgangsobjekt wieder her. Ist 'forcereset' true werden auch alle nicht 'interactable' Movestates zurückgesetzt.

void MOVESTATESET::ResetMoveState(unsigned int index);

Setzt den per Index angegebenen Movestate auf 0%. (Entspricht UseMovestateset (...,0,GFX3D_NONE))

Die Animation kommt über frameweise Veränderungen des Prozentwertes der verschiedenen Movestates zustande. In unserem Beispiel führt uns die Abhängigkeit dieses Prozentwertes von den Audiodatenberechnungen zu den gewollten synchronen Lippenbewegungen.

5.4 Kurzes Tutorial 1 : 3D-Artist-Workflow

Dieses kleine Tutorial beschreibt, wie für die Beispielanwendung der 3D-Charakter und die Movestates vorbereitet werden. Als 3D-Software wird 3D Studio Max R5 verwendet. Movestates werden über das MovestateExporter-Script v1.11 exportiert und das Hauptobjekt als *.X-Datei[47] über den Panda-DirectX-Exporter v4.3.0.47[48] abgespeichert.

Zunächst ist das Angleichen der Einheiten zwischen 3D Studio und der 3D-Engine wichtig. Eine generische Unit in 3D Studio entspricht einer Einheit (Meter) in der Engine.

47 Standard-Objektformat für Microsoft DirectX (Direct3D)
48 Freeware-XFile-Exporter-Plugin für 3D Studio Max, Homepage von Pandasoft: www.pandasoft.demon.co.uk

Nun bauen wir unter Berücksichtigung der LowPoly-Regeln[49] unseren Charakter. Der finale 3D-Charakter wird in ein EditableMesh[50] umgewandelt und der Modifier-Stack[51] wird reduziert. Falls wir das 3D-Objekt skaliert haben so müssen wir über Hierarchy -> ResetScale die Skalierung fest zuweisen. Dadurch wird sie aus der Objektmatrix rausgerechnet und direkt in die Vertexpositionen eingerechnet. Spiegelungen bzw. negative Skalierungen müssen ebenfalls rausgerechnet werden. Dies geschieht in 3D Studio Max über das Tool Utilities -> ResetXForm.

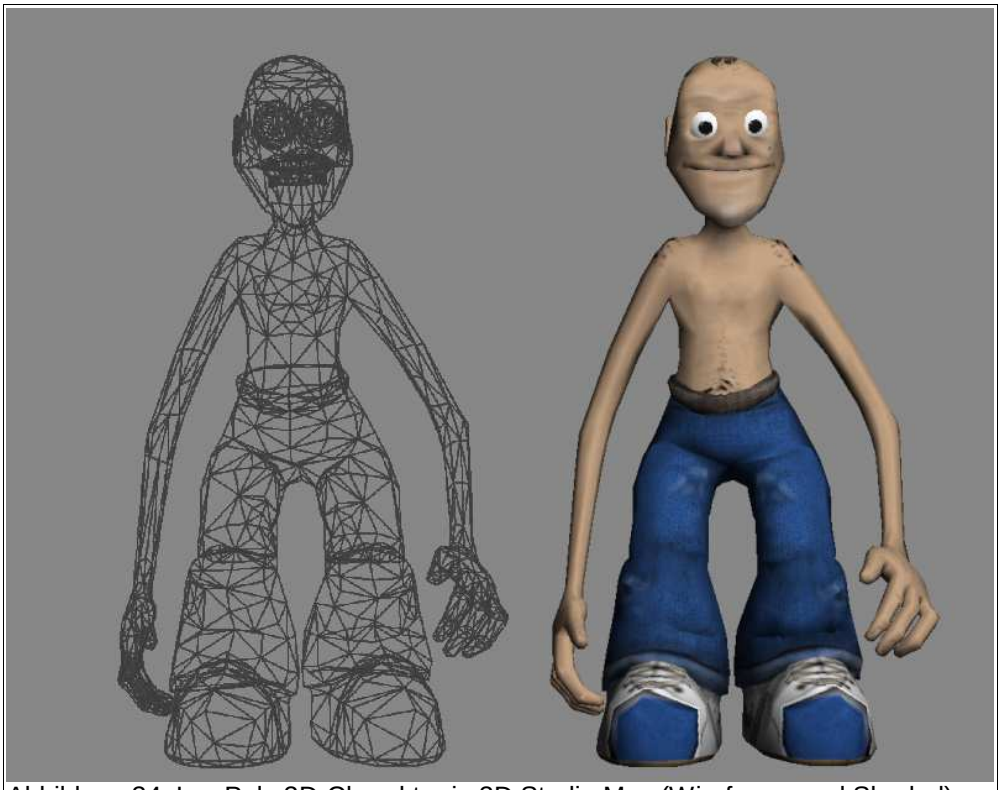

Abbildung 34: LowPoly-3D-Charakter in 3D Studio Max (Wireframe und Shaded)

Nun erstellen wir für jeden Movestate eine Kopie des Objektes und bearbeiten die Meshvertices nach unseren Vorstellungen. Anschließend benennen wir die Movestate-Objekte, um sie später in der Engine wieder zuordnen zu können.

49 LowPoly-Regeln: z.B. Anzahl der Polygone, Größe der Texturen,...Ziel ist es ein Objekt mit so wenig Meshdaten wie möglich zu beschreiben
50 EditableMesh: 3D-Objekt wird über Gitternetz(Mesh) beschrieben
51 Modifier-Stack: Bearbeitungshistory eines Objektes

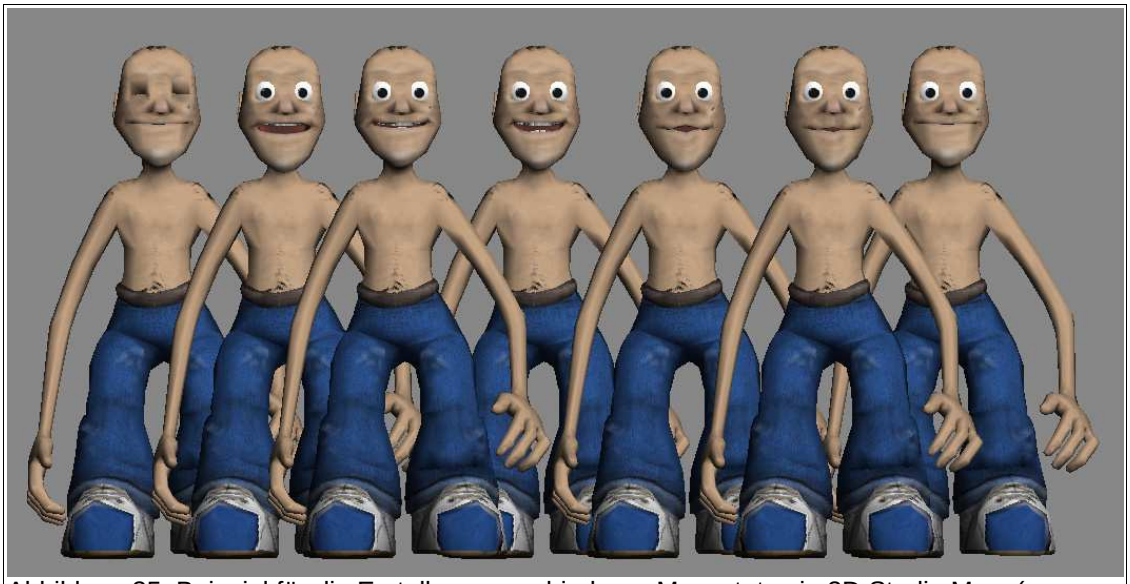

Abbildung 35: Beispiel für die Erstellung verschiedener Movestates in 3D Studio Max. (von links nach rechts: Augen zu, A, I, E, O, U, M)

Zunächst exportieren wir die Movestates über das MovestateScript. Wir starten das Script und wählen über den Pickbutton „Pick MainObject“ unser Hauptobjekt an. Nun selektieren wir die Movestates, die wir exportieren möchten und klicken „Export MoveStates“. Die Daten werden im angegebenen *.3dm-File gespeichert.

Schlussendlich benötigen wir noch das Hauptobjekt als *.X-File. Wir selektieren das Objekt und exportieren es über das PandaX-File-Exporter-Plugin. Dabei ist wichtig, dass wir „Left Handed-Coordinate System“ und „Use Local Objectspace“ anwählen, da die Engine im Gegensatz zu 3D Studio Max im Left-Handed-Koordinatensystem arbeitet.

Abbildung 36 stellt nochmal das Prinzip des kompletten Workflow zur Erstellung der Objektdaten dar. Den Rest wird die Engine für uns übernehmen und wird programmiertechnische Arbeit sein.

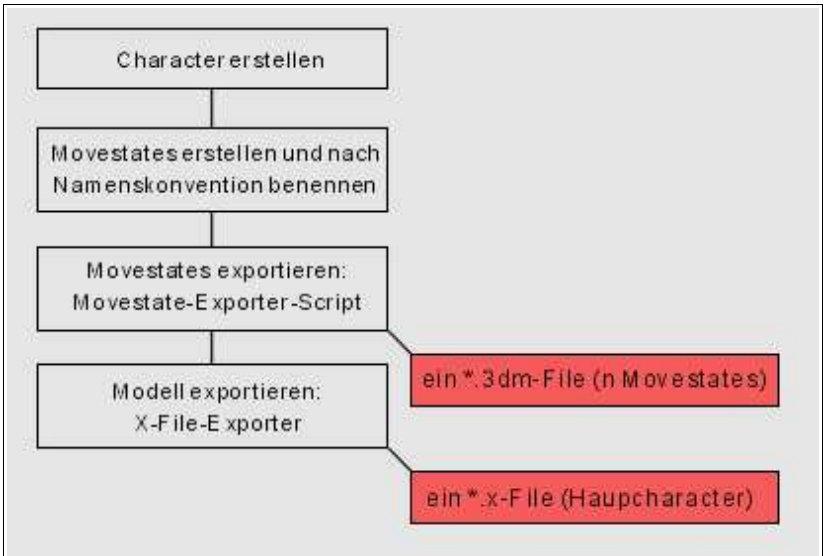

Abbildung 36: 3D-Artist-Workflow

5.5 Kurzes Tutorial 2 : Programmer-Workflow

Der Programmierteil in unserem Beispiel gestaltet sich durch die Funktionen der Grafik- und Soundengine in unserer Beispielanwendung sehr komfortabel. Wir müssen eigentlich nur noch die Modell-, Movestate- und Sounddaten laden, den Sound abspielen, die Phonemerkennung starten und die erhaltenen Werte auf die Movestates anwenden. Eventuell sollten die erhaltenen Phonemdaten zur Intensität des Movestates noch etwas weichgezeichnet werden um harte Übergänge zu vermeiden.

Da die Phonemerkennung nur ca alle 10-20ms durchgeführt werden soll ist dieser Vorgang timergesteuert.

Folgend eine Abbildung der grundsätzlichen Implementierung der LippenSynchronisation in unserer Beispielanwendung.

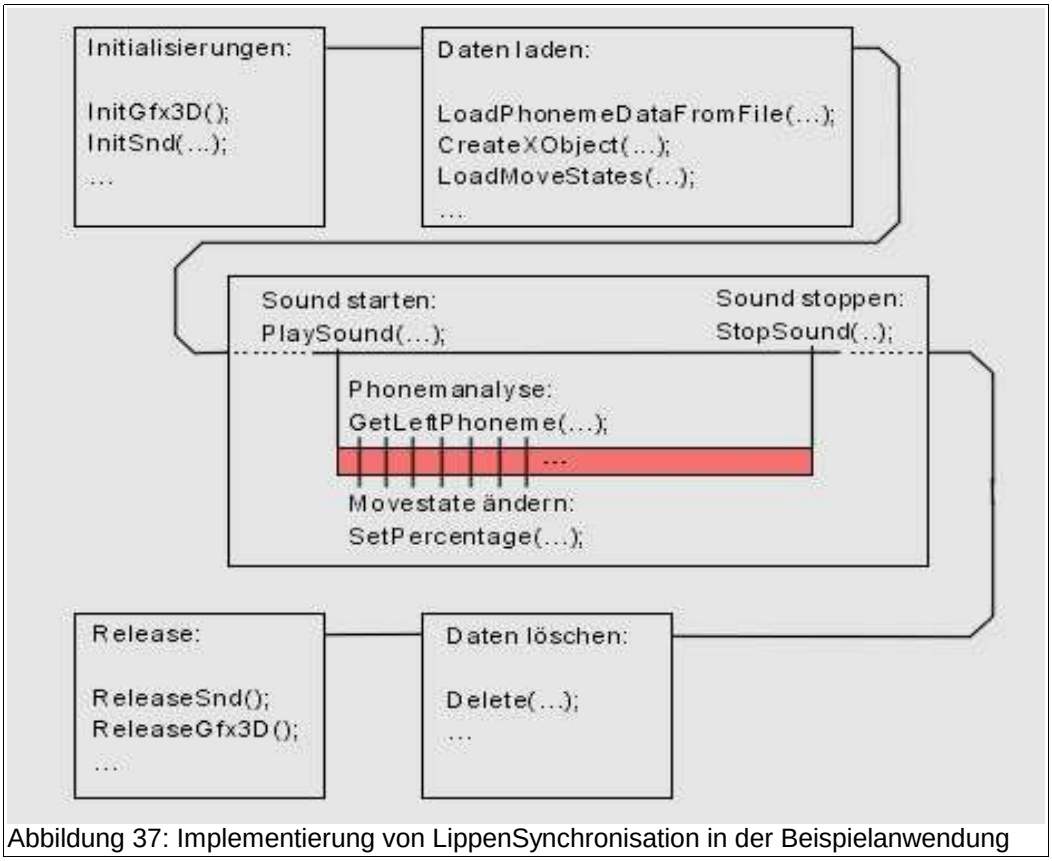

Abbildung 37: Implementierung von LippenSynchronisation in der Beispielanwendung

5.6 Das Beispielprogramm

Das Beispielprogramm Multi3D verknüpft die Soundengine SndMus und die Grafikengine Gfx3D zu dem gewünschten LipSync-Demoprogramm. Bei der Erstellung der 3D-Daten und der Implementation der Sprachanalyse wurden die in den vorigen Kapiteln besprochenen Workflows verwendet.

Die Beispielanwendung soll den Beweis dafür liefern, dass die Spracherkennung und die synchrone Übertragung der entsprechenden Mundstellung vom Ansatz her funktionieren. Hierbei wird nicht das Hauptaugenmerk auf eine hundertprozentige Lippensynchronisation gelegt sondern auf das

Funktionieren des Zusammenspiels der Einzelteile und der im Verhältnis zum Erstellungsaufwand betrachteten Resultate des Workflows.

Das Beispielprogramm lädt zunächst die erstellten 3D-Daten und initialisiert die benötigten Engines und weitere Daten wie Timer, etc. Daraufhin befindet sich das Programm im Wartezustand. Über einen Button kann nun ein Wave-Sprachfile geladen und abgespielt werden. In diesem Fall wird anhand eines Timers alle 10ms eine Phonemanalyse des Signals vorgenommen und anschließend das 3D-Objekt zum entsprechenden Movestate gemorpht. Dies geschieht solange, bis das Signal zu Ende ist.

Zusätzlich ist eine Navigation in der Szene mit Hilfe der Maus und der Tastatur möglich.

Folgend ein Screenshot aus dem Beispielprogramm sowie ein Auszug aus der Dokumentation:

Abbildung 38: Screenshot der LipSync-Beispielanwendung Multi3D

Das **Fenster** zeigt die geladene Demoszene und den Charakter.

Oben links werden einige Informationen wie z.B. Tastaturbelegung, Navigation etc angezeigt.

***.wav laden** öffnet ein Dialogfenster zum Laden eines Wavefiles, das dann sofort abgespielt wird (muss unkomprimiertes *.wav, 44100Hz, 16Bit, mono sein)

***.wav stoppen** stoppt das aktuell abspielende Wavefile.

Phonemdaten öffnet ein Dialogfenster, das das Laden neuer Phonemerkennungsregeln ermöglicht.

Beenden schließt die Anwendung.

Genauere Spezifikationen und Anforderungen der Beispielanwendung sind in der Dokumentation auf der beliegenden CDRom beschrieben.

Um die Funktionalität des Systems unter realistischen Bedingungen zu prüfen, wurden Sprachdateien aus den Computerspielen „Gothic“ und „Morrowind“ extrahiert. Um zu überprüfen, wie das System mit unterschiedlichen Sprechern zurechtkommt wurden unterschiedliche männliche und weibliche Sprachfiles für den Testvorgang ausgewählt. Die Sprachdateien befinden sich ebenfalls auf der CDRom im Unterverzeichnis /audiofiles/.

Das Abspielen der Audiodaten in der Beispielanwendung bestätigt das Funktionieren des Systems und somit auch des Workflows. Einige Schwächen der Phonemalysefunktion beeinträchtigt die Mundbewegung noch negativ. Dies wirkt sich vor allem durch falsche Mundbewegungen aus, so dass manchmal statt einem /i/ ein /o/ erkannt wird. Durch eine Verbesserung der Erkennungsregeln kann dieses Problem allerdings beseitigt werden. Der Workflow an sich kann dadurch nicht in Frage gestellt werden.

Das Beispielprogramm liegt ebenfalls in kompilierter und unkompilierter Form auf der beiliegenden CDRom vor. Bitte beachten Sie die Voraussetzungen des Programms, die in der Dokumentation beschrieben sind.

6 Zum Schluss

6.1 Fazit

Die Auseinandersetzung mit der Thematik Lippensynchronisation in Computerspielen sowie der Entwicklung eines Automatisierungsworkflows und eines Echtzeit-LipSync-Systems waren sehr spannend und es gab einige Hürden zu überwinden.

Vor allem das Spracherkennungsmodul verlangte sehr viel Zeit und Experimentierfreude. Zwar gibt es in der Literatur einige Vorgehensweisen und Vorschläge zur Vorfilterung des Audiosignals, jedoch konnte ich zumeist keinerlei Verbesserungen nach den Implementierungen feststellen. Da die Implementierungen meist viel Zeit in Anspruch nahmen war dies doch etwas ernüchternd, weshalb ich schließlich den direkten Weg über die Merkmalsmodellierung wählte. Diese gestaltete sich ebenfalls nicht einfach, da es quasi keinerlei Anhaltspunkte über die Unterscheidung der Laute im Frequenzspektrum in der Literatur gab. Die Merkmalsunterscheidung ist deshalb von mir über Sonagrammauswertungen vorgenommen worden. Für eine computergestützte Auswertung blieb leider zu wenig Zeit. Letztendlich ist die auf die wenigen Erkennungsregeln reduzierte Phonemerkennung relativ zuverlässig geworden und wie in Kapitel 3.6 erwähnt gibt es noch jede Menge Spielraum für Verbesserungen. Für die beispielhafte Umsetzung reicht die entwickelte Phonemanalyse aus, obwohl eine hochprozentige Lippensynchronisation aufgrund der Schwächen der Phonemerkennung noch nicht möglich ist.

Im Nachhinein betrachtet war die Thematik sehr umfangreich, was vor allem mit der Thematik der Phonemerkennung zusammenhängt, bei der ich ursprünglich auf schon verfügbare Erkennungsmodule gehofft hatte. Die selbst entwickelte Phonemerkennung arbeitet nicht so genau wie ich dies ursprünglich gehofft hatte. Beispielsweise gibt es noch Probleme bei der Unterscheidung zwischen /u/ und /i/. In Anbetracht, dass vor allem dieses Thema für mich komplett Neuland war, und im Vergleich zu den Sprachsystemen aus Gothic und Morrowind bin ich mit dem Ergebnis zufrieden.

Vor allem das Zusammenspiel der einzelnen Module sowie der sehr zeiteffiziente Workflow lassen mich zu dem Fazit gelangen, dass das in dieser Arbeit entwickelte Echtzeit-Analyse-System für jedes Computerspiel eine Bereicherung darstellen könnte.

6.2 Schlusswort

Nach einem halben Jahr schließe ich nun diese Arbeit in der Hoffnung, dass das Lesen Ihnen ebenso Freude bereitet hat wie mir das Erarbeiten und Schreiben. Ich hoffe ich konnte Ihnen die Ideen und das Wissen abwechslungsreich und interessant vermitteln.

Sind wir gespannt, was die Zukunft in diesem Bereich bieten wird. Vielleicht geht die Entwicklung schneller vorwärts als erwartet und vielleicht stellt diese Arbeit ein Teilchen in dieser Entwicklung dar.

7 Anhang

7.1 Literaturhinweise und Internetlinks

Grundlagen der Spracherkennung:

Harry R. Ihm: "Das grosse Spracherkennungsbuch" (1999: Linguatec Sprachtechnologien)

L.R. Rabiner, B.H. Juang, Biing-Hwang Juang: „Fundamentals of Speech Recognition“ (1993: Prentice Hall)

L.R. Rabiner, R.W. Schafer „Digital Processing of Speech Signals“ (1978: Prentice Hall)

James Loton Flanagan, L.R. Rabiner „Speech Synthesis“ (1973: Dowden, Hutchinson & Ross)

A. Nejat Ince: „Digital Speech Processing, Speech Coding, Synthesis and Recognition“ (2002: Kluwer A.P.)

John Deller, John Proakis, John Hansen: „Discrete-Time Processing of Speech Signals“ (1999: John Wiley & Sons Inc)

Sonagramme:

http://www.phonetik.uni-muenchen.de/SGL/SGLKap1.html (28.10.1004)

Phonetik:

Peter Ladefoged: „A Course in Phonetics“ (2000: Thomson Learning)

John Clark, Colin Yallop: „An Introduction to Phonetics and Phonology“ (1995: Blackwell Publishers)

Magnus Petursson, Joachin M.H. Neppert: „Elementarbuch der Phonetik“ (2002: Buske)

Klaus J. Kohler: „Einführung in die Phonetik des Deutschen“ (1995: Erich Schmidt Verlag)

T.A. Hall: „Phonologie“ (2000: Gruyter)

Lippenanimation/Facial Animation:

Jason Osipa: „Stop Staring: Facial Modeling & Animation Done Right“ (2003: Sybex)

DirectX und C++:

Dirk Louis: „C/C++“ (2000: Markt & Technik)

David Scherfgen: „3D-Spieleprogrammierung“ (2003: Hanser)

Stefan Zerbst: „Spieleprogrammierung mit DirectX“ (2004: Markt & Technik)

Mark DeLoura, Dante Treglia u.a.: „Game Programming Gems“ (Charles River Media)

MaxScript:

Alexander Bicalho, Simon Feltman: „Mastering MAXScript and the SDK for 3DSMax“ (2000: Sybex)

7.2 Quellenangabe

Harry R. Ihm: "Das grosse Spracherkennungsbuch" (1999: Linguatec Sprachtechnologien)

L.R. Rabiner/B.H. Juang: „Fundamental of Speech Recognition“ (1993: Prentice Hall)

Tony Robinson: „Speech Analysis“ http://svr-www.eng.cam.ac.uk/~ajr/SpeechAnalysis/ (22.12.2004)

http://www.sengpielaudio.com (27.10.2004)

http://www.ub.uni-duisburg.de/diss/diss0126/Anhang.pdf (04.01.2005)

http://medi.uni-oldenburg.de/download/docs/lehre/kollm_phystechmed_akustik/aku4.pdf (04.01.2005)

http://lcavwww.epfl.ch/~minhdo/asr_project/asr_project.pdf (04.01.2005)

http://www.fbi.fh-darmstadt.de/~kkasper/asr5.pdf (04.01.2005)

http://www.fbi.fh-darmstadt.de/~kkasper/asr6.pdf (04.01.2005)

http://www.ci.tuwien.ac.at/~weingessel/docs/papers/diplomarbeit.ps.gz (04.01.2004)

http://www.informatik.htw-dresden.de/~iwe/Belege/Jahn/ (04.01.2004)

http://129.247.105.233/rpleger/ebuss.cgi/EbussWikipedia2.htm (04.01.2005)

http://www.annosoft.com/phoneset.htm (04.01.2005)

http://www.legamedia.net/legamall/2002/02-06/0206_weiss_oliver_sprachanalyse-methoden_01.php (04.01.2005)

7.3 Inhalt der CDRom

\3d-scene\:

3D-Daten der Demoszene (3D Studio Max R5)

\audiofiles\:

Sprach- und Testdateien (aus den Computerspielen Gothic und Morrowind)

\DA\:

schriftliche Ausarbeitung als *.doc, *.pdf, *.sxw

\directx9c\:

MS DirectX 9c Runtime

\dokus\:

Dokumentationen und Hilfedateien zu Quellcodes und Programmen

\exporters\:

MovestateExporter-Script für 3D Studio Max (MaxScript)

Panda-DirectX-Exporter v4.3.0.47 für 3D Studio Max R5 (Plugin)

\präsentation\:

Folien und Handzettel der Diplomarbeitspräsentation

\programs\multi3d\:

kompiliertes Beispielprogramm Multi3D (enthält Grafikengine und Ressourcen)

\programs\phonemeanalyser\:

kompiliertes Programm PhonemeAnalyser

\programs\soundengine\:

kompilierte Soundengine (sndmus.dll und sndmus.lib in Version 0.9.0.0)

\quellcodes\multi3d & gfx3d\:

MS Visual Studio 6 Projektfiles und C++ Quellcodes zur Beispielanwendung Multi3D inklusive Grafikengine-Quellcode Gfx3D

\quellcodes\phonemeanalyser\:

MS Visual Studio 6 Projektfiles und C++ Quellcodes zum Analyseprogramm PhonemeAnalyser

\quellcodes\soundengine\:

MS Visual Studio 6 Projektfiles und C++ Quellcodes zur Soundengine sndmus

\recherche\:

recherchierte Dokumente für die Ausarbeitung

\videos\:

Videos aus den Computerspielen „Mafia“, „Warcraft 3“, „Gothic 2“ und „Morrowind“

XVid-VideoCodec

7.4 Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Diplomarbeit selbständig und ohne unzulässige fremde Hilfe angefertigt habe. Alle verwendeten Quellen und Hilfsmittel sind angegeben.

Furtwangen, den 22. Februar 2005

Michael Brempel

7.5 Quellcodeanhänge

7.5.1 Movestate-Exporter-Script v1.2

Siehe Begleit-CDRom /exporters/MoveStateExporter.ms

7.5.2 Modifiers.h

Siehe Begleit-CDRom /quellcodes/multi3d & gfx3d/modifiers.h

7.5.3 SndMus.h

Siehe Begleit-CDRom /quellcodes/soundengine/sndmus.h