

Das MDLX-Format

Inhaltsverzeichnis

Das MDLX-Format.....	1
1.1 Kurzfassung.....	2
1.2 Englischsprachige Kurzfassung.....	3
1.3 Vorwort.....	4
1.3.1 Historisches/Verwendung.....	4
1.3.2 Binär und „von Menschen lesbar“.....	5
1.4 Spezifikationen.....	5
1.5 Das BLP-Format – Texturen.....	6
1.6 Software.....	7
1.7 Eigene Implementation.....	8
1.7.1 Das Projekt „wc3lib“.....	8
1.7.1.1 Plattformen und „Build Tool“.....	9
1.7.1.2 Externe Bibliotheken/Abhängigkeiten.....	10
1.7.1.2.1 Kern und Modul „mdlx“.....	10
1.7.1.2.2 Modul „blp“.....	10
1.7.1.2.3 Modul „editor“.....	10
1.7.1.3 Klassen, Funktionen und Header.....	11
1.7.1.3.1 Kern.....	11
1.7.1.3.2 Modul „blp“.....	12
1.7.1.3.3 Modul „mdlx“.....	12
1.7.1.3.4 Modul „editor“.....	12
1.8 Eigene Definition des MDLX-Formats.....	14
1.8.1 MDX-Format.....	14
1.8.2 Texturen.....	15
1.8.3 Materialien.....	16
1.8.4 Geosets.....	16
1.8.4.1 Vertices.....	16
1.8.4.2 Normale.....	16
1.8.4.3 Texturpunkte.....	16
1.8.4.4 Flächen.....	16
1.8.5 Kameras.....	17
1.8.6 Kollisionsformen.....	19
1.8.7 Typischer Aufbau einer Datei (MDL-Format).....	20
1.9 Ausblick.....	27
1.10 Quellen- und Literaturangaben.....	28

1.1 Kurzfassung

In dieser Arbeit möchte ich das sogenannte MDLX-Format vorstellen. Dabei handelt es sich genaugenommen um zwei verschiedene digitale Dateiformate, welche beide das Gleiche beschreiben: eine 3D-Grafik.

Selbstverständlich ist das Format bei Weitem nicht das Einzige seiner Art. Was das Ganze dennoch recht interessant macht, ist die Tatsache, dass es ein proprietäres und somit nicht offiziell öffentlich dokumentiertes bzw. spezifiziertes Format ist.

Ich versuche mit dieser Projektarbeit, es in einem von mir geschriebenen Programm schreib-, les- und vor allem darstellbar zu machen.

Die Projektarbeit umfasst dabei nur bestimmte Teile des Formats. Außerdem ist es mit der Projektarbeit möglich Bilddateien des BLP-Formats (BLP0/BLP1) einzulesen und darzustellen.

Die Besonderheit dabei liegt im direkten Einlese- und Darstellungsverfahren. Die Formate werden nicht zu erst mit vorhandenen Programmen in ein anderes Format konvertiert, sondern direkt eingelesen und nach Möglichkeit dargestellt.

Die Schwierigkeit liegt zum Einen im Verständnis der Formate und der dazugehörigen im Internet verfügbaren Informationen und zum Anderen in der Implementation.

Für das Einlesen, das Schreiben und die Darstellung der Formate wurde von mir die Programmiersprache C++, mit der sich vor allem abstrakter und effizienter Code schreiben lässt, der letztendlich in eine Maschinsprache kompiliert wird, um das Maximum an Leistung aus dem Prozessor herauszuholen.

Außerdem wurden verschiedene, ebenfalls in C++ geschriebene Bibliotheken für die Darstellung der grafischen Oberfläche, der 3D-Szene und verschiedene Grundalgorithmen (I/O, generische Container, Parallelisierung usw.) verwendet.

Im Gegensatz zu bereits vorhandener Software, wurde das ganze Projekt plattformübergreifend entwickelt. Es lässt sich daher sowohl unter Unix-artigen als auch unter Windows-artigen Betriebssystemen kompilieren und verwenden.

1.2 Englischsprachige Kurzfassung

This project work mainly aims at making parts of MDX format files to be displayed correctly by a C++ written "Open Source" program.

As MDX and MDL formats describe same properties of a 3d graphic they can be unified as "MDLX format" which is the actual topic of this project work.

The MDX supporting program itself can be seen as the practical part whereas the MDL format is used to describe all supported MDLX properties theoretically in the project work's documentation.

For better data abstraction, portability and performance the well known programming language C++ and some extensive cross-platform libraries have been used by the program. Since the MDLX format and its corresponding texture format BLP are not specified officially in public by their developer company Blizzard Entertainment I had to rely on some unofficial ones created by various people who had been interested in Blizzard's game "Warcraft III".

At its current state the program might be quite useless for usual artists since it doesn't support animations and editing of the MDLX 3d graphics but it's a rather strong, abstract and extensible framework for such tools.

As mentioned above portability is guaranteed by the used cross-platform libraries and the programming language standard itself. Therefore you theoretically should be able to compile and run the program on Unix as well as Windows systems.

1.3 Vorwort

1.3.1 Historisches/Verwendung

Die amerikanische Firma Blizzard Entertainment, die inzwischen ein Tochterunternehmen des Unterhaltungsriesen Vivendi ist, begann bereits 1998 mit der Entwicklung ihres ersten 3D-Computerspiels. Nach den Erfolgen der „Warcraft“- , „Starcraft“- und „Diablo“-Reihe, begann das Entwicklerstudio mit den Arbeiten an „Warcraft III: Reign of Chaos“¹. Da es ihr erstes 3D-Echtzeitstrategiespiel war, mussten sie sich erstmals für ein 3D-Grafikformat entscheiden. Anstatt auf ein bereits vorhandenes zu setzen, entwickelte Blizzard sein eigenes: das MDLX-Format.

„Warcraft III“ wurde ein Riesenerfolg; es erschien 2002 und bereits 2003 folgte die Erweiterung „Warcraft III: The Frozen Throne“, die ebenfalls ein sehr großer Erfolg wurde. Beide Spiele verwenden das von Blizzard entwickelte 3D-Grafikformat.

2004 erschien schließlich das bekannte und weitaus profitablere MMORPG „World of Warcraft“.

Sowohl das ältere Echtzeitstrategiespiel und seine Erweiterung als auch das MMORPG und all dessen Erweiterungen nutzen das MDLX-Format für 3D-Grafiken.

Während das von Blizzard entwickelte Texturformat BLP, welches ebenfalls zu dieser Arbeit gehört und später kurz beschrieben wird, auch in anderen Spielen genutzt wurde, blieb das MDLX-Format stets auf Blizzards Eigenentwicklungen beschränkt.

Inzwischen sind andere Formate vermutlich ausgereifter. Dazu zählen Formate wie das Blend-Format des populären und freien Programms „Blender“, sowie das 3ds-Format des kommerziellen Programms „3d Studio Max“. Jedoch werden auch heute noch viele 3D-Grafiken im MDLX- bzw. MDX-Format zur Verfügung gestellt, da das Computerspiel Warcraft III immer noch eine große Fan-Gemeinde hat.

Allein auf „The Hive Workshop“², einer der größten Warcraft-III-Fan-Websites werden ca. 4200³ unterschiedliche selbst erstellte 3D-Grafiken im MDX-Format zum Download angeboten und ca. 840⁴ Skins (Texturen).

Merkwürdigerweise verwendete Blizzard Entertainment mit ziemlicher Sicherheit „3d Studio Max“ der Version 4 für die Entwicklung von „Warcraft III“, da sie später auch ein Werkzeug für dieses veröffentlichten, das ihr eigenes Format unterstützt. Dennoch nutzten sie nicht das Format des Programms selbst, eventuell um herstellerunabhängig zu bleiben, vermutlich aber auch aus technischen Gründen.

1 <http://eu.blizzard.com/de-de/games/war3/>

2 <http://www.hiveworkshop.com/>

3 Stand: 2011-04-29

4 Stand: 2011-04-29

1.3.2 Binär und „von Menschen lesbar“

Wie bereits in der Kurzfassung beschrieben, handelt es sich eigentlich um zwei verschiedene Formate. Zum Einen existiert das MDL-Format, welches das von Menschen lesbare und unkomprimierte Gegenstück zum MDX-Format ist. Zum Anderen das erwähnte MDX-Format, welches binär ist. Beide haben ihren Zweck und somit eine Daseinsberechtigung. Ersteres lässt sich gut überblicken und schon mit einem einfachen Texteditor auf die Schnelle bearbeiten. Auch beim Einlesen bzw. „Parsen“ können Fehler leichter gefunden werden. Zweiteres ist dafür wesentlich kompakter und schneller einlesbar bzw. schreibbar. Es eignet sich also hervorragend für den Gebrauch im fertigen Spiel.

Das BLP-Format ist selbstverständlich ausschließlich binär, da die Daten bis auf den Header ein Bild beschreiben. Hierbei geht es ausschließlich um effiziente Kompression.

1.4 Spezifikationen

Die verfügbaren inoffiziellen Spezifikationen im Internet stammen hauptsächlich von Phil Laing⁵, Jimmy Campbell⁶ und Magnus Ostberg⁷.

Sie wurden durch Analyse konvertierter Modelle erstellt. Die Konvertierungen fanden dabei mit den „Blizzard Art Tools“ statt, einer offiziellen, von Blizzard entwickelten Erweiterung für „3ds Max“ 4 und 5.

Ein Teil der Spezifikation ergibt sich aus dem offiziellen Handbuch dieser „Art Tools“. Dort sind unter anderem gültige Textur- und Materialeinstellungen definiert.

Die inoffizielle Spezifikation des BLP-Formats (BLP0/BLP1) stammt ebenfalls von Magnus Ostberg und einigen Mitgliedern der Seite Wc3C.net⁸.

5 <http://kmkdesign.8m.com/downloads/>

6 <http://www.wc3c.net/tools/specs/NubMdxFormat.txt>

7 <http://home.magosx.com/index.php?topic=20.0>

8 <http://www.wc3c.net/showthread.php?t=103242>

1.5 Das BLP-Format – Texturen

Bei 3D-Grafikformaten ist es üblich, dass sogenannte Texturen auf Polygone (meist Dreiecke) gelegt bzw. gespannt werden, um dem Modell bzw. Netz eine Art Haut zu geben. Dies gleicht die physisch bedingten Limitierungen der Software aus und stellt dar, was mit einzelnen Punkten (Vertices) nur mit enormer Rechenleistung darstellbar wäre.

Diese Texturen sind also ganz gewöhnliche digitale Bilder, wie man sie z. B. im JPEG- oder dem PNG-Format abspeichern kann. Blizzard hat auch hier ein eigenes Binärformat entwickelt. Dateien dieses Formats haben normalerweise die Endung „.blp“, weshalb man vom BLP-Format spricht.

In der Computerspielreihe „World of Warcraft“ wird das Format „BLP2“ verwendet, welches ebenfalls auf Wikipedia⁹ beschrieben wird. Bei Warcraft III handelt es sich jedoch um das „BLP1“-Format (bzw. in dessen Betaphase noch das „BLP0“-Format, welches sich jedoch bis auf die Bezeichnung nicht von seinem Nachfolger unterscheidet).

Die inoffizielle Spezifikation¹⁰ des Formats ist recht einfach. Es existieren grundsätzlich zwei Arten der Kompression. Zum Einen eine gewöhnliche, konfigurierbare JPEG-Kompression (JFIF) und zum Anderen eine Palettenkompression, wie man sie z. B. vom GIF-Format her kennt (auch mit maximal 256 verschiedenen Farben und alternativ mit einem Alphakanal).

Außerdem erlaubt das Format die Speicherung von insgesamt bis zu 16 MIP-Maps. Der Begriff MIP-Map wird in der 3D-Grafik häufiger verwendet und bezeichnet lediglich eine kleiner skalierte Form einer Textur. Dabei werden Länge und Breite jeweils durch zwei geteilt, sprich, die Hälfte genommen. Die Speicherung von MIP-Maps erlaubt eine schnellere Skalierung bei großen Distanzsprüngen der Ansicht, da die Skalierung nicht dynamisch zur Laufzeit am Ausgangsbild vorgenommen werden muss, sondern bereits auf kleinere Versionen dessen zugegriffen werden kann.

Allgemein kann man sagen, dass es empfehlenswert ist, das BLP-Format ebenfalls zu implementieren, wenn man selbiges mit dem MDLX-Format tut, da es durchgehend von Blizzard-Ressourcen verwendet wird.

⁹ <http://en.wikipedia.org/wiki/BLP>

¹⁰ <http://www.wc3c.net/showthread.php?t=103242>

1.6 Software

Es existiert selbstverständlich inzwischen eine Reihe von Software, die das MDLX-Format unterstützt. Die eine besser, die andere schlechter. Nur ein Programm stammt offiziell vom Entwickler Blizzard: die sogenannten „Blizzard Art Tools“. Dieses ist jedoch nur mit den Programmen „3ds Max“¹¹ 4 und 5 kompatibel, welche zum Einen kostenpflichtig und proprietär und zum Anderen veraltet sind.

Inzwischen wurden einige Konvertierungswerkzeuge für neuere 3D-Grafikprogramme entwickelt. Diese, wie auch die meisten anderen Werkzeuge, stammen von Hobby-Entwicklern. Jedoch wurden sämtliche Programme für eine Windows-Umgebung entwickelt und sind teilweise nicht einmal „Open Source“.

Eine Link-Liste vorhandener Software zur Bearbeitung des MDLX-Formats:

- Texture Pather: <http://www.wc3c.net/showthread.php?t=85245>
- MDX Squisher: <http://www.wc3c.net/showthread.php?t=83416>
- Warcraft 3 Viewer: <http://www.wc3c.net/showthread.php?t=82558>
- BlizzImporter: <http://www.wc3c.net/showthread.php?t=92856>
- MDLX converter: <http://www.wc3c.net/showthread.php?t=91130>
- War3 Model Editor (von Magnus Ostberg):
<http://home.magosx.com/index.php?topic=6.0>
- MdxLib (von Magnus Ostberg): <http://home.magosx.com/index.php?topic=20.0>

Auch für das BLP-Format existiert bereits eine Reihe von Software:

- BLP Laboratory: <http://www.wc3c.net/showthread.php?t=107500>
- BLPaletter: <http://www.wc3c.net/showthread.php?t=86878>

¹¹ <http://www.autodesk.de/>

1.7 Eigene Implementation

1.7.1 Das Projekt „wc3lib“

Im August 2009 begann ich mit dem Projekt „wc3lib“ (engl. für „Warcraft-III-Bibliothek“). Ursprünglich bestand meine Absicht bei der Erstellung des Projekts darin, die Formate des Spiels besser kennen zu lernen und eigene Werkzeuge dafür zu entwickeln. Tatsächlich wollte ich zunächst das MDLX-Format konvertierbar machen, jedoch kam außer einem besseren Verständnis des Formats und der Erstellung zahlreicher in C++ geschriebener Klassen nicht viel mehr dabei heraus als dass das Format teilweise einlesbar wurde. Die Entwicklung der MDLX-Unterbibliothek wurde daraufhin von mir zunächst auf Eis gelegt, da mir das BLP- (Texturformat) und das MPQ-Format (Archivformat) als notwendiger erschienen.

Ich schrieb zwischen 2009 und 2010 eine Menge Code, der hauptsächlich für Ansätze gedacht war, las die Spezifikationen der unterschiedlichen Formate und versuchte diverse externe Bibliotheken einzubinden, um weniger auszuführenden Code selbst schreiben zu müssen.

Schließlich hatte ich ein einfaches Konvertierungswerkzeug geschrieben, das die unterschiedlichen Formate Blizzard's einlesen und schreiben konnte, falls implementiert.

In den Sommerferien 2010 setzte ich mich schließlich wieder an das MDLX-Format und begann allmählich den eigentlichen Aufbau einer MDL-Datei und die Funktion der einzelnen Elemente zu verstehen und damit herum zu experimentieren bzw. nach vielen Beschreibungen und Anleitungen dazu im Internet zu suchen. Beim besseren Verständnis des Aufbaus und der Darstellung halfen mir auch bereits vorhandene Programme und deren Quell-Code, wie z. B. der des Programms „War3 Model Editor“¹².

Für die Darstellung der 3D-Grafikformate Blizzard's werden neben dem Kern der Bibliothek die Module „mdlx“, welches für das Einlesen und Schreiben der Formate MDL und MDX zuständig ist, „blp“, welches für das Einlesen und Schreiben des Formats BLP zuständig ist und „editor“, welches für die Darstellung in grafischen Benutzeroberfläche zuständig ist, benötigt.

Letzteres Modul diente zunächst als testweises Modul zur Darstellung von Blizzard's Texturformat (BLP), entwickelte sich aber mit der Zeit tatsächlich immer mehr zu einer Emulation von Blizzard's eigenem Karteneditor, einem Werkzeug für die Bearbeitung von Spielinhalten Warcraft III's, welche auf den anderen formatspezifischen Modulen der Bibliothek aufbaut.

Während der Kern, das „blp“- und das „mdlx“-Modul, hauptsächlich auf die Standard- und „Boost“-Bibliotheken setzen und damit relativ wenig Abhängigkeiten haben, benötigt das „editor“-Modul für die grafische Benutzeroberfläche die Bibliotheken des „Qt“- und des „KDE“-Projekts und für die Darstellung der 3D-Grafiken die Bibliothek „OGRE“.

12 <http://home.magosx.com/index.php?topic=6.0>

1.7.1.1 Plattformen und „Build Tool“

Die Bibliothek „wc3lib“ ist als objektorientierte, plattformübergreifende C++-Bibliothek gedacht. Das bedeutet, dass sie sich auf möglichst vielen verschiedenen Plattformen (Betriebssystemen/Architekturen) kompilieren und verwenden lassen soll.

Zum Einen wird dies durch die Verwendung der Standardbibliothek von C++ und externer, ebenfalls plattformübergreifender Bibliotheken gewährleistet und zum Anderen mit einem plattformübergreifenden Automatisierungs-Build-Tool. Ein Build-Tool ermöglicht in der Regel die Abarbeitung verschiedener Kompilierungsschritte anhand bestimmter Dateien. Das wohl bekannteste ist „Make“ bzw. „GNU Make“. Damit der Entwickler das sogenannte „Makefile“, welches die benötigten Kompilierungsregeln enthält, nicht selbst schreiben und dabei auf alle unterstützten Plattformen und deren Eigenheiten achten muss, existiert eine Reihe von Automatisierungswerkzeugen wie z. B. „qmake“, „automake“ (veraltet), „jam“ oder „CMake“.

Diese Werkzeuge ermöglichen die Generierung von „Makefiles“ oder Projektdateien für bestimmte Entwicklungsumgebungen anhand einfacher Regeln und Befehle in Abhängigkeit des aktuellen Betriebssystems auf welchem kompiliert wird. Beim Projekt „wc3lib“ habe ich auf das Werkzeug „CMake“¹³ gesetzt, das unter anderem vom KDE-Projekt verwendet wird.

„CMake“ bietet eine recht einfache, jedoch teils ungewohnte, eigene Skriptsprache an, mit der Projekteigenschaften spezifiziert werden können.

Der Benutzer der Bibliothek muss mittels „CMake“ zunächst alle „Makefiles“ auf seiner Plattform erzeugen und kann die Bibliothek danach mit einem einfachen „make“-Befehl kompilieren. Mit „make install“ lässt sie sich in der Regel installieren.

„CMake“ wurde gewählt, da es relativ flexibel ist. Mit dem Programm lassen sich ebenfalls Projektdateien für bekannte IDEs erzeugen und zudem sind Versionen (auch mit grafischer Oberfläche) für die meisten Plattformen verfügbar.

13 <http://cmake.org/>

1.7.1.2 Externe Bibliotheken/Abhängigkeiten

Es existiert eine Reihe von benötigten externen Bibliotheken, die vom Projekt „wc3lib“ genutzt werden, um nicht sämtlichen Code selbst schreiben bzw. pflegen zu müssen. Dabei wurde darauf geachtet, dass sämtliche Abhängigkeiten auch unter anderen Plattformen wie z. B. Nicht-Unix-Systemen zur Verfügung stehen und selbstverständlich „Open Source“ und kostenlos sind.

1.7.1.2.1 Kern und Modul „mdlx“

Neben einem C++-fähigen Compiler und den Werkzeugen „CMake“ und „Make“ bzw. einer von „CMake“ unterstützten IDE, benötigt der Anwender der Bibliothek weitere externe Bibliotheken, deren Funktionalität von der „wc3lib“ genutzt wird.

Allen voran stehen die „Boost C++ Libraries“¹⁴, die eine Reihe von Kernmodulen für C++ anbieten und es um im ISO-Standard fehlende Elemente erweitern. Sie sind selbstverständlich ebenfalls plattformübergreifend.

Leider dauert die Entwicklung des C++-Standards länger als die vieler anderer Programmiersprachenstandards. Das „Boost“-Projekt hat es sich zum Ziel gemacht, das anzubieten und zu entwickeln, was dem Standard (noch) fehlt.

Im Entwicklerteam sitzen viele der erfahrensten C++-Entwickler überhaupt, die teilweise selbst bei der Erstellung des Standards mitbestimmen. Teile des „Boost“-Projekts werden voraussichtlich im kommenden C++-Standard sogar fast eins zu eins übernommen werden.

1.7.1.2.2 Modul „blp“

Da Blizzard's BLP-Format JPEG-Kompression unterstützt, wird für das blp-Modul eine JPEG-Bibliothek¹⁵ benötigt. Diese wird auf Anforderung dynamisch zur Laufzeit geladen (Klasse „LibraryLoader“).

1.7.1.2.3 Modul „editor“

Des Weiteren werden für die grafischen Benutzeroberflächen die Bibliotheken des plattformübergreifenden, quelloffenen Frameworks „Qt“¹⁶ und des darauf aufbauenden, ebenfalls quelloffenen und inzwischen auch teilweise plattformübergreifenden Projekts „KDE“¹⁷ verwendet. Beide Projekte sind wie bereits erwähnt „Open Source“ und kostenlos verwendbar (durch die Lizenzierung von „Qt“ nur für nicht kommerzielle Projekte).

Für die eigentliche Darstellung dreidimensionaler Körper, welche in die grafische Benutzeroberfläche integriert ist, wird die plattformübergreifende Rendering Engine „OGRE“¹⁸ genutzt.

14 <http://www.boost.org/>

15 <http://jpegclub.org/>

16 <http://qt.nokia.com/>

17 <http://kde.org/>

18 <http://www.ogre3d.org/>

1.7.1.3 Klassen, Funktionen und Header

Da C++ eine objektorientierte Programmiersprache ist, liegt es Nahe, eine Reihe von Klassen zu definieren, mit deren Hilfe man eine Art Modell des Formats erhält und dieses darstellen kann. Diese Abstraktion vereinfacht die Implementation und vor allem die Erweiterung und macht das Programm übersichtlicher und weniger fehleranfällig.

Man sollte dabei beachten, dass es sich bei der Implementation des MDLX-Formats um zwei voneinander getrennte, aufeinander aufbauende Module handelt. Im Modul „mdlx“ befinden sich ausschließlich Klassen, um das Format einzulesen und zu schreiben. Dagegen befinden sich im Modul „editor“ wiederum ausschließlich Klassen, um das Format darzustellen und optional per grafische Benutzeroberfläche editierbar zu machen.

Die Editierbarkeit gehört jedoch nicht zum Umfang der Projektarbeit und bezieht sich auf die zukünftigen Erweiterungsmöglichkeiten des Projekts.

Jedes Modul besitzt eine Header-Datei mit der Bezeichnung „platform.hpp“. In dieser werden grundlegende Datentypen deklariert, die für das jeweilige Format spezifiziert wurden.

1.7.1.3.1 Kern

Der Kern der Bibliothek „wc3lib“ enthält einige Basisklassen und -funktionen, die Implementationen von Formaten vereinfachen sollen. Hierzu zählt die abstrakte Klasse „Format“, welche abgeleitete Klassen dazu zwingt, Lese- und Schreibelementfunktionen zu implementieren und diese mit der Serialisierungsschnittstelle der „Boost“-Bibliotheken verbindet. Des Weiteren erlaubt die statische Klasse „LibraryLoader“ das dynamische Laden von Bibliotheken („Shared Objects“) zur Laufzeit, was oftmals die einfache Austauschbarkeit einer Formatbibliothek durch die Benutzerkonfiguration erlaubt. In der Header-Datei „utilities.hpp“ sind nützliche Lese- und Schreibfunktionen für Binärformate, sowie eine Textausgabefunktion für Größen deklariert. Allesamt sind als Templates deklariert und daher für fast beliebige Datentypen zu gebrauchen.

Die Klasse „Exception“ stellt die oberste Basisklasse für sämtliche geworfene Ausnahmen der Bibliothek dar.

In der Header-Datei „internationalisation.hpp“ bzw. ihrer Aliasdatei „i18n.hpp“ werden sämtliche Header-Dateien eingebunden und Makros definiert, die zur Internationalisierung der Ausgabe benötigt werden.

Sämtliche weitere Header-Dateien dienen der Einbindung der einzelnen Module der Bibliothek. Mit der Header-Datei „wc3lib.hpp“ kann die gesamte Bibliothek eingebunden werden. Mit der Header-Datei „core.hpp“ dagegen nur der gesamte Kern der Bibliothek.

1.7.1.3.2 Modul „blp“

Das Modul „blp“ enthält Klassen zum Einlesen und Schreiben des BLP-Formats (Texturen). Es beinhaltet lediglich eine einzige Klasse namens „Blp“, welche eine einzelne, dekomprimierte BLP-Textur mit einer variablen Anzahl von „MIP Maps“ darstellt.

Für die Deklaration der primitiven Datentypen existiert die Datei „platform.hpp“.

1.7.1.3.3 Modul „mdlx“

Die Hauptklasse des MDLX-Moduls ist selbstverständlich die Klasse „Mdlx“. Sie stellt ein einzelnes MDLX-Modell dar und beinhaltet sämtliche Elemente als Eigenschaften.

Wie in allen formatbezogenen Modulen der Bibliothek existiert in diesem Verzeichnis eine Header-Datei namens „platform.hpp“. In ihr sind global zu verwendende Standardtypen deklariert, die teils eine exakt definierte Größe haben müssen. Diese ist durch die Spezifikation vorgegeben. Zu ihnen gehören unter Anderem die Struktur „VertexData“, sowie die enums „LineType“ und „ReplaceableId“.

Das Format wird fast eins zu eins anhand der Spezifikation durch die Klassen dargestellt. Es wurde an einigen Stellen abstrahiert wodurch die Klassen „MdlxProperty“, „MdlxBlock“, „MdlxGroupBlock“, „MdlxGroupBlockMember“, „MdlxScalings“, „MdlxTranslations“, „MdlxRotations“, „MdlxAlphas“, „Node“ und „Object“ entstanden sind, die eine Basis für bestimmte Eigenschaften darstellen.

Sämtliche Klassen, die einen neuen binären Block des „MDX“-Formats beschreiben, erben von der abstrakten Klasse „MdlxBlock“. Für Objekt- und Knoten-Elemente existieren die Klassen „Object“ und „Node“. Es gilt zu beachten, dass viele Klassen für die Projektarbeit nicht benötigt werden, da es sich um erweiterte Elemente des Formats handelt (wie z. B. Partikel-Emitter). Diese existieren der Vollständigkeit halber, da ich zunächst mit einem Leseprogramm für das Format begann.

1.7.1.3.4 Modul „editor“

Dieses Modul macht hauptsächlich Gebrauch der Frameworks „Qt“, „KDE“ und „OGRE“. Die zentrale Klasse „Editor“ repräsentiert einen Karteneditor des Spiels „Warcraft III: The Frozen Throne“. Dieser Karteneditor enthält wiederum Module, die teilweise eigene Editoren sind.

Dazu zählt auch der benötigte Modelleditor, der in dieser Projektarbeit lediglich als Modellbetrachter interpretiert werden sollte, da die Editierbarkeit nicht zum Umfang der Projektarbeit gehört.

Zum Einen erbt die Klasse „ModelEditor“ von der abstrakten Klasse „Modul“, wie auch alle anderen Module des zentralen Editors, und zum Anderen enthält sie eine Instanz der Klasse von der Klasse „ModelView“ abgeleiteten Klasse „ModelEditorView“, welche die eigentliche Implementation der Darstellung dreidimensionaler Objekte ist.

Die Klasse „ModelView“ verbindet die beiden Frameworks „Qt“ (grafische Benutzeroberfläche) und „OGRE“ (3D-Rendering-Engine), indem sie die Einbettung einer 3D-Anzeige als Widget erlaubt. Es wurde von mir eine einfache Maussteuerung anhand von Beispielen aus dem Internet¹⁹ implementiert.

¹⁹ <http://qt-apps.org/content/show.php?action=content&content=92912>

Daher lässt sich die Ansicht verschieben und drehen.

Außerdem bietet die Klasse „ModelView“ Funktionen zum Laden bzw. Darstellen von MDLX-Instanzen an. Diese werden über die Klasse „OgreMdlx“ mit Funktionen des „OGRE“-Frameworks dargestellt. Die Klasse „ModelEditorView“ ist eine speziellere Form eines Modellbetrachtungswerkzeuges, da sie zusätzlich den sogenannten „Hit Test“ erlaubt. Dieser wurde ebenfalls anhand von Beispielen aus dem Internet²⁰ implementiert.

Die Klasse „OgreMdlx“ erzeugt aus den Informationen einer „Mdlx“-Instanz diverse dreidimensionale Objekte über „OGRE“-Klassen und versucht die 3D-Grafik möglichst so darzustellen wie in Blizzard's originalem Spiel. Um sie entwickeln zu können, musste ich mich daher mit der Bibliothek „OGRE“ und dem Format „MDLX“ und deren Unterschieden bei der Beschreibung einer 3D-Grafik beschäftigen.

Für die Auswahl der Team-Farbe und den Team-Schein existiert die Klasse „TeamColorDialog“. Beide Eigenschaften können in einem Modelleditor für alle Modelle über Menüaktionen bestimmt werden.

Die abstrakte Klasse „Resource“ erlaubt es dem Editor alle geladenen Ressourcen zu überwachen und zudem, da er von der Klasse „MpqPriorityList“ erbt, eine Überwachung der Verzeichnisse geladener Ressourcen. Wird z. B. eine externe MDX-Datei zur Anzeige geladen, die Texturdateien (z. B. BLP-Dateien) verwendet, die sich relativ zur Datei, in deren Verzeichnis befinden, ermittelt dies der Editor über seine von der Klasse „MpqPriorityList“ geerbte Elementfunktion „findFile“ automatisch, die auch die relativen Dateipfade in den Verzeichnissen aller hinzugefügten Ressourcen überprüft und kann die Texturen laden.

Die Klasse „MpqPriorityList“ dient also dazu mehrere Verzeichnisse bzw. Archive als Quelle in Betracht zu ziehen und je nach Priorität abzusuchen. Dabei kann auch die Sprache der Datei eine Rolle spielen, da Blizzard's Archivformat „MPQ“ unterschiedliche Dateiversionen für verschiedene Sprachen mit dem selben Dateipfad erlaubt.

Des Weiteren existiert ein Plugin für Blizzard's eigenes Texturformat „BLP“, das durch die Klassen „BlpIOPlugin“ und „BlpIOHandler“ in „Qt“-Oberflächen als reguläres Bildformat verwendbar gemacht wird.

Grafische Oberflächen werden durch den „Designer“, ein zu „Qt“ zugehöriges Werkzeug, umgesetzt. Damit können XML-Dateien der Widgets erzeugt werden, die wiederum mit dem Programm „uic“ („User Interface Compiler“) zu C++-Headern konvertiert werden können. Die eigene Klasse des Widgets erbt dann zusätzlich (möglich durch Mehrfachvererbung) von der generierten.

Die XML-Dateien befinden sich im Verzeichnis „src/editor/ui“.

Da „KDE“ auf „Qt“ aufbaut, bietet es eine ganze Reihe von eigenen Vorlageformen für den „Designer“.

20 <http://www.ogre3d.org/tikiwiki/tiki-index.php?page=Raycasting+to+the+polygon+level>

1.8 Eigene Definition des MDLX-Formats

Im Folgenden möchte ich eine eigene Definition des Formats bzw. der Teile, die in der Projektarbeit behandelt werden, wiedergeben. Es soll keine exakte Spezifikation mit Byte-Größen und Begriffen darstellen, sondern eine verständliche Erklärung und vor allem das „Warum?“ beantworten.

1.8.1 MDX-Format

Beim binären MDX-Format ist alles in Blöcke eingeteilt, die mit einem vier Byte langen Schlüssel beginnen, der im ASCII-Format normalerweise die Abkürzung für den Blockbezeichner ist.

Innerhalb der Blöcke werden zu Beginn entweder Bytegrößen oder eine bestimmte Anzahl angegeben. Besitzen die im Block enthaltenen Datensätze eine konstante Größe, so wird die Anzahl angegeben, ansonsten eine Bytegröße. Dies ermöglicht theoretisch ein sehr schnelles Einlesen von Daten mit konstanter Größe. Datensätze die keine konstante Größe besitzen sind in der Regel Objekte mit animierbaren Eigenschaften.

Bei den Bytegrößen handelt es sich manchmal auch um inklusive Bytegrößen (die Größe der Bytegröße wird miteinbezogen).

Welchen Sinn dies hat, ist leider der Spezifikation nach nicht ersichtlich.

1.8.2 Texturen

Texturen bestehen in der Regel aus einer externen Bilddatei, also einer 2d-Grafik und einigen Optionen. Eine Textur wird später auf eine bestimmte Art „gefaltet“ und auf die Flächen des Modells gelegt. Sie stellt praktisch die Bemalung oder Haut des Modells dar.

Für die Texturen des MDLX-Formats gelten einige Besonderheiten, die jedoch ausschließlich auf Blizzard's eigene Implementation zurückzuführen sind. Als Standardbildformat wird Blizzard's eigenes „BLP“-Format verwendet, welches das sogenannte „MIP Mapping“ erlaubt.

Als zwingende Eigenschaft muss selbstverständlich der Dateipfad der Textur (maximal 256 Zeichen) angegeben werden, welcher sowohl absolut als auch relativ zur Modelldatei bzw. zum geöffneten Datenarchiv sein sollte.

Texturen dürfen maximal 512x512 Pixel groß sein und ihre Länge und Breite müssen eine Zweierpotenz sein, was vermutlich die mathematischen Berechnungen erheblich vereinfacht und zudem die Berechnung der sogenannten „MIP Maps“, die in BLP-Dateien gespeichert werden, um den Texturfilterprozess zu beschleunigen, da so verkleinerte Versionen der Textur einfach aus der Datei geladen werden können, ermöglicht. Das Computerspiel „Warcraft III“ unterstützt bilineare Texturfilterung.

Zudem darf das Verhältnis zwischen Länge und Breite und umgekehrt maximal 8 zu 1 sein.

Des Weiteren können ausschließlich entweder 24 oder 32 Bit als Farbtiefe der RGBA-Farben verwendet werden.

Der Alphakanal hat bei bestimmten Materialeinstellungen eine besondere Bedeutung (Transparenz/Verlauf oder Team-Farbe/Leuchten).

Schwarz (0x00) beim Alphakanal bedeutet Transparenz und weiß (0xFF) undurchsichtig.

Es gibt eine spezielle Form von Textur, die etwas Allgemeineres als eine zum Modell zugehörige Bilddatei beschreibt. Dabei handelt es sich um sogenannte ersetzbare Texturen. Diese werden zur Spielzeit im Computerspiel „Warcraft III“ durch entsprechende vorgegebene Texturen ersetzt. Daher wird bei ihnen in der Regel ein leerer Dateipfad angegeben.

Zum Beispiel kann es sich bei einer ersetzbaren Textur um die Team-Farbe einer Einheit des Spiels handeln. Folglich wird die entsprechende Team-Farbe der Einheit zur Spielzeit als Textur geladen.

Es handelt sich also um eine Art dynamische Textur.

1.8.3 Materialien

Materialien bestehen aus mehreren Ebenen, denen jeweils eine bestimmte Textur zugeordnet ist. Ein Material kann daher aus mehreren sich gegenseitig überlappenden Texturen bestehen.

Wie genau diese Überlappung stattfindet, wird in den Definitionen der einzelnen Ebenen bestimmt. Zum Beispiel kann der Alphakanal einer Textur als Transparenzwert verwendet werden, was eine gewisse Durchsichtigkeit (z. B. zu einer anderen Textur) erlaubt. Außerdem kann eine Ebene entweder beidseitig oder einseitig mit der Textur bedeckt sein.

Materialien bzw. deren Ebene bestimmen ebenfalls die Licht- und Schatteneinwirkungen auf die zugehörigen Texturen.

1.8.4 Geosets

Geosets dienen der Unterteilung eines Modells in mehrere zu rendernde Abschnitte. Daher kann jedes Geoset genau ein Material verwenden, welches nochmals in mehrere Ebenen unterteilt wird.

Vermutlich wurde diese Unterteilung der Geschwindigkeit und der Vergabe von einzelnen Prozessen halber so festgelegt, da alle Flächen (bzw. Dreiecke) eines Geosets zur selben Zeit gerendert werden.

Die Unterteilung wird normalerweise automatisch vom Modellbearbeitungswerkzeug vorgenommen.

Ein Geoset kann die folgenden Elemente enthalten:

1.8.4.1 Vertices

Ein Vertex ist ein einzelner Punkt bzw. Vektor im dreidimensionalen Raum. Er besitzt daher drei Koordinaten. Vertices dienen als Bezugspunkte des Geosets und dessen Flächen.

1.8.4.2 Normale

Beim MDLX-Format besitzt normalerweise jeder Vertex eine entsprechende Normale bzw. einen weiteren Vertex, der die Werte der Normale enthält. Normale werden ebenfalls in der Datei gespeichert und dienen der einfacheren Berechnung von Ebenen bzw. den Flächen des Geosets.

1.8.4.3 Texturpunkte

Texturpunkte bzw. Textur-Vertices sind zweidimensionale Bezugspunkte auf der vom Geoset verwendeten Textur. Jeder dreidimensionale Vertex des Geosets besitzt normalerweise einen zugehörigen Texturpunkt zur Orientierung der über das Geoset gezogenen Flächen.

1.8.4.4 Flächen

Flächen bzw. Primitive können beim MDLX-Format ausschließlich Dreiecke sein, auch wenn das Format noch Platz für andere Flächentypen lässt. Es werden immer drei Vertices anhand ihrer Indices zu einem Dreieck zusammengefasst.

1.8.5 Kameras

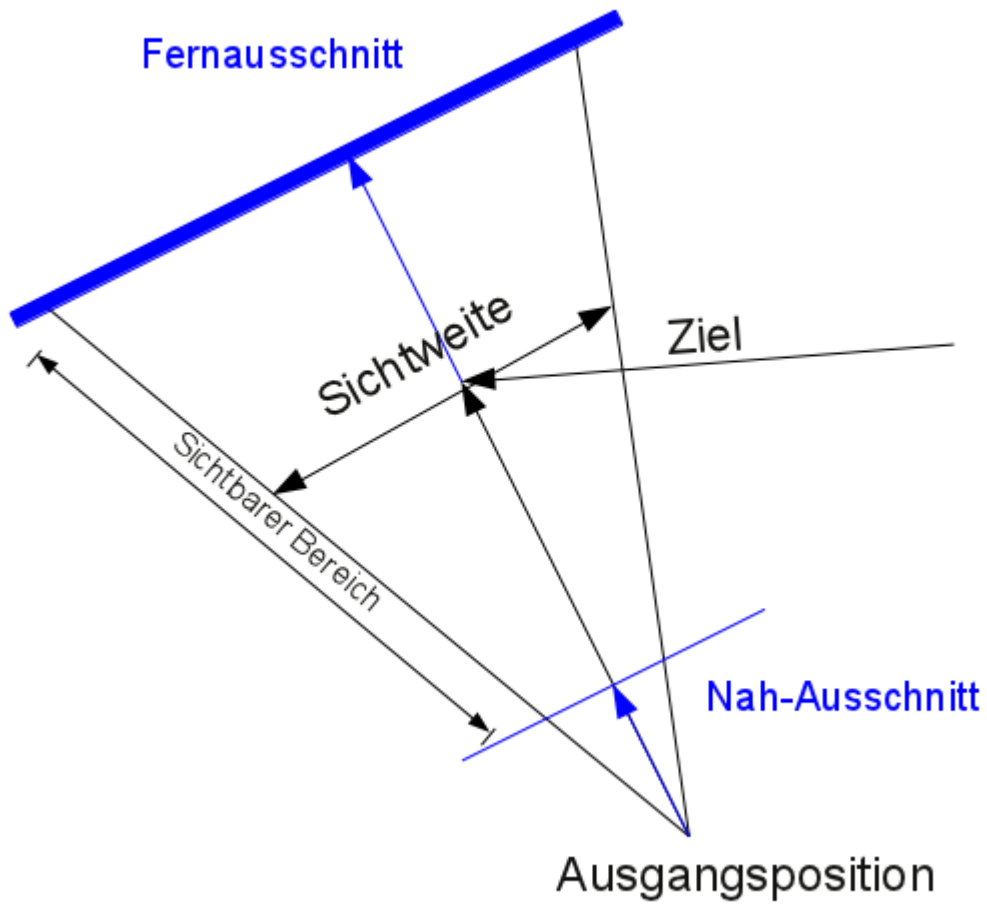
Kameras dienen der Betrachtung eines Modells während des Spiels. Für Einheiten (steuerbare Spielfiguren) existiert dazu meist eine weitere Modelldatei für das Portrait mit der Endung „_Portrait“ im Dateinamen, da im Portrait weit weniger von der Einheit zu sehen ist als auf dem Spielfeld. Diese wird während des Spiels automatisch geladen und am unteren Bildschirmrand angezeigt wenn solange Einheit ausgewählt ist:



Bei Objekten, die im Portrait fast eins zu eins angezeigt werden, existiert nur eine Version des Modells, die die Portraitkameras enthält.

Kameras haben eine Ausgangsposition (die des Betrachters) und ein Ziel. Beide Eigenschaften werden als Vertex angegeben. Zudem wird als Fließkommazahl die sogenannte Sichtweite angegeben, die die Ausdehnung der Ansicht in die Ferne angibt.

Die beiden Fließkommazahlenwerte für den Fern- und den Nahausschnitt der Ansicht geben an, wie weit in die Ferne und wie nah das Modell betrachtet werden kann:

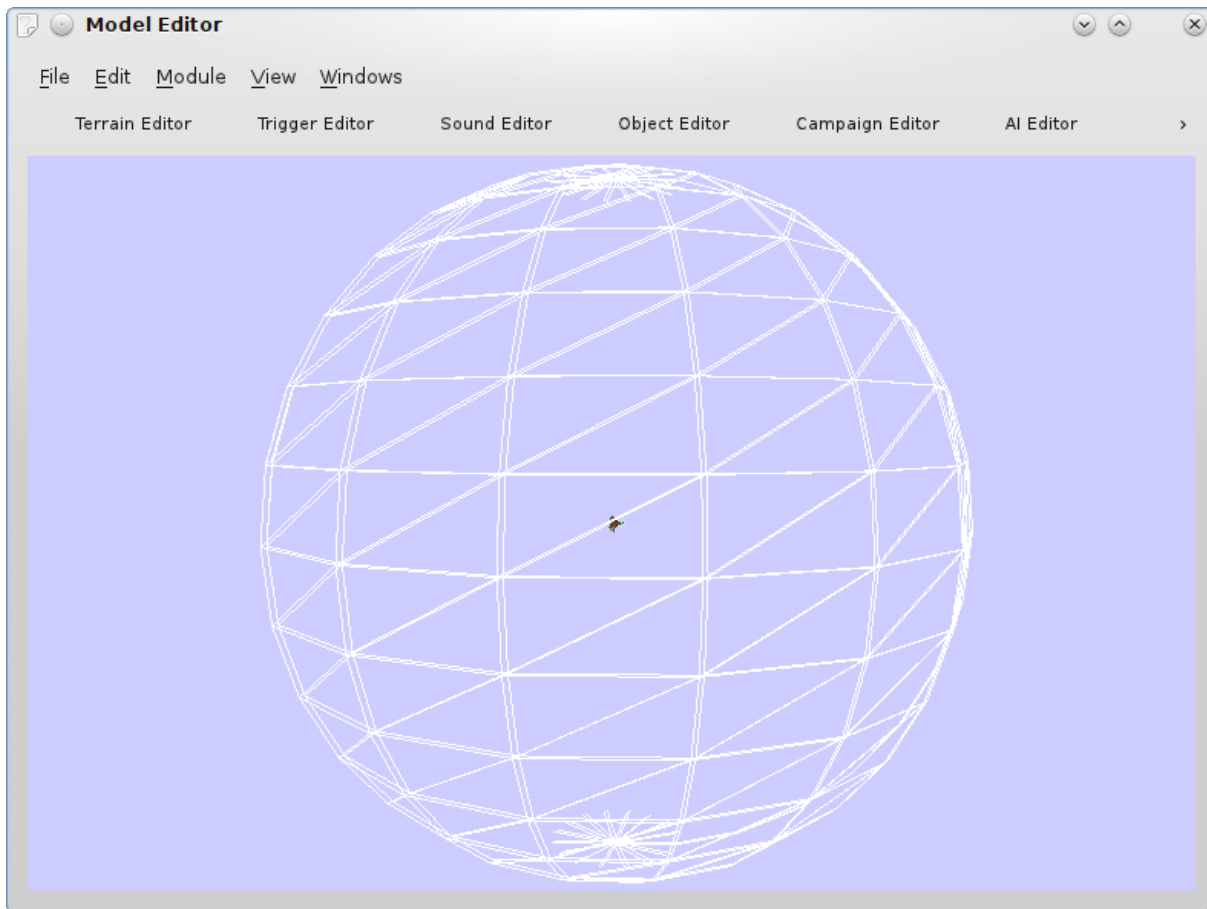


1.8.6 Kollisionsformen

Kollisionsformen dienen dem sogenannten „Hit Test“. Dieser von Blizzard Entertainment verwendete Begriff steht für die Abfrage, ob der Mauscursor ein dreidimensionales Modell berührt. Dies ist während des Spiels von Bedeutung, da der Spieler die Spielfiguren mit dem Mauscursor auswählt und steuert.

Blizzard Entertainment verwenden diese Extraformen nach eigener Beschreibung daher, da tatsächliche Kollisionsabfragen mit dem eigentlichen Modell sehr rechenintensiv sein können. Da Kollisionsformen lediglich aus einem Kasten oder einer Kugel bestehen können, wird der „Hit Test“ wesentlich vereinfacht. Für eine Kugel werden ein Vertex und ein Radius benötigt, für einen Kasten dagegen zwei Vertices.

Hier eine Darstellung eines Rohmodells mit sichtbarer Kollisionsform:



1.8.7 Typischer Aufbau einer Datei (MDL-Format)

Als Beispiel habe ich ein Modell aus dem Computerspiel „Warcraft III: The Frozen Throne“ ausgewählt. Es handelt sich um dunkles Portal, das eine spezielle Bedeutung im Spiel hat. Es eignet sich hervorragend für die Veranschaulichung, da es auch ohne Animationen, also als statisches Objekt verwendbar ist und sich seine Portraitkameras in derselben Datei befinden.

Zunächst ein Bild aus dem Originalspiel:

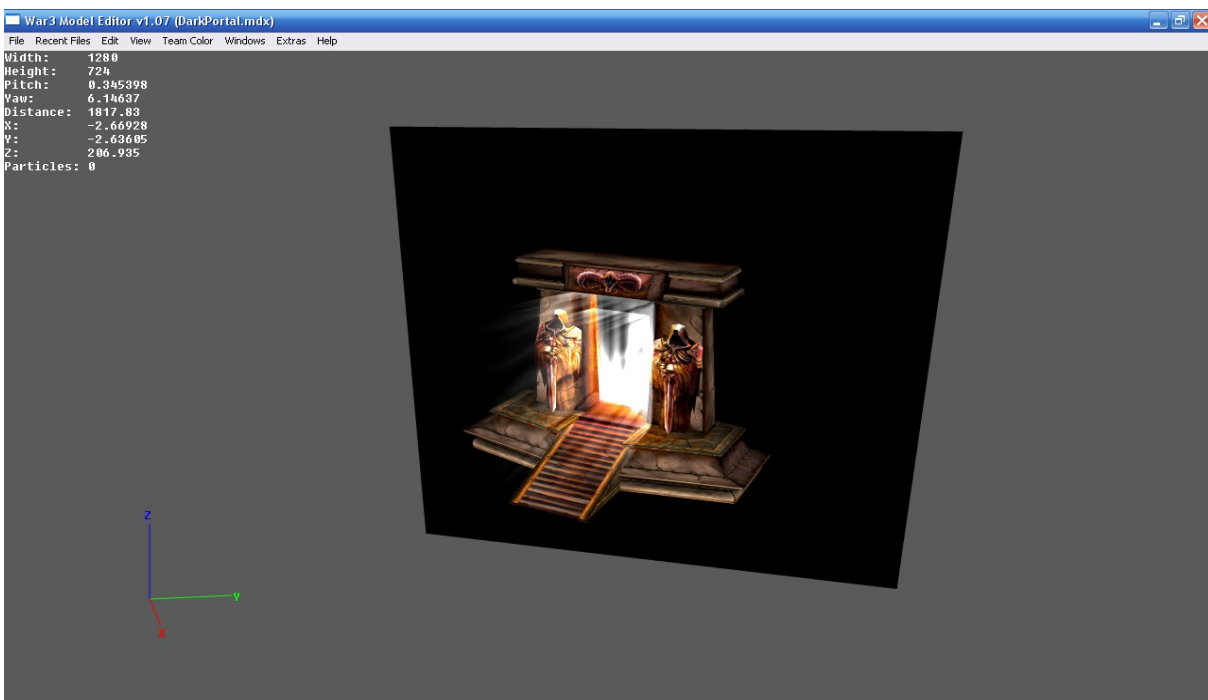


Das Leuchten ist Teil der Animation und wird daher in der Projektarbeit nicht dargestellt.

Ohne Animationen sieht das Portal folgendermaßen im Spiel aus:

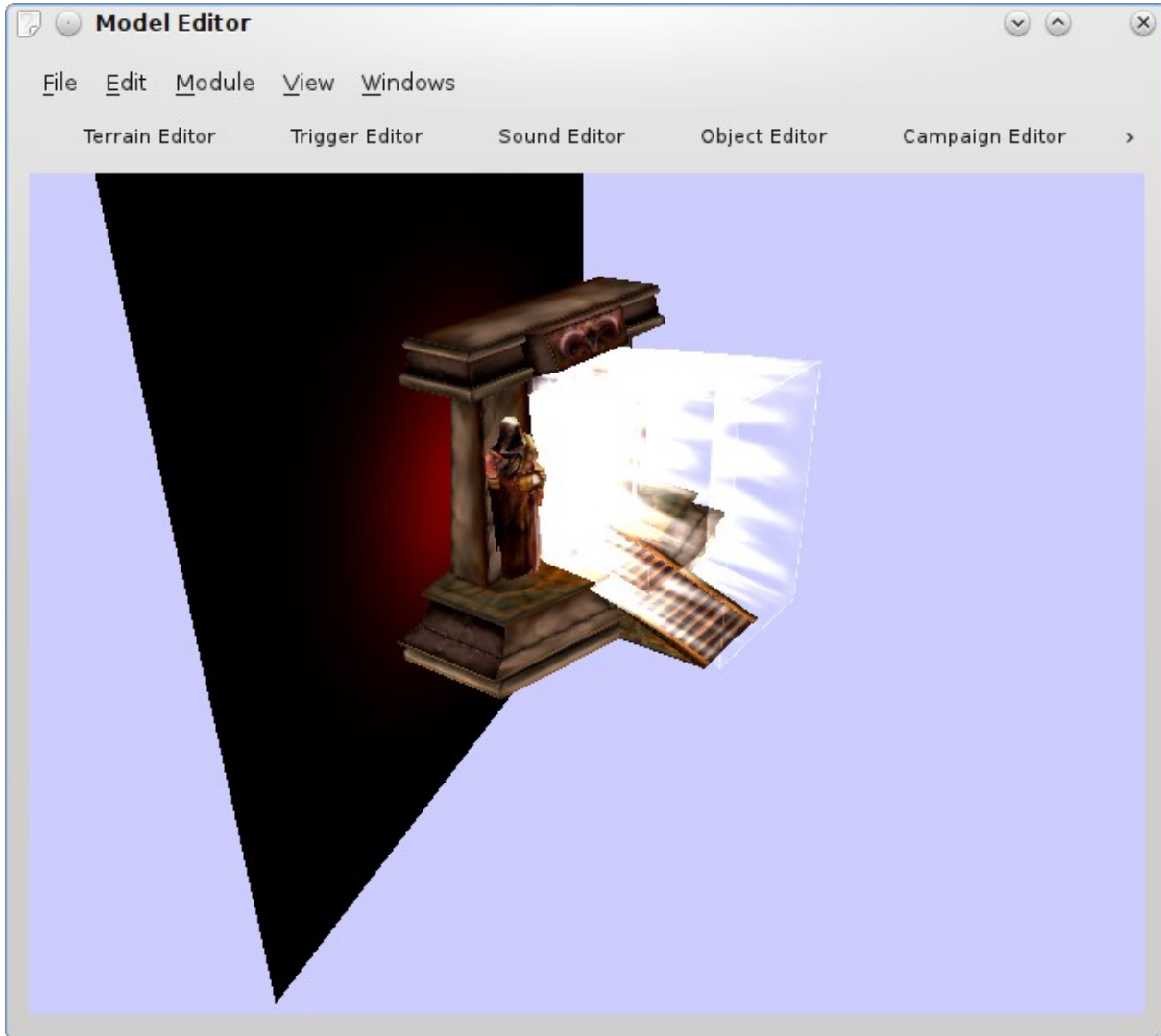


Nun ein Bild des dargestellten Modells ohne Animationen in Magos „War3 Model Editor“:



Im Modelleditor werden im Gegensatz zum Originalspiel die Lichtflächen in der Mitte des Portals angezeigt, welche für die Animationen bestimmt sind angezeigt. Dies ist kein Fehler. Die Flächendarstellung wurde bei der oberen Version der 3D-Grafik lediglich entfernt, da es im Spiel nicht ohne weiteres Möglich ist, Modelle ohne Animationen darzustellen.

Zu guter Letzt ein Bild des nicht animierten Modells im Modelleditor der Projektarbeit:



Die schwarze Fläche im Hintergrund dient dem Modellportrait. Sie wird bei der Portraitanimationen eingeblendet und bleibt ansonsten unsichtbar. Bei Spielfiguren, die einem Spieler gehören, wird diese Hintergrundfläche meist dazu genutzt, eine bestimmte Spielerfarbe hervorzuheben.

Die MDL-Datei, die das Modell beschreibt, beginnt für MDL-Dateien typischerweise mit einem Versionsblock:

```
1 // Converted Sat Mar 26 15:20:39 2011
2 // MdlxConv Version Jun 29 2010.
3 Version {
4     »     FormatVersion 800,
5 }
```

Die derzeitige Version des MDLX-Formats ist 800.

Zwei Slash-Zeichen leiten in einer MDL-Datei einen Kommentar ein, der als Hinweis für andere Grafiker dienen kann oder - wie hier – angibt, wann die Datei mit welchem Programm erstellt wurde.

Als Nächstes folgt der Modellblock:

```
6 ▼ Model "DarkPortal" {
7   »   BlendTime 150,
8   »   MinimumExtent { -282.828003, -351.631989, -0.585090 },
9   »   MaximumExtent { 420.002991, 339.186005, 467.748993 },
10  }
```

Er enthält sowohl den Modellnamen als auch die maximale und minimale Ausdehnung des Modells, was für das Rendern von Bedeutung ist.

Nun überspringen wir die Blöcke „Sequences“ und „GlobalSequences“, da diese für die Bewegungen des Modells gedacht sind und kommen zum Block „Textures“:

```
40 ▼ Textures 5 {
41   ▼ »   Bitmap {
42     »   »   Image "Textures\BloodWhiteSmall.blp",
43     »   »   }
44   ▼ »   Bitmap {
45     »   »   Image "Textures\Dust5A.blp",
46     »   »   }
47   ▼ »   Bitmap {
48     »   »   Image "Textures\DarkPortal03.blp",
49     »   »   }
50   ▼ »   Bitmap {
51     »   »   Image "Textures\PortalEnergy.blp",
52     »   »   }
53   ▼ »   Bitmap {
54     »   »   Image "",
55     »   »   ReplaceableId 2,
56     »   »   }
57 }
```

Er enthält die Liste der Dateipfade der verwendeten Texturdateien, welche auf das Netz des Modells gelegt werden.

Als Nächstes folgt der Block „Materials“:

```
58 ▼ Materials 4 {
59 ▼ » Material {
60 ▼ »     » Layer {
61 »     »     » FilterMode None,
62 »     »     » static TextureID 2,
63 »     »     » }
64 »     » }
65 ▼ » Material {
66 ▼ »     » Layer {
67 »     »     » FilterMode None,
68 »     »     » Unshaded,
69 »     »     » static TextureID 2,
70 »     »     » }
71 »     » }
72 ▼ » Material {
73 »     » ConstantColor,
74 ▼ »     » Layer {
75 »     »     » FilterMode Additive,
76 »     »     » TwoSided,
77 »     »     » static TextureID 3,
78 »     »     » static Alpha 0.500000,
79 »     »     » }
80 »     » }
81 ▼ » Material {
82 ▼ »     » Layer {
83 »     »     » FilterMode None,
84 »     »     » Unshaded,
85 »     »     » static TextureID 4,
86 »     »     » }
87 »     » }
88 » }
```

Er enthält wiederum die Liste der verwendeten Materialien, die zum Beispiel die Lichteinwirkung und den Verlauf bzw. die Überschneidung von Texturen bestimmen.

Nun folgen die Definitionen der wichtigsten Elemente einer MDLX-Grafik, nämlich der „Geosets“:

```
89  Geoset {
90  »      Vertices 56 {
148  »      Normals 56 {
206  »      TVertices 56 {
264  »      VertexGroup {
322  »      Faces 1 96 {
327  »      Groups 5 7 {
334  »      MinimumExtent { -98.262703, -351.631989, 39.058800 },
335  »      MaximumExtent { 420.002991, 90.938103, 424.589996 },
336  »      BoundsRadius 323.425995,
337  »      Anim {
341  »      Anim {
345  »      Anim {
350  »      Anim {
354  »      MaterialID 2,
355  »      SelectionGroup 0,
356  }
```

Vertices, Normale, Textur-Vertices und Flächen sind dabei in eigenen Blöcken pro „Geoset“ aufgelistet. Wie man sieht verwendet jedes „Geoset“ genau ein Material dessen Id als „MaterialID“ werden muss.

Hier noch die Form der Auflistung von Vertices, Normalen, Textur-Vertices und Flächen:

```
4050  »      Vertices 4 {
4051  »      »      { -465.165985, -478.022003, 745.473999 },
4052  »      »      { -465.165985, -478.022003, -432.089996 },
4053  »      »      { -6.953510, 747.523987, 745.473999 },
4054  »      »      { -6.953500, 747.523987, -432.089996 },
4055  »      »      }
4056  »      Normals 4 {
4057  »      »      { 0.936672, -0.350207, 0.000000 },
4058  »      »      { 0.936672, -0.350207, 0.000000 },
4059  »      »      { 0.936672, -0.350207, 0.000000 },
4060  »      »      { 0.936672, -0.350207, 0.000000 },
4061  »      »      }
4062  »      TVertices 4 {
4063  »      »      { 0.000000, 0.000000 },
4064  »      »      { 0.000000, 1.000000 },
4065  »      »      { 1.000000, 0.000000 },
4066  »      »      { 1.000000, 1.000000 },
4067  »      »      }
4068  »      VertexGroup {
4069  »      »      0,
4070  »      »      0,
4071  »      »      0,
4072  »      »      0,
4073  »      »      }
4074  »      Faces 1 6 {
4075  »      »      Triangles {
4076  »      »      »      { 0, 1, 2, 3, 2, 1 },
4077  »      »      »      }
4078  »      »      }
```

Die Vertex-Gruppen haben ebenfalls nur für die Bewegungen des Modells eine Bedeutung, sowie auch die meisten anderen Blöcke, die den „Geoset“-Blöcken folgen.

Zu guter Letzt wird noch eine Kamera definiert:

```
4341 Camera "Camera01" {
4342     » Position { 861.617004, -90.756500, 121.652000 },
4343     » FieldOfView 0.785398,
4344     » FarClip 2000.000000,
4345     » NearClip 8.000000,
4346     » Target {
4347     »     » Position { 128.565994, 4.545440, 219.223007 },
4348     »     » }
4349 }
```

1.9 Ausblick

2011 könnte ein glückliches Jahr für alle C++-Freunde werden. Angeblich soll dieses Jahr der neue Standard erscheinen, der selbstverständlich bereits größtenteils von der „GCC“ (GNU Compiler Collection)²¹ unterstützt wird. Er würde der Projektarbeit völlig neue Möglichkeiten im Design erlauben und alte Fehlkonzepte aus C++ aufräumen.

Durch die Flexibilität des „OGRE“-Frameworks ist es rein theoretisch möglich, Bearbeitungswerkzeuge für 3D-Grafiken zu entwickeln. Daher könnte man das Format über eine einfache grafische Oberfläche zumindest grundlegend bearbeitbar machen (zum Beispiel indem man die Verschiebung von Vertices mit der Maus zuließe).

Außerdem ermöglicht das „OGRE“-Framework die Serialisierung in andere Formate, was eine einfache Konvertierung per grafische Oberfläche zuließe.

Insgesamt macht es wohl am meisten Sinn, die restlichen Eigenschaften des „MDLX“-Formats darstellbar zu machen (vor allem Bewegungen und spielinterne Effekte), um ein exaktes Anzeigeprogramm für das Format zu erhalten.

Zudem könnte man das „MDL“-Format der Vollständigkeit halber über Parser-Bibliotheken bzw. -Funktionen einlese- und schreibbar machen, da bereits die notwendigen Klassen im „mdlx“-Modul dafür existieren.

Bei der Implementation der Lese- und Schreibfunktionen des „BLP“-Formats kam zum ersten Mal die Technik des „Multithreading“ zum Einsatz (bei der Kompression bzw. Dekompression der „JFIF“-Bilder). Die Verwendung von mehreren Threads macht an diversen Stellen Sinn, um die Effizienz auf Mehrkernprozessorsystemen zu steigern. Das „OGRE“-Framework bringt von Haus aus einige Funktionen mit, die das automatisch gesteuerte Rendern mit mehreren Threads erlauben. Dies könnte in Zukunft den manuellen Aufruf der Elementfunktion „render“ der Klasse „ModelView“, vor allem bei Bewegungen der 3D-Grafiken, ersetzen.

²¹ <http://gcc.gnu.org/>

1.10 Quellen- und Literaturangaben

- QtOgreFramework: <https://ogreaddons.svn.sourceforge.net/svnroot/ogreaddons/trunk/QtOgreFramework/>
- Qt: <http://qt.nokia.com/>
- CMake: <http://cmake.org/>
- OGRE: <http://www.ogre3d.org/>
- Boost C++ Libraries: <http://www.boost.org/>
- KDE: <http://kde.org/>
- MDLX-Spezifikation: <http://www.wc3c.net/tools/specs/NubMdxFormat.txt>
- War3 Model Editor: <http://home.magosx.com/index.php?topic=6.0>
- OGRE-Tutorial für manuelle Objekte: <http://www.ogre3d.org/tikiwiki/ManualObject>
- OGRE-Tutorial für Auswahlabfragen: <http://www.ogre3d.org/tikiwiki/tiki-index.php?page=Raycasting+to+the+polygon+level>