Matthias Belz
Bernd Gaßmann

Studienarbeit

Entwicklung einer dreidimensionalen Benutzerschnittstelle für digitale Bibliotheken unter Quake 2

Betreuerin: Dipl.-Inform. Bethina Schmitt

Institut für Programmstrukturen und Datenorganisation
Lehrstuhl Prof. P. C. Lockemann
Fakultät für Informatik
Universität Karlsruhe

Februar 1999

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation und Zielsetzung

Zahlreiche Projekte beschäftigen sich mit der grafischen Visualisierung von Ergebnismengen, sei es bei der klassischen Literaturrecherche, bei der Suche in Nachrichtenarchiven, Bookmarks oder im WWW. Ziel dieser Visualisierungen ist es, dem Benutzer die Beschaffenheit des Suchergebnisses besser zu veranschaulichen: Teilweise liegt der Fokus auf der *Zusammensetzung der Gesamtergebnismenge*, so daß der Benutzer einen schnellen Überblick gewinnen und sofort beurteilen kann, ob das Ergebnis dem gewünschten entspricht. Falls nicht, soll der Benutzer dadurch direkte Anhaltspunkte erhalten, in welcher Richtung die Anfrage abzuwandeln ist. Teilweise werden auch gewisse *Eigenschaften der Ergebnistreffer* in grafischer Form dargestellt, da sich diese von einem Benutzer so schneller erfassen lassen als in textueller Form.

In Anbetracht der ständig zunehmenden Informationsfülle ist die Benutzerunterstützung durch geeignete Visualisierungstechniken durchaus erstrebenswert, ja sogar notwendig. Aber obwohl mittlerweile die notwendigen technischen Voraussetzungen und entsprechende Programmpakete wie VRML 2.0 und Java 3D existieren, ist noch kein grafisch unterstütztes Recherchesystem populär geworden, noch immer verwenden die Benutzer die gewohnten textbasierten Schnittstellen.

Über die Gründe dafür kann nur spekuliert werden:

- Die Benutzer sind ungeduldig, grafische Ausgaben sind immer langsamer als reine Textanzeigen.
- Die Benutzer sind an textuelle Ergebnislisten und deren schnelles Überfliegen gewöhnt.
- Die Benutzer haben zu wenig Zeit, sich auf Neues einzulassen, und bleiben lieber bei Bekanntem.
- Die technischen Startschwierigkeiten wirken abschreckend, z.B. wenn erst ein VRML-Browser oder Java 3D installiert werden muß.
- Die grafischen Systeme bieten keine prägnante Visualisierung, die dem Benutzer wirklich weiterhilft, bzw. der Benutzer erkennt den Mehrwert nicht.

- Die Systeme sind nur als Forschungs-Prototyp realisiert und noch nicht öffentlich im Betrieb.
- Die Darstellung wirkt künstlich bzw. unrealistisch, der Spaßfaktor bei der Benutzung ist gering.
- Die Benutzerinteraktionen sind ungewohnt, z.B. bei VRML.

In dieser Studienarbeit soll kein passendes Erklärungsmodell dafür gefunden werden, sondern eine andere neuartige Herangehensweise erprobt werden.

Es gibt zahlreiche 3D-Anwendungen, die sich mittlerweile durchgesetzt haben und sehr populär geworden sind, z.B. die Klasse der 3D-Spiele. Diese Programme sind mittlerweile sehr ausgereift, d.h. sie verfügen über leistungsstarke 3D-Engines zur Darstellung von realistischen Szenen und verfügen über eine auch für ungeübte Benutzer leicht zu bedienende Interaktionsschnittstelle. Die weite Verbreitung zeigt die große Benutzerakzeptanz dieser Produkte.

Einige Spiele erlauben es, Anwendungen mit selbst definierter Funktionalität zu erstellen, die dann ebenso von derselben 3D-Engine dargestellt werden und über dieselbe Benutzerschnittstelle bedient werden können.

Es soll deshalb im Rahmen dieser Studienarbeit mit einem dieser Spiele, Quake 2, eine 3D-Benutzerschnittstelle für das UniCats-Projekt entwickelt werden. Dies umfaßt die Entwicklung einer geeigneten räumlichen Umgebung (z.B. durch Nachbilden des Gebäudes der Universitätsbibliothek), die Konzeption und Realisierung einer Anfrageschnittstelle und einer 3D-Ergebnispräsentation sowie die Anbindung der Benutzerschnittstelle an das bestehende System (z.B. über TCP/IP). Neben den konzeptionellen Überlegungen und der tatsächlichen Realisierung soll auch das im Laufe der Arbeit erworbene Detailwissen zusammengestellt werden, um für folgende Arbeiten eine schnellere Einarbeitung zu ermöglichen. Darüberhinaus soll aufgrund der gesammelten Erfahrungen die Eignung von Quake 2 als 3D-Modellierungswerkzeug beurteilt werden, z.B. im Hinblick auf notwendige technische Voraussetzungen, Entwicklungsaufwand, Stärken und Schwächen, geeignete und ungeeignete Anwendungsgebiete etc.

1.2 Einbettung in das UniCats-Projekt

UniCats[1] [4] ist ein von der DFG gefördertes Digital-Libraries-Projekt des Instituts für Programmstrukturen und Datenorganisation in Zusammenarbeit mit der Universitätsbibliothek Karlsruhe. Ziel ist die Verbesserung der Literaturversorgung von Universitätsangehörigen durch Integration der diversen existierenden Recherche- und Beschaffungsdienste. Die dem UniCats-Projekt zugrundeliegende Architektur umfaßt drei Komponenten (Benutzeragent, Trader und Wrapper), die durch Zusammenarbeit die gewünschte Funktionalität erbringen.

[1] a UNiversal Integration of Catalogues based on an Agent-supported Trading and wrapping System

Der Benutzeragent bildet die Schnittstelle zwischen Benutzer und System. Entsprechend sind seine Aufgaben die Bereitstellung einer einheitlichen, verständlichen und benutzerfreundlichen Schnittstelle zur Literaturrecherche und -beschaffung sowie das Schützen des Benutzers vor Informationsüberflutung.

Um eine Anfrage bearbeiten zu können, benötigt ein Benutzeragent zunächst Informationen über verschiedene Dienstanbieter (Provider). Diese erhält er von Tradern. Sie nennen dem Benutzeragenten die Provider, die am besten zu seiner Anfrage passen und darüber hinaus weitere relevante Informationen, z.B. über Ausleihkonditionen und Kosten.

Mit diesem Wissen kann sich der Benutzeragent an die Provider wenden. Da nun aber in der Praxis jeder Dienst in einer anderen Syntax angesprochen werden muß, gibt es Wrapper, die eine Anfrage semantisch und syntaktisch so umformulieren, daß sie vom entsprechenden Provider bearbeitet werden kann. Die Provider können also weiterhin autonom bleiben.

Dank der Wrapper kann der Benutzeragent nun die Provider in einheitlicher Form ansprechen. Die Ergebnisse seiner Anfragen bearbeitet er und präsentiert sie dem Benutzer. Hier ist unser Projekt anzusiedeln, denn wir möchten für den Benutzeragenten mit Quake 2 eine 3D-Benutzerschnittstelle entwickeln, um die Ergebnisse zu visualisieren. Außerdem werden Anfragen vom Benutzer entgegengenommen und abgesetzt. Die Zusammenarbeit mit Tradern und Wrappern erfolgt jedoch nach wie vor durch den eigentlichen Benutzeragenten.

1.3 Aufbau der Studienarbeit

Zunächst werden in Kapitel 2 die Grundlagen des Benutzerschnittstellendesigns erläutert. Während im ersten Teil allgemeine Anforderungen an Benutzerschnittstellen genannt werden, werden im zweiten Teil des Kapitels verschiedene Benutzerschnittstellen aus dem Digital-Libraries-Umfeld vorgestellt und kurz bewertet. Der dritte Teil beschäftigt sich mit den 3D-Modellierungsumgebungen VRML und Java 3D. In Kapitel 3 wird dann das Programm vorgestellt, das wir zur 3D-Modellierung verwendet haben, Quake 2. Nach einer allgemeinen Einführung folgen Informationen zum internen Aufbau und der Struktur des Spiels. Anschließend werden Beschränkungen der Quake 2-Engine beschrieben. Kapitel 4 beschreibt, wie mit Quake 2 eigene Welten erstellt werden können. Neben den einzelnen Schritten werden hier auch die dafür nötigen Programme vorgestellt. Kapitel 5 enthält schließlich eine Zusammenfassung dieser Studienarbeit, eine Bewertung sowie einen Ausblick auf mögliche Erweiterungen bzw. Verbesserungen.

Der Anhang besteht aus vier Teilen: Anhang A enthält ein Glossar Quake 2-spezifischer Fachausdrücke. In Anhang B wird die Installation und die Bedienung unserer Erweiterung beschrieben, weiterhin werden die wichtigsten Konsolenvariablen und -kommandos vorgestellt. Es folgt eine ausführliche Dokumentation der Quake 2-Spielelogik und der von uns erstellten Erweiterungen. Anhang C besteht aus einer Übersicht über die von uns zur Modellierung verwendeten Programme. In Anhang D schließlich findet man eine Auflistung von Internetadressen zu Quake 2 und zu diversen Digital-Libraries-Projekten.

2 Benutzerschnittstellendesign und -entwicklungswerkzeuge

2.1 Allgemeine Designgrundsätze

Eine Benutzerschnittstelle ist nach [13] „der Teil eines Anwendungsprogramms, der die Darstellung und Durchführung von Ein- und Ausgaben übernimmt und den Dialog mit dem Benutzer führt“. Jeglicher Kontakt zwischen Benutzer und Programm erfolgt also über die Benutzerschnittstelle. Daraus folgt die entscheidende Bedeutung von Benutzerschnittstellen, mit ihrer Qualität steht und fällt der Erfolg des Gesamtsystems. Auch ein noch so gutes Programm wird nicht populär werden, wenn es schlecht zu bedienen ist. Im folgenden werden daher einige Anforderungen beschrieben, denen Benutzerschnittstellen genügen sollten.

Gemäß [14] ist das Gesamtoptimierungsziel beim Mensch-Maschine-Dialog „mit einem Minimum an geistig-körperlichem Aufwand und Zeit (und Kosten) ein Maximum an Aufgabenbewältigung/Arbeitsleistung zu erzielen“. Um dies zu erreichen, werden die beiden folgenden Maßnahmen vorgeschlagen:

Der körperliche Interaktionsaufwand ist zu minimieren, so daß der Benutzer u.a. möglichst wenig tippen, lesen, die Augen anstrengen und die Maus bewegen muß. Außerdem sollte eine Anpassung an das intellektuelle und körperliche Leistungsvermögen des Benutzers erfolgen, Überforderungen und Unterforderungen sollten vermieden werden.

Auch der zeitliche Aufwand muß minimiert werden, wenn man höchste Arbeitseffizienz erreichen will. Dazu gehört zum einen die Möglichkeit zur schnellen Bedienung durch kurze, mächtige Kommandos, zum anderen ist allerdings auch eine schnelle Programmausführung wichtig.

Ein weiterer wichtiger Aspekt betrifft die Prägnanz: Bei der Informationsdarstellung müssen die wesentlichen Informationen im Mittelpunkt stehen, der Benutzer darf nicht durch Unwesentliches abgelenkt werden. In der Anthropotechnik spricht man in diesem Zusammenhang von Nutz- und Störinformationen. So wird z.B. in [17] als „Satz zur Gestaltung der Informationsquelle“ festgehalten: „Nutzinformation soll deutlich auffälliger sein als Störinformation.“

Neben hoher Effizienz und Prägnanz ist für eine Schnittstelle auch die Akzeptanz durch den Benutzer von großer Bedeutung. Dazu gehört z.B., daß sich ein Benutzer leicht zurechtfindet. Eine Möglichkeit, dies zu erreichen, sind Metaphern. Damit werden neue Konzepte mit Begriffen bereits bekannter Konzepte erklärt [6]. Ein bekanntes Beispiel ist die Desktop-Metapher, die in vielen Betriebssystemen verwendet wird. Ein Papierkorb symbolisiert hier z.B. die Möglichkeit zum Löschen von Dateien, analog zum Papierkorb der wirklichen Welt. Die Autoren von [6] schreiben: „Nur selten wird der Wert einer guten Metapher für die Steigerung der unmittelbaren Vertrautheit zwischen Anwender und Computer in Frage gestellt“. Gleichzeitig weisen sie aber auch auf die Gefahren durch ungeeignete Metaphern hin. Ein Beispiel dafür findet man an der Benutzerschnittstelle des Apple Macintosh, hier muß man das Icon einer Diskette auf den Papierkorb ziehen, um sie auszuwerfen. Hier wird die Desktop-Metapher überstrapaziert, sie ist nicht mächtig genug, um diese Funktionalität auszudrücken.

2.2 Benutzerschnittstellen in digitalen Bibliotheken

Im Digital Libraries-Umfeld finden sich die unterschiedlichsten Ansätze zur Informationsvisualisierung. Die älteste und immer noch am weitesten verbreitete Form ist die Textdarstellung, wie man sie z.B. beim Karlsruher Virtuellen Katalog (KVK) findet (Abbildung 2.1). Ein Sonderfall der Textdarstellung ist die tabellarische Darstellung. Der Vorteil liegt hier

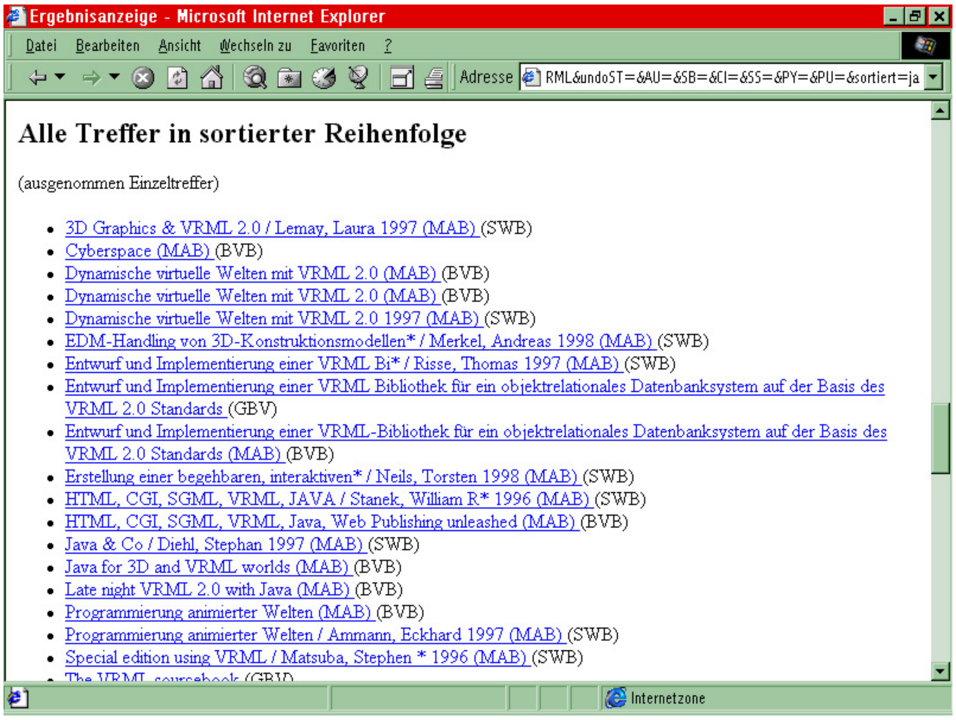


Abbildung 2.1: Ergebnisanzeige beim KVK

in der einfachen Vergleichbarkeit der einzelnen Einträge. Nachteile sind laut [2] der große horizontale Platzbedarf (der meist ein Scrollen unvermeidlich macht) und die Tatsache, daß bei Dokumenten aus heterogenen Quellen oft Felder leer bleiben und unnötig den Lesefluß hemmen.

Strunk, William Jr. and E. B. White. *The Elements of Style.* Third edition. New York: Macmillan Publishing Co., Inc., 1979.

Morris, William and Mary Morris. *Morris Dictionary of Word and Phrase Origins*. New York: Harper and Row, Publishers, 1977.

Abbildung 2.2: Beispiel für Titelaufnahmen

Dies trifft auf Titelaufnahmen (englisch „Citations“), wie z.B. in Abbildung 2.2, nicht zu. Diese Darstellungsform findet man meist in gedruckten Werken, aber auch fast alle Suchmaschinen im Internet liefern ihre Treffer in Form von textuell fortlaufenden Beschreibungen. Sie brauchen weniger Platz als Tabellen, weil nur die Informationen dargestellt werden, die tatsächlich vorhanden sind. Allerdings sind Titelaufnahmen unübersichtlicher, und es fällt hier schwerer, Dokumente zu vergleichen.

An der Stanford University hat deshalb die Vorteile beider Darstellungsformen vereint: *SenseMaker* [1] ist Teil des *Stanford Digital Library Project* und dient als Interface zur Informations- bzw. Literaturrecherche. Ursprünglich wurden auch hier die Informationen in Tabellenform angezeigt, wobei der Benutzer die Spalten dynamisch konfigurieren konnte. Inzwischen hat man aber das Konzept der Hi-Cites entwickelt [2]. Dabei handelt es sich um dynamisch erstellte Titelaufnahmen mit der Möglichkeit zur aktiven Hervorhebung von Attributen. Für die Titelaufnahmen kann man verschiedene Stile definieren, die u.a. die Reihenfolge und Schriftart der Attribute festlegen. Die Hervorhebung einzelner Attribute geschieht dadurch, daß der Benutzer mit der Maus kurz über dem entsprechenden Attribut (z.B. „Autor“) verweilt. Es wird dann in allen Titelaufnahmen der Autor z.B. rot hervorgehoben (siehe Abbildung 2.3).

Betrachtet man die textuelle Visualisierung unter dem Gesichtspunkt der im vorigen Abschnitt genannten Kriterien, so ergibt sich ein zweischneidiges Bild. Einige Kriterien werden ohne weiteres erfüllt, wie z.B. die Prägnanz. Der größte Teil der Bildschirmanzeige besteht nämlich aus Nutzinformationen, Störinformationen spielen kaum eine Rolle. Auch eine Überforderung des Benutzers liegt nicht vor: Der Mensch ist von Kind auf trainiert, mit textueller Darstellung umzugehen und kann Auflistungen etc. relativ schnell überblicken. Allerdings ist hier die Organisation der Informationsdarstellung von entscheidender Bedeutung, eine tabellarische Darstellung ist zum Beispiel übersichtlicher als eine fortlaufende Darstellung. Andererseits muß man bei der tabellarischen Darstellung oft zur Seite scrollen, um alle Informationen zu überschauen. Das wirkt sich aber wiederum negativ auf den körperlichen und den zeitlichen Aufwand aus. SenseMaker hat diese Problematik recht gut

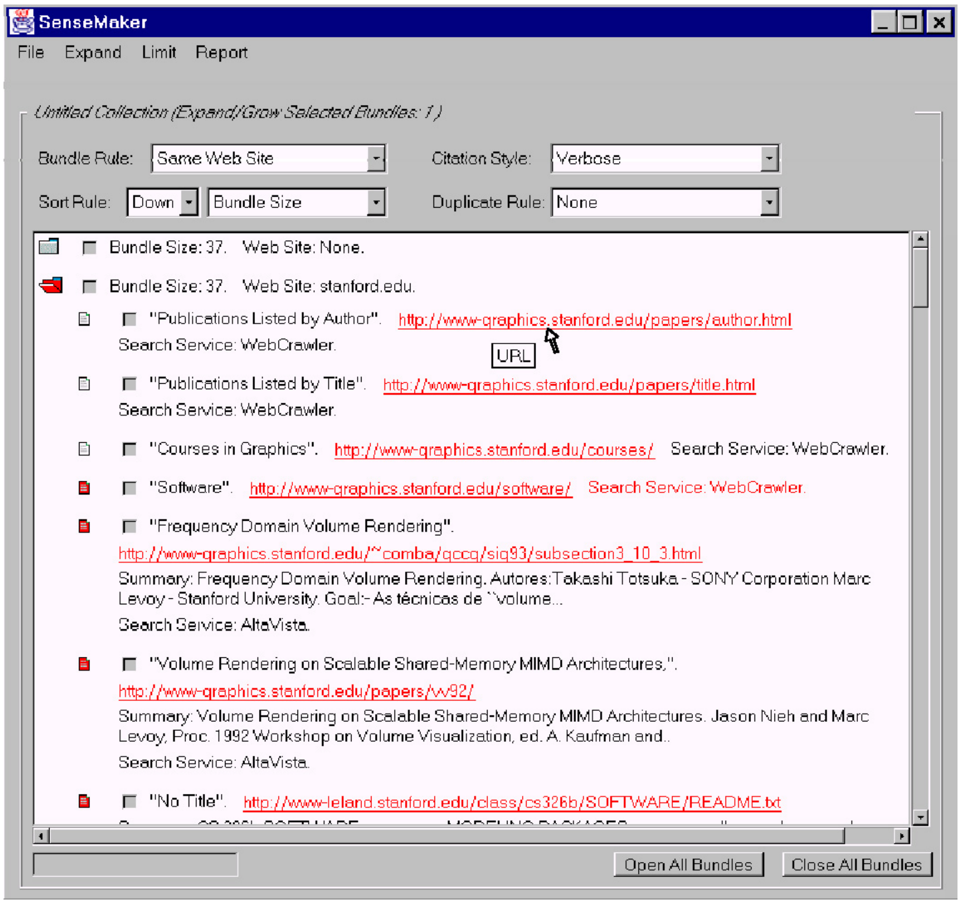


Abbildung 2.3: SenseMaker mit HiCites-Interface

gelöst, indem es mit einem Kompromiß aus beiden Darstellungsformen arbeitet.

Natürlich lassen sich Informationen aber auch ganz anders darstellen, und zwar grafisch. Man kann hier zwei- und dreidimensionale Darstellungsformen unterscheiden. Das *ETD Digital Library*-Projekt [12] unter der Leitung von Neill A. Kipp (Virginia Polytechnic Institute and State University) unterstützt sogar beide Formen. Nach Kipp sind Menschen daran gewöhnt, räumliche Informationen (wie z.B. bei einem Bücherregal) zu verarbeiten, und es ist für sie natürlicher, als eine Liste von einheitlichen Einträgen auf einem Bildschirm abzuarbeiten. Deshalb arbeitet das Projekt mit der Metapher einer realen Bibliothek. Entsprechend gibt es in dieser Bibliothek Flure, Räume, Aufzüge, eine Lobby usw. Ein HTML-Interface sorgt dafür, daß die notwendigen Informationen zweidimensional dargestellt werden können. Die einzelnen Räumlichkeiten werden jeweils durch eine HTML-Seite repräsentiert, ein Java-Applet sorgt in Verbindung mit GIF-Grafiken für die Darstellung. Abbildung 2.4

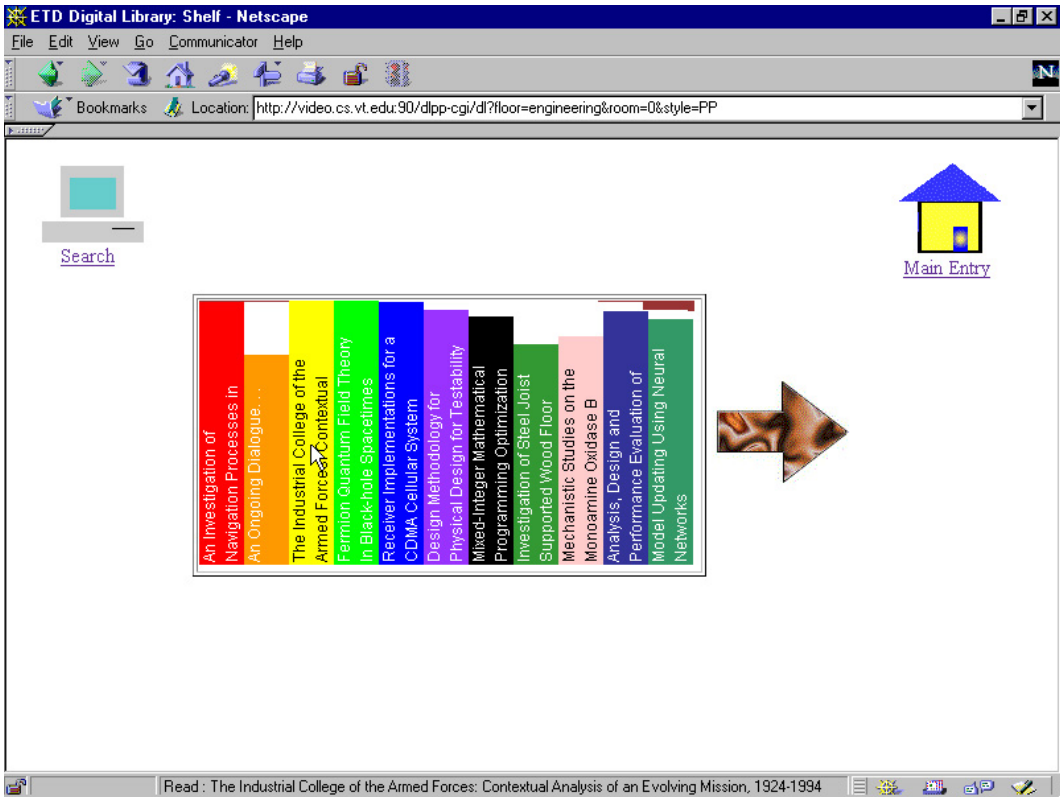


Abbildung 2.4: ETD Digital Library-Projekt mit 2D-Darstellung

zeigt das Aussehen dieses Interfaces.

Der Schwerpunkt des ETD-Projekts liegt allerdings bei der dreidimensionalen Darstellung. Verwendet wird VRML 2.0. Abbildungen 2.5 und 2.6 vermitteln einen Eindruck dieses Ansatzes. Der Benutzer bewegt sich mit den Kontrollen des VRML-Betrachters durch den dreidimensionalen Raum und nutzt Hyperlinks, die durch einige Objekte repräsentiert werden. Ein Klick auf eine Tür führt zum Beispiel in den dahinterliegenden Raum.

Leider ist dieses System nicht öffentlich verfügbar, so daß man wenig über den körperlichen und zeitlichen Aufwand bei der Bedienung sagen kann. Erfahrungsgemäß ist jedoch die Navigation unter VRML grundsätzlich eher mühsam und nicht sonderlich schnell. Die Prägnanz ist bei der 3D-Darstellung geringer als beispielsweise bei der Textdarstellung, was allerdings vor allem an der Verwendung einer Metapher liegt. Bei der 2D-Darstellung des ETD-Projektes ist die Prägnanz höher, wenn auch die Bücher etwas größer dargestellt sein könnten. Trotzdem ist die hier verwendete Methode zur Visualisierung völlig unbrauchbar, denn die Beschriftung der Bücher erfolgt vertikal, wie in Abbildung 2.4 zu sehen ist. Um sie lesen zu können, muß der Benutzer also den Kopf drehen, d.h. der körperliche Interak-

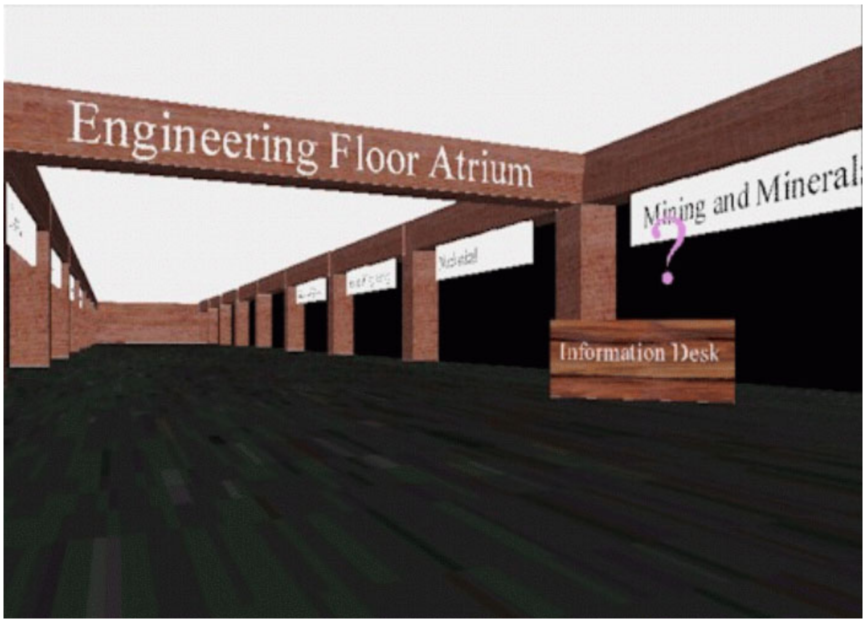


Abbildung 2.5: ETD Digital Library-Projekt mit 3D-Darstellung: Blick auf einen Flur

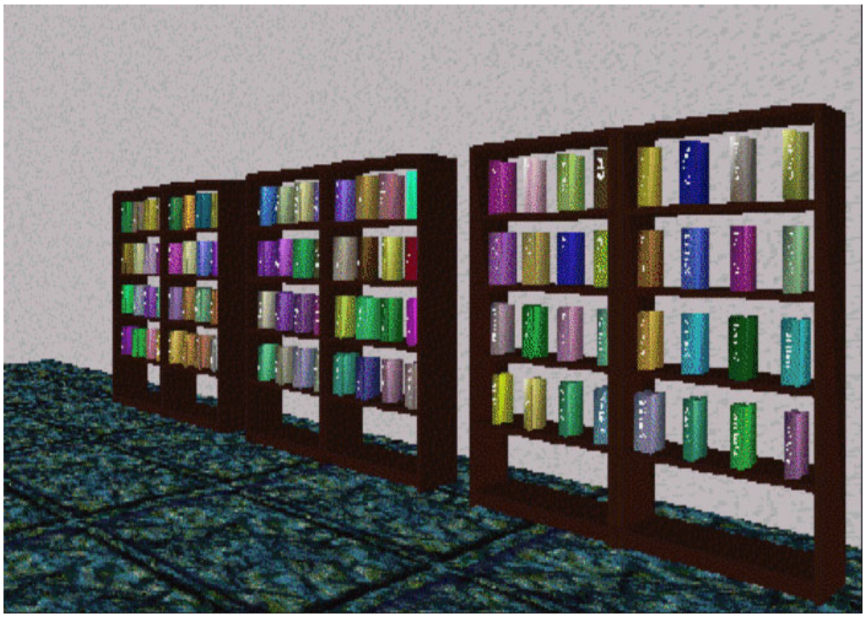


Abbildung 2.6: ETD Digital Library-Projekt mit 3D-Darstellung: Bücher und Regale

tionsaufwand ist hier sehr hoch. Der Titel des Buches, über dem der Mauszeiger verweilt, wird zwar auch noch in der Fußzeile des Browsers angezeigt, trotzdem wird hier der Sinn der grafischen Visualisierung in Frage gestellt.

Ein anderes 3D-basiertes Projekt ist der *VRML-Document-Finder* von der Fachhochschule Potsdam [5]. Hintergrund ist auch hier die Tatsache, daß lange textuelle Listen nur

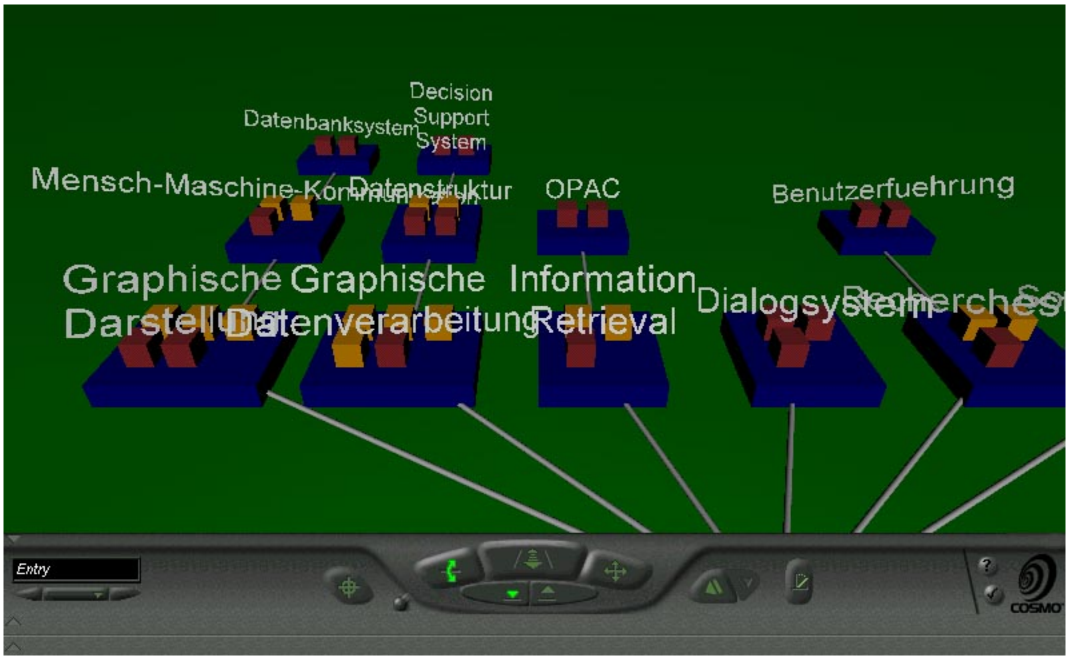


Abbildung 2.7: VRML-Document-Finder der FH Potsdam

schwer zu überblicken sind. Der VRML-Document-Finder gruppiert deshalb Suchergebnisse, die er aus HTML-Dateien einliest, nach ihrer Ähnlichkeit und bildet eine Hierarchie. Ein VRML-Generator erzeugt daraus eine dreidimensionale Welt, die vom Benutzer mit den VRML-eigenen Navigationsmöglichkeiten erforscht werden kann. Wie auch beim ETD Digital Library-Projekt ist die Darstellung konkreter Natur, d.h. sie orientiert sich an Metaphern der wirklichen Welt. In diesem Fall handelt es sich um die sogenannte CityScape-Metapher: Die einzelnen Dokumente werden durch Blöcke dargestellt, die an Gebäude erinnern. Ähnliche Dokumente befinden sich auch räumlich dicht beeinander, dadurch entstehen in der dreidimensionalen Welt verschiedene „Städte“ (siehe Abbildung 2.7).

Obwohl hier mit Metaphern gearbeitet wird, wirkt diese Art der Visualisierung für einen ungeübten Benutzer eher verwirrend, eine Überforderung ist deshalb nicht auszuschließen. Der Screenshot zeigt, das die Informationsdarstellung ziemlich unübersichtlich erfolgt, obwohl nicht sonderlich viele Informationen dargestellt werden. Ob sich dieses System schnell bedienen läßt, darf ebenfalls bezweifelt werden. Alles in allem scheint dieser Ansatz nicht sehr vielversprechend zu sein.

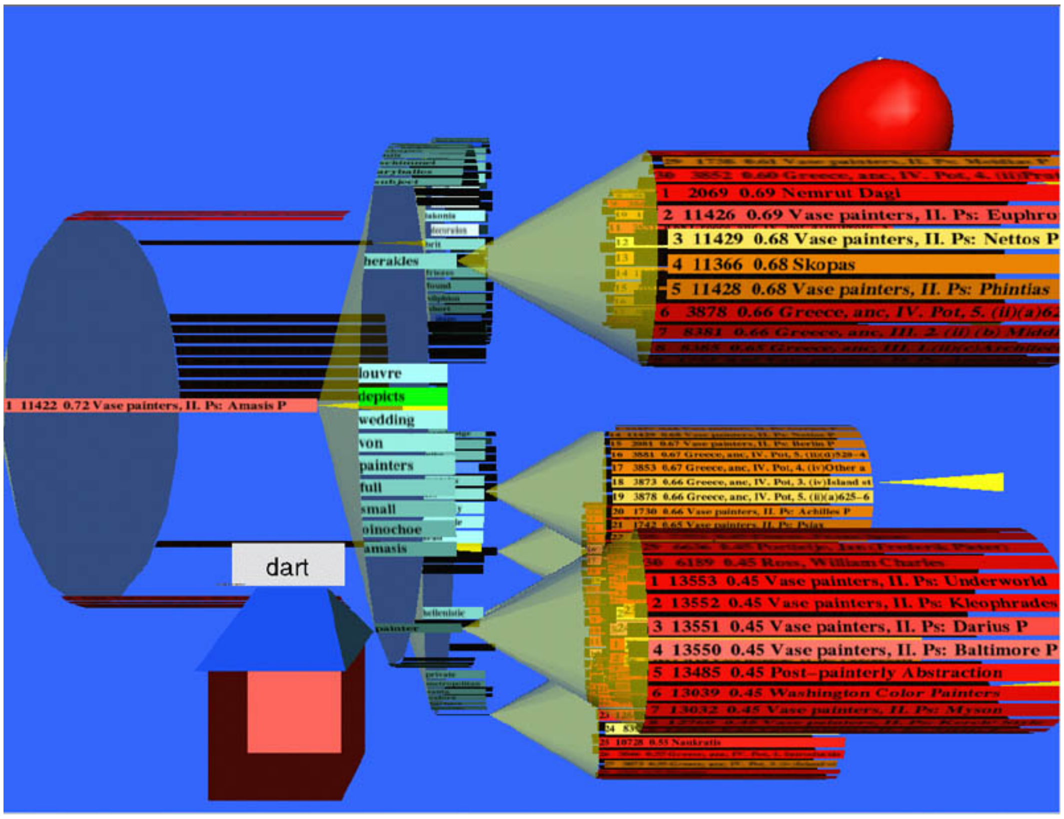


Abbildung 2.8: LyberWorld-Projekt: Kegelbäume

Im Gegensatz zur konkreten gibt es auch noch die abstrakte dreidimensionale Darstellung. Ein Beispiel dafür ist das *LyberWorld*-Projekt [8] der GMD Darmstadt. Ziel dieses Projekts ist die Visualisierung großer Informationsmengen. Die Informationen werden hier durch abstrakte Gebilde wie Kugeln oder Kegelbäume dargestellt. Kegelbäume sind eine dreidimensionale Erweiterung herkömmlicher Bäume, die Gesamtheit der Äste eines Knotens wird dabei durch einen Kegel repräsentiert (Abbildung 2.8). Auf der Kegeloberfläche befinden sich einzelne Textzeilen, die das Ergebnis einer Suchanfrage repräsentieren. Mit Hilfe der Kegelbäume kann der Benutzer durch den Informationsraum navigieren. Daneben gibt es auch noch sogenannte Relevanzkugeln (siehe Abbildung 2.9). Dokumente werden hier durch Blöcke, Begriffe durch kleine blaue Kugeln dargestellt. Die Nähe eines Dokuments zu einer Begriffskugel veranschaulicht seine Relevanz für diesen Begriff. Die Entfernung zwischen einem Dokument und der Kugeloberfläche entspricht seiner Wichtigkeit im gesamten Suchkontext.

Auch bei diesem System könnte ein untrainierter Benutzer überfordert werden. Die In-

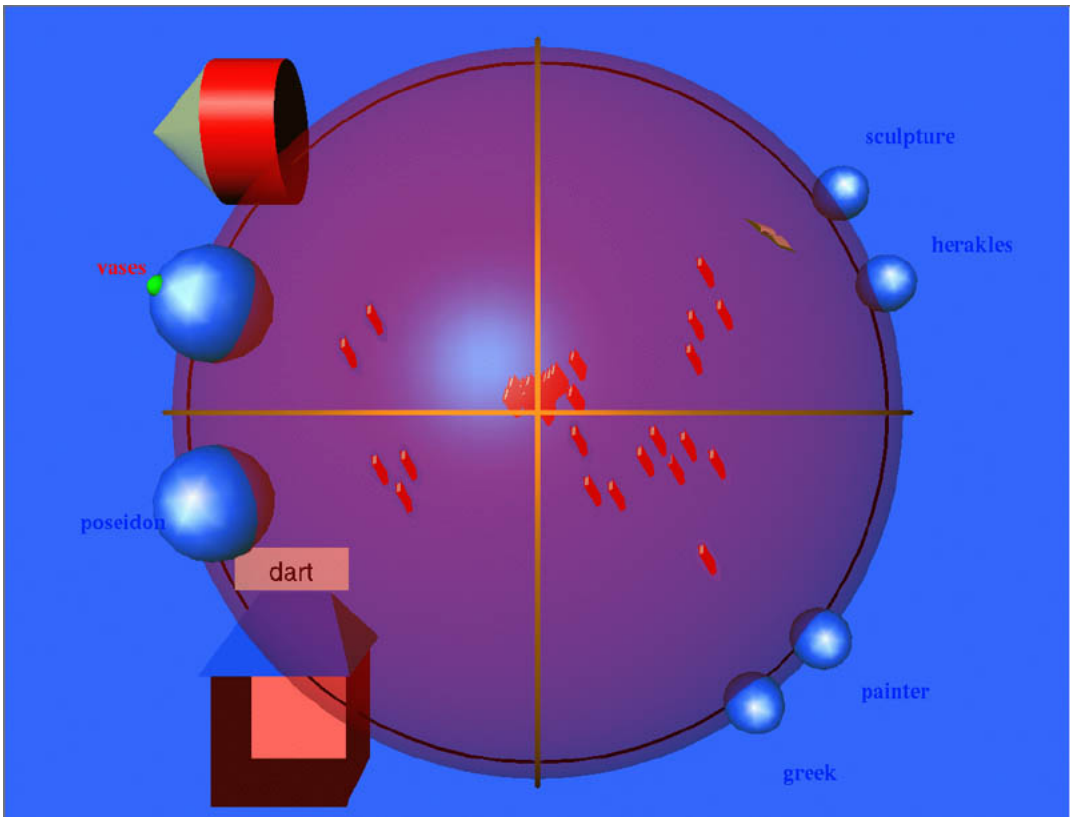


Abbildung 2.9: LyberWorld-Projekt: Relevanzkugel

formationsdarstellung wirkt allerdings übersichtlicher und durchdachter als beim VRML-Document-Finder. Über die Bedienung läßt sich keine Aussage treffen, denn das System ist nicht öffentlich verfügbar. Es drängt sich aber der Eindruck auf, daß dieses Projekt zwar wissenschaftlich gesehen sehr interessant ist, sich aber in der Praxis nie durchsetzen wird.

Wie man an diesen Beispielen sieht, erfreut sich die grafische Visualisierung im Digital Libraries-Bereich großer Beliebtheit. Insbesondere wird die dreidimensionale Darstellung immer öfter genutzt, seit die Computer die dafür nötige Rechenleistung besitzen. Die Vorteile dieser Darstellung liegen unter anderem darin, daß es dem Menschen naturgemäß leicht fällt, sich in dreidimensionalen Umgebungen zurechtzufinden. Außerdem ist es durch die dreidimensionale Visualisierung problemlos möglich, Metaphern aus der wirklichen Welt heranzuziehen, um den Benutzern das Interface näherzubringen. Tatsache ist allerdings, daß sich all diese grafischen Benutzerschnittstellen noch nicht durchsetzen konnten, möglicherweise weil sie noch nicht ausgereift sind, oder weil die Bedienung zu langsam oder zu kompliziert ist. Deshalb wird bei unserem Projekt ein anderer Ansatz verfolgt. Wir gehen von einer Benutzerschnittstelle aus, die sich bereits bewährt hat, beliebt und weit verbreitet ist.

2.3 Werkzeuge zur Entwicklung von 3D-Schnittstellen

2.3.1 VRML

Die '**V**irtual **R**eality **M**odelling **L**anguage Version 2.0' (VRML 2.0, im weiteren nur noch VRML genannt) ist ein Dateiformat, um interaktive 3D-Welten und -Objekte zu beschreiben. Es kann benutzt werden, um komplexe dreidimensionale Szenen zu repräsentieren, die sowohl statische als auch animierte Objekte enthalten können. Die Szenen werden mit Hilfe eines Interpreters auf dem Bildschirm dargestellt. Da VRML vor allem in Verbindung mit dem World-Wide-Web (WWW) nutzbar ist, bietet es die Möglichkeit der Einbindung von anderen Medien wie Musik, Filme und Bilder über „Hyperlinks". Nach der Sprachdefinition von VRML kann Benutzerinteraktion in verschiedenen Formen erfolgen. Sensoren, die in die 3D-Welten integriert werden, lösen je nach Benutzeraktion (Mausklicks, Navigation im Interpreter) Ereignisse (Events) aus, die Objekte in der 3D-Welt beeinflussen können (z.B. Timer einschalten, Bewegungsinterpolatoren starten, ...). Weiterhin ist es möglich, Skript-Sprachen (z.B. Javascript) direkt in VRML einzubinden oder über die Programmiersprache Java mit Hilfe eigens dafür definierter VRML-Klassen die 3D-Welt zu beeinflussen. Damit bestehen weitreichende Einflußmöglichkeiten in die 3D-Welt, z.B. können Objekte dynamisch gelöscht oder neu hinzugefügt werden.

2.3.2 Java 3D

Java 3D ist eine API (**A**pplication **P**rogramming **I**nterface) aus der Java Media Familie, um 3D-Grafikprogramme in Java zu entwickeln (sowohl Applets als auch Applications). Die Java 3D API folgt wie auch Java dem Motto „Write once, run anywhere" (Einmal schreiben, überall ausführen); resultierende Programme sind unabhängig von der Rechnerplattform und unterstützen darüber hinaus mehrere Eingabe- und Darstellungsgeräte (z.B. Eingabegeräte mit sechs Freiheitsgraden wie „headtrackers" oder am Kopf angebrachte Darstellungsgeräte). Java 3D beinhaltet einen reichhaltigen Satz an Klassen und Methoden, um Formen (Shapes) zu erzeugen, deren Verhaltensweisen zu definieren, mit dem Benutzer zu interagieren sowie Details der Darstellung (des Renderings) zu kontrollieren. Desweiteren wird eine Vielzahl an Dateiformaten unterstützt, so daß die 3D-Szenen entweder im Java-Programm erzeugt oder aus einer externen Datei geladen werden können (z.B. einer VRML-Datei). Java 3D wurde unter dem Gesichtspunkt der hohen Leistungsfähigkeit entworfen, wobei Entwicklern höhere objektorientierte Programmierparadigmen zur Verfügung gestellt werden, so daß schnell hochentwickelte 3D-Anwendungen entstehen. Hierbei können sich die Entwickler ganz auf die geometrischen Objekte und die Komposition ihrer 3D-Welt konzentrieren, ohne dabei über Dreiecke und die Details zur effizienten Darstellung der Szene nachdenken zu müssen – ganz im Gegensatz zu 3D-APIs der dritten Generation wie OpenGL. Eine zentrale Rolle hierbei spielt das sogenannte Szenengraph-Programmier-Modell, welches einen einfachen und flexiblen Mechanismus zur Verfügung stellt, um potentielle komplexe 3D-Umgebungen zu modellieren und darzustellen. Der Szenengraph enthält eine komplette Beschreibung der gesamten Szene – des virtuellen Universums. Dies beinhaltet die geometrischen Daten, die Attributinformationen sowie die Informationen bezüglich der Sichtbarkeit, um die Szene aus bestimmten Blickrichtungen darzustellen.

Desweiteren erlaubt das Szenengraph-Modell eine einfache Optimierung der Darstellung, gute Skalierbarkeit sowie Parallelität der Java 3D-API: z.B. wird durch Lockerung der strengen Darstellungsreihenfolge anderer APIs eine parallele Abarbeitung und Darstellung des Szenengraphs möglich; indem bei der Darstellung sowie den Verhaltensberechnungen der Objekte nur die im Moment sichtbaren berücksichtigt werden, kann ein sehr großes virtuelles Universum in akzeptabler Qualität dargestellt werden. Neben Optimierungen auf der API-Ebene entscheidet vor allem die Zeit für die Darstellung der sichtbaren Geometrien über das Leistungsvermögen von Java 3D. Um dies zu optimieren, sind Java 3D-Implementationen in Schichten aufgebaut, so daß Vorteile der hardwarenahen APIs (wie OpenGL, Direct3D) auf den vorhandenen Systemen genutzt werden können. Java 3D ist, wie der Name schon sagt, in Java integriert und zusammen mit anderen Java-APIs einsetzbar. Die Ausführung der Anwendungen basiert somit auf Java, so daß eine nahezu unbegrenzte Vielfalt an Verhaltensweisen der Objekte möglich wird.

2.3.3 Systemanforderungen

Zu Beginn der Gegenüberstellung von VRML, Java 3D und Quake 2 sollte zuerst einmal auf die Systemanforderungen eingegangen werden, die ein Benutzer bzw. dessen Rechnersystem erfüllen muß, um überhaupt in den Genuß der dreidimensionalen Welten gelangen zu können. Beginnen wir mit den Softwareanforderungen:

Um eine VRML-Datei darstellen zu können, wird ein VRML-Interpreter benötigt; dieser ist i.d.R. als „plug-in“ (Zusatzmodul) eines WWW-Browsers realisiert. Damit eine Skriptsprache in VRML möglich wird, müssen WWW-Browser und VRML-Interpreter diese unterstützen; um auch über Java-Anwendungen (hier i.d.R. Applets) mit der VRML-Welt interagieren zu können, muß der WWW-Browser Java unterstützen. Zusätzlich müssen zu den schon verwendeten Klassenbibliotheken die speziellen VRML-Klassenbibliotheken vorhanden sein (sind i.d.R. beim Interpreter enthalten). WWW-Browser mit Java-Unterstüzung gibt es für fast alle Rechnerplattformen/Betriebssysteme; die VRML-Interpreter sind hingegen vornehmlich für die WWW-Browser von Netscape und Microsoft erhältlich und dies darüber hinaus auch nicht für alle Rechnerplattformen, wohl aber für die bedeutenden. Somit ist VRML mit mehr oder minder großem Such- und Installationsaufwand auf den meisten Rechnern einsetzbar.

Bei Java 3D Anwendungen verhält es sich im Grunde wie bei Java-Anwendungen, nur daß hier noch die Java 3D-spezifischen Klassenbibliotheken benötigt werden. Ist die Anwendung also eine Application, wird eine Virtuelle Java Maschine benötigt, ist sie ein Applet, ist ein Java-fähiger WWW-Browser notwendig; in jedem Fall müssen alle verwendeten Klassenbibliotheken vorhanden sein. Alle angesprochenen javaspezifischen Programme und

Bibliotheken sind bei Sun Microsystems [15] bzw. JavaSoft [16] für die meisten Rechnerplattformen erhältlich.

Im Vergleich zu den meist kostenlos über das WWW erhältlichen Hilfsmitteln für VRML und Java 3D muß man das Programm Quake 2 kaufen, welches im Moment für etwa 100 DM erhältlich ist; darüber hinaus erschwert die Indizierung des Spiels in Deutschland die Suche noch zusätzlich. Die so erworbene Version des Spiels ist ausschließlich für Windows-basierte Systeme auf i386-kompatiblen Prozessoren ausgelegt; jedoch sind die Programmdateien für eine Vielzahl von Rechnerumgebungen bei id Software erhältlich (z.B. für Linux, Solaris oder Irix).

Die Hardwareanforderungen für die jeweiligen Programme sind objektiv nicht genau erfaßbar, da diese stark vom individuellen Eindruck des Betrachters sowie von der Komplexität der jeweiligen 3D-Welten abhängen. Die Mindestanforderungen nach Angaben des Herstellers von Quake 2 beschreiben sich wie folgt: Prozessor: 90 MHz Pentium (133 MHz empfohlen); Arbeitsspeicher: 16 MB (Windows 95) bzw. 24 MB (Windows NT); 100% Soundblaster kompatible Soundkarte; 4-fach CD-ROM Laufwerk; Festplattenspeicher: 25 MB (Spiel von CD), 250 MB (Spiel von Festplatte); 3D-Beschleunigerkarte empfohlen. Um sich in einer virtuellen dreidimensionalen Welt im Umfang des untersten Stockwerks der Universitätsbibliothek Karlsruhe reibungslos bewegen zu können, sind in Quake 2 ein Prozessor der Pentium 200 MHz MMX Klasse, 64 MB Arbeitsspeicher sowie vor allem eine 3D-Beschleunigerkarte unverzichtbar.

Diese Eckdaten sollten weder für VRML noch für Java 3D unterschritten werden. Im Gegenteil ist eher zu erwarten, daß in beiden Fällen eine weitaus bessere Hardwareausstattung nötig wird, um die gleiche Darstellungsqualität (gleiche Bildschirmauflösung, gleiche Bilderanzahl pro Sekunde) zu erreichen. Bei VRML ist dies damit zu begründen, daß die 3D-Welt zum einen erst zur Laufzeit interpretiert wird und zum anderen der Zugriff über einen WWW-Browser der Performanz nicht gerade zugute kommt. Bei der Implementierung von Java 3D wurde zwar viel Wert auf die Performanz gelegt, jedoch kann zur Zeit wohl niemand bestreiten, daß Java-Programme bezüglich der Geschwindigkeit im Vergleich zu C- oder Assemblerprogrammen (in diesen Sprachen ist Quake 2 geschrieben) wesentlich schlechter dastehen. Weiterhin setzt Quake 2 direkt auf die OpenGL-API auf, so daß durch individuelles Optimieren das Rendering speziell auf das Spiel abgestimmt wurde (zur internen Darstellung der 3D-Welt wurden hier – im Vergleich zum Szenengraph-Modell in Java 3D – sogenannte BSP-Bäume verwendet), wohingegen der Programmierer in Java 3D keinen Einfluß auf die Ebene der OpenGL-API hat.

2.3.4 Entwicklungsaufwand

Zum Entwicklungsaufwand in VRML und Java 3D können mangels hierfür notwendiger Erfahrungen zwar keine konkreten Angaben gemacht werden, jedoch sollen die folgenden allgemeineren Überlegungen einige Vor- und Nachteile sowie potentielle Schwierigkeiten aufzeigen. Egal für welche Umgebung (VRML, Java 3D oder Quake 2) man eine dreidimensionale Welt entwickeln möchte, sind etliche Schritte unabdingbar und könnten – vorausgestezt es gäbe ein Werkzeug mit den diversen Exportmöglichkeiten – mit ein und demselben Werkzeug durchgeführt werden. Hierbei steht zuerst die Definition der dreidimensionalen Körper und deren Plazierung im Raum an; daraufhin muß noch deren Aussehen, also die Texturen der Oberflächen, erstellt und zugeordnet werden.

Anschließend gilt es, je nach Umgebung, die Welt mit „Leben" zu füllen und sie mit Hilfe verschiedener Stilmittel realer zu gestalten, d.h. der Benutzer muß auf irgendeine Weise in der 3D-Welt navigieren und bei einer interaktiven Welt seine Umgebung beeinflussen können. Die Navigation in VRML wird größtenteils über den externen Interpreter abgewickelt, so daß hier normalerweise nur noch verschiedene „Viewpoints" zur schnelleren Navigation definiert werden (mehrere vordefinierte „Kamerapositionen", die der Benutzer i.d.R. im Interpreter auswählen kann). Möchte man jedoch andere als die vom jeweiligen Interpreter vorgegebenen Navigationsmöglichkeiten nutzen, so stehen in einfacher Weise jegliche Möglichkeiten von Java-Applets zur Verfügung, um eigene Navigationsschnittstellen zu realisieren. Dies ist natürlich mit einigem Mehraufwand verbunden, sollte aber bei den oftmals sehr eigenwilligen Bedienschnittstellen der Interpreter auch in Betracht gezogen werden. In Java 3D existieren schon einige Klassen speziell zur Benutzerinteraktion, so daß man beim Erstellen einer Navigationsschnittstelle auf einem vorhandenen Gerüst aufbauen kann. Hierbei stehen eine Vielzahl möglicher Eingabegeräte zur Verfügung, so daß der Phantasie in puncto Benutzerschnittstelle so schnell keine Grenze gesetzt wird. In Quake 2 hingegen ist die Navigationsschnittstelle fest vorgegeben und im großen und ganzen nicht veränderbar.

Für die Benutzerinteraktion sowie zur weiteren Gestaltung stehen in VRML etliche Mittel zur Verfügung z.B. verschiedene Sensoren, Bewegungsinterpolatoren, eine zeitliche Struktur, Lichteffekte, Nebel, Kollisionserkennung, Level of Detail-Funktionalität, Sound, Schalter, etc. Diese sind mit relativ geringem Aufwand in eine Szene einzubauen. Möchte man neben diesen vorhandenen Mitteln noch spezielle Funktionen hinzufügen, so steht jederzeit der Weg über Java oder für einfachere Dinge auch Javascript zur Verfügung. In Java 3D stehen vor allem zur weiteren Beeinflußung der Darstellung detaillierte Klassen und Methoden bereit; sonstige komplexere Funktionen müssen dann vollständig in Java entwickelt werden. Hier liegt der Vorteil darin, daß man an keinerlei Vorgaben gebunden ist und seine eigenen Vorstellungen von Anfang an verfolgen kann, dafür ist jedoch ein gewisser Aufwand zu betreiben. Dagegen existieren in Quake 2 wiederum vielfältige Einflußmöglichkeiten auf die 3D-Welt nach dem Baukastensystem: Türen, fahrende Plattformen, Knöpfe, sich bewegende Objekte, ... Diese sind hier mittels eines Editors schnell und leicht einzufügen. Somit läßt sich in kürzester Zeit eine ansehnliche 3D-Welt zusammenstellen. Man hat jedoch auf die grafische Darstellung der 3D-Welt keinerlei Einfluß, was aber aufgrund der guten Darstellung (Level of Detail, Licht, transparente Objekte, ...) kaum störend ist, es sei denn, man möchte in größerem Umfang 2D-Objekte darstellen, wie z.B. Texte. Will man neben den vorhandenen Elementen noch eigene Funktionalität in die Szene einbringen, so wird es auch hier schwieriger und zeitaufwendiger (siehe Kapitel 4.4).

3 Quake 2

Nachdem nun mit VRML und Java 3D zwei bekannte Modellierungswerkzeuge vorgestellt wurden, soll in den nächsten beiden Kapiteln das von uns zur Modellierung verwendete Programm, Quake 2, beschrieben werden. Kapitel 3 gibt einen einleitenden Überblick und behandelt die wichtigsten Aspekte des Spieles, unter anderem Aufbau und Struktur sowie zu berücksichtigende Einschränkungen. Danach folgt in Kapitel 4 eine Beschreibung der Modellierung eigener Welten mit Quake 2. Dort werden die dafür notwendigen Programme vorgestellt und auch einige der hier beschriebenen Komponenten noch detaillierter erläutert.

Abschließend noch eine Anmerkung: In den folgenden Kapiteln werden viele englischsprachige Fachausdrücke verwendet. Nicht alle werden im Text erklärt, bei Unklarheiten sollte daher das Glossar (Anhang A, Seite 63) zu Rate gezogen werden.

3.1 Überblick

Quake 2 ist das neueste in einer Generation von Computerspielen der Firma id Software, deren Genre man im Englischen treffend mit „3D Shooter“ oder „First Person Shooter“ bezeichnet. Diese Namen bringen auf den Punkt, worum es bei den Spielen geht: Der Spieler bewegt bzw. kämpft sich durch eine dreidimensionale, virtuelle Welt und sieht dabei das Geschehen aus der Ich-Perspektive. Etwas ausführlicher läßt sich das Spielprinzip dieses Genres wie folgt charakterisieren:

Beim Einzelspielermodus spielt eine einzige Person gegen den Computer. Meistens ist es hier das Ziel, einen mächtigen Endgegner zu besiegen. Um zu ihm zu gelangen, muß der Spieler zuerst diverse Level durchqueren und sich gegen Monster zur Wehr setzen, die sich ihm in den Weg stellen. Dafür steht ihm ein umfangreiches Waffenarsenal zur Verfügung. Außerdem müssen meist noch diverse Schalter betätigt, Schlüssel gefunden und Fallen überwunden werden.

Im Mehrspielermodus dagegen treten mehrere (bis zu 64) menschliche Spieler gegeneinander an, z.B. in einem lokalen Netzwerk oder über das Internet. In der einfachsten Variante, dem klassischen „Deathmatch“, ist es das Ziel, die anderen Spieler so oft wie möglich abzuschießen und dabei selbst möglichst lange am Leben zu bleiben. Allerdings gibt es auch zahlreiche Varianten, die mit komplexeren Regeln aufwarten; bei „Capture the Flag“ (CTF) z.B. sind die Spieler in zwei Gruppen eingeteilt und versuchen jeweils, die Flagge des gegnerischen Teams zu erobern und in ihre eigene Basis zu bringen.

Unabhängig von dem sich wenig ändernden Spielprinzip haben sich jedoch im Lauf der Zeit die Qualität der Grafikdarstellung und die Möglichkeiten zur Gestaltung der dreidimensionalen Welten stark verbessert. Bei „Wolfenstein 3D“ zum Beispiel, das 1992 das Genre der 3D Shooter begründete, waren alle Wände im rechten Winkel angeordnet, und es gab keine verschieden hohen Ebenen. Dies änderte sich im Dezember 1993, als „Doom“ veröffentlicht wurde. Erstmals gab es hier verschiedene Höhen innerhalb der Level, dadurch wurden zum Beispiel Treppen und Aufzüge möglich. Auch die Wände konnten jetzt beliebig dick sein und in beliebigem Winkel zueinander plaziert werden. Sie mußten allerdings immer senkrecht zum Boden stehen, denn Doom war nicht wirklich dreidimensional. Daraus ergaben sich noch weitere Einschränkungen, so konnte man z.B. nicht mehrere Ebenen übereinander plazieren und auch schiefe Ebenen waren nicht möglich.

Erst der Nachfolger „Quake“, der im August 1996 veröffentlicht wurde, gestattete die Gestaltung beliebiger dreidimensionaler Welten. Außerdem wurden hier Gegner, Waffen und ähnliches durch dreidimensionale Objekte (Models) dargestellt und nicht mehr durch zweidimensionale Bilder (Sprites). Der Schritt zu Quake 2 (veröffentlicht im Dezember 1997) war dann weniger groß: Die Gegner und die Waffen sind hier besser animiert, transparente Objekte (z.B. Fensterscheiben) wurden möglich und die Lichtberechnung wurde verbessert (unter anderem gibt es jetzt auch farbiges Licht). Wichtiger war jedoch eine Design-Entscheidung: Die eigentliche Spiele-Logik wurde in eine DLL-Datei ausgelagert, für die id Software den Quellcode veröffentlicht hat. Die zugrundeliegende Programmiersprache ist C, dadurch ergeben sich für einen Programmierer große Einflußmöglichkeiten.

Quake 2 ist wegen seiner Gewaltdarstellungen nicht unumstritten: Während die ELSPA (European Leisure Software Publishers Association) eine Freigabe ab 15 Jahren erteilt hat, ist Quake 2 in Deutschland indiziert. Das bedeutet, es darf nur an Personen über 18 Jahren verkauft, nicht beworben und nicht öffentlich ausgestellt werden.

Nichtsdestotrotz sind Quake und Quake 2 sehr populär und entsprechend weit verbreitet. Viele Privatpersonen beschäftigen sich auch tiefer mit Quake und konstruieren eigene Level, schreiben Tutorials oder erstellen Freeware-Programme, die sie dann über das Internet zur Verfügung stellen. Daher gibt es im WWW Hunderte von Seiten, die sich mit Quake befassen, und ohne diese Programme und Informationen, die man dort findet, wäre es unmöglich, ein Projekt wie dieses zu realisieren.

3.2 Aufbau und Struktur

3.2.1 Hardwareunterstützung und -anforderungen

Quake 2 läuft unter den unterschiedlichsten Betriebssystemen, wie z.B. Win32, Linux, Solaris oder Irix. Als Mindestanforderung nennt id Software einen Pentium-90 mit 16 MB Speicher. Wie so oft bei Mindestanforderungen liegt dieser Wert jedoch viel zu niedrig. Um eine zufriedenstellende Performanz zu erhalten, sollte es schon ein Pentium-200 MMX mit 64 MB Speicher und eine 3D-Beschleunigerkarte sein.

Für die Bilddarstellung bietet Quake 2 vier unterschiedliche Videotreiber an: Software, 3Dfx OpenGL, PowerVR OpenGL und Default OpenGL. Der Software-Renderer ist den OpenGL-Renderern allerdings in Geschwindigkeit und Bildqualität deutlich unterlegen. Außerdem vermag er es nicht, komplexe Level (wie den unsrigen) mit einer hohen Zahl an gleichzeitig sichtbaren Polygonen korrekt darzustellen. Eine 3D-Grafikkarte, die mit dem Voodoo-Chipsatz von 3Dfx ausgestattet ist, leistet deutlich mehr, sorgt für bessere Bildqualität und kostet zur Zeit (Dezember 1998) nur noch ca. 100 DM. Die 3D-Grafikkarten der nächsten Generation (mit Voodoo2- oder Riva TNT-Chipsatz) bieten noch mehr Leistung und fallen beinahe täglich im Preis. Es ist daher anzunehmen, daß solche Grafikkarten bald zur Standardausstattung eines PCs dazugerechnet werden können.

3.2.2 Zweiteilung des Programmcodes

Schon zu Zeiten von Doom war es unter den Spielern sehr beliebt, das Spiel zu erweitern oder zu verändern, z.B. durch das Erstellen von Levels oder das Einbinden neuer Grafiken. Strukturbedingt war dies allerdings nur sehr eingeschränkt möglich. id Software wollte deshalb bei Quake 1 dafür Sorge tragen, daß den Benutzern mehr Möglichkeiten offen stehen. Deshalb entschied man sich für eine Zweiteilung des Programmcodes: Die eigentliche Engine, also der Programmteil, der für Grafikberechnung usw. zuständig ist, steckt in der Datei `quake.exe`. Die Spielelogik, also die zugrundeliegenden Regeln für das Verhalten von Spielern, Monstern und Entities, ist in eine externe Datei ausgelagert (`progs.dat`). Der Quellcode und ein Compiler dafür wurden freigegeben, so daß es jedermann offen steht, Änderungen vorzunehmen. Basis des Quellcodes ist QuakeC, eine proprietäre, an C angelehnte Sprache. Dies ist auch der Nachteil dieser Lösung, denn als Programmierer ist man auf die gegeben Funktionen und Strukturen beschränkt. Deshalb wurde das Prinzip bei Quake 2 leicht geändert. Die Spielelogik steckt jetzt in einer DLL-Datei (`gamex86.dll`), die zugrundeliegende Programmiersprache ist C. Damit steht Programmierern das ganze Spektrum von C sowie Zugriffe auf Windows-Systembefehle zur Verfügung. Dadurch ist es zum Beispiel möglich, von Quake 2 aus eine TCP-Verbindung zu einem anderen Rechner aufzubauen, was essentiell für unser Projekt ist.

3.2.3 Client-Server Architektur

Doom war das erste Spiel von id Software mit Mehrspielermodus. Die Zahl der Spieler ist hier auf vier begrenzt. Jeder Computer kommuniziert mit allen anderen Computern und berechnet selbständig die aktuelle Spielsituation. Dieses Verfahren hat mehrere Nachteile: Zum einen ist es schlecht skalierbar, denn der Kommunikationsaufwand steigt quadratisch. Zum anderen kann es passieren, daß einzelne Spielzustände inkonsistent zueinander werden.

Seit Quake wird deshalb eine Client-Server-Architektur verwendet. Die Clients schicken dem Server die Eingabeereignisse, die sie vom Spieler erhalten (Mausbewegungen, Tastaturanschläge usw.). Im Gegenzug erhalten sie Pakete, die die Informationen enthalten, die zum Zeichnen des aktuellen Bildes notwendig sind. Der Kommunikationsaufwand steigt also nicht mehr quadratisch, sondern linear. Die Client-Server-Architektur gilt auch für den Einzelspielermodus, hier handelt es sich dann bei Client und Server um den selben Rechner.

Damit eine schlanke Kommunikation gewährleistet ist, ist die Länge der Pakete, die der Server verschickt, auf 1400 Byte begrenzt. Dadurch ergeben sich Einschränkungen, die für unser Projekt relevant sind und auf die in Abschnitt 3.3 näher eingegangen wird.

3.2.4 Komponenten eines Levels

Quake 2-Level bestehen aus zwei Arten von Objekten, Brushes[1] und Entities. Brushes sind konvexe geometrische Körper und bilden die eigentliche Struktur eines Levels. Bei Worldcraft gibt es vier verschiedene Grundtypen: Quader, Keil, Dorn und Zylinder. Größe, Position und Orientierung sind beliebig. Konkave Körper sind allerdings nicht erlaubt, sie müssen aus mehreren konvexen zusammengesetzt werden. Gekrümmte Oberflächen sind ebenfalls nicht möglich, siehe dazu auch Kapitel 3.3.

Entities sind die aktiven Elemente in einem Level. Man unterscheidet zwei Arten: Solid Entities sind an Brushes gebunden. Ein Beispiel hierfür sind Türen oder Plattformen. Point Entities hingegen existieren unabhängig von Brushes. Manche der Point Entities werden durch Models dargestellt (z.B. Monster oder Munition), andere sind unsichtbar (z.B. der Startpunkt des Spielers oder die Lichtquellen).

Models sind dreidimensionale Objekte, die im Gegensatz zu Brushes dynamisch sind, d.h. sie können entstehen, wieder verschwinden und sich beliebig bewegen. Sie bestehen aus einem Drahtgittermodell und einer Textur („Skin“), die darübergelegt wird. Einige Models, z.B. die Monster, sind animiert, andere sind es nicht (z.B. die Munitionsboxen). Mit einem entsprechenden Editor ist es möglich, eigene Models zu entwerfen.

Auch die Brushes werden von Texturen überzogen. Diese liegen bei Quake 2 im WAL-Format vor. Eine WAL-Datei enthält vier verschieden große Bilder einer Textur, wobei jedes halb so groß wie das vorhergehende ist. Im Spiel wird dann abhängig von der Entfernung des Blickpunktes zur Textur dynamisch zwischen diesen Bildern umgeschaltet. Beim Erstellen von Texturen muß man die Quake 2-Palette verwenden, die nur 256 Farben enthält. Außerdem ist zu beachten, daß Quake 2 nur Texturen akzeptiert, deren Höhe und Breite Vielfache von 16 sind. In einem Level können den verschiedenen Seiten eines Brushes verschiedene Texturen gegeben werden, die außerdem verschoben, skaliert und rotiert werden können. Daneben können über Flags noch bestimmte Eigenschaften der Textur ein- bzw. ausgeschaltet werden, wie z.B. die Transparenz.

[1] Bei Worldcraft wird anstatt „Brushes“ die Bezeichnung „Solids“ verwendet.

3.2.5 Kompilieren eines Levels

Leveleditoren speichern ihre Dateien im MAP-Format[2]. Dabei handelt es sich (ähnlich wie bei VRML) um Textdateien, die die Position und Ausprägung aller Brushes und Entities eines Levels enthalten. Eine genaue Spezifikation des MAP-Formats findet man bei [11].

MAP-Dateien können aber nicht direkt in Quake 2 verwendet werden, sie müssen zuerst noch kompiliert werden. Die dafür nötigen Programme wurden von id Software mitsamt des Quelltextes veröffentlicht [9]. Das Kompilieren geschieht in drei Schritten:

Der erste und wichtigste Schritt wird vom Programm QBSP3 vorgenommen. Es erzeugt aus der MAP- eine BSP-Datei, die dann von Quake 2 gelesen werden kann. BSP steht für *binary space partition*, was daran liegt, daß die Welt hier auf einen Binärbaum abgebildet wird. Für weiterführende Informationen zu BSP-Bäumen sei auf [7] verwiesen.

Das nächste Programm ist QVIS3. Im Gegensatz zu QBSP3 ist dieser Schritt (genau wie der nächste) jedoch optional, d.h. um einen Level in Quake laufen zu lassen, genügt das Kompilieren mit QBSP3. Das Verwenden von QVIS3 steigert allerdings die Performance erheblich, denn es werden hier Informationen über die Sichtbarkeit (*visibility*) im Level vorausberechnet und abgespeichert. QVIS3 bestimmt für jedes Blatt des BSP-Baumes, welche anderen Blätter von dort aus potentiell sichtbar sind. Nur diese muß die Engine dann später zeichnen. Bei einem komplexen Level können die Sichtbarkeitsberechnungen sehr lange dauern, im Extremfall sogar mehrere Tage. Detailinformationen darüber, was QBSP3 und QVIS3 tun, findet man bei [3].

Auch das letzte Programm, QRAD3, nimmt Vorausberechnungen vor, und zwar bezüglich der Beleuchtung. Der Name rührt vom verwendeten Modell her, dem *radiosity lighting*. Anhand der Lichtquellen im Level werden die dort herrschenden statischen Lichtverhältnisse vorausberechnet und in der BSP-Datei abgespeichert. Dynamisches Licht, z.B. durch Waffen oder Explosionen, wird während des Spiels zur Laufzeit berechnet.

Da zu diesen Programmen der Quellcode veröffentlicht wurde, haben einige Personen darauf aufbauend Varianten entwickelt, die sich durch größere Ausführungsgeschwindigkeit bzw. erweiterten Funktionsumfang auszeichnen. Wir haben für unser Projekt neben id Softwares QBSP3 Steven Boswells Variante von QVIS3 sowie Tim Wrights ArghRad benutzt (siehe auch Anhang C).

3.3 Beschränkungen der Quake 2-Engine

Grundsätzlich kann man zwei Arten von Beschränkungen unterscheiden: Zum einen werden durch die Engine einige Obergrenzen festgelegt, die nicht überschritten werden können,

[2] Viele Editoren speichern die Daten parallel dazu in einem eigenen Format, das zusätzliche für das Editieren relevante Informationen enthält. Bei Worldcraft tragen diese Dateien die Endung .RMF.

andererseits sind aber auch manche Dinge unter Quake 2 prinzipiell nicht möglich.

Zur zweiten Gruppe muß man zum Beispiel die Tatsache zählen, daß sich Kurven bzw. runde Flächen nicht verwirklichen lassen. Sie können bestenfalls durch Polygone angenähert werden, damit verbunden ist allerdings ein eingeschränkter Realismus. Dies trifft auch auf die Modellierung der Universitätsbibliothek Karlsruhe zu, da es hier im Erdgeschoß einige runde Formen gibt, wie z.B. die Drehtüren im Eingangsbereich und die Verglasung der Wendeltreppen. Hätte man diese Konstruktionen wirklich rund modellieren können, wäre es dem Realismus der Darstellung sicherlich zugute gekommen.

Dies gilt auch für folgende Beschränkung: Transparente Flächen, also zum Beispiel Glas, sind mit Quake 2 zwar möglich, allerdings dürfen sie nicht beweglich sein. Insbesondere lassen sich also Glastüren unter Quake 2 nicht verwirklichen. Leider gibt es mehrere davon im Eingangsbereich der Universitätsbibliothek, so daß für diese Türen alternative Texturen gewählt werden mußten.

Auch reflektierende Oberflächen sucht man bei Quake 2 vergebens. Obwohl es in der Bibliothek einige spiegelnde Metallflächen gibt, fällt diese Einschränkung aber nicht so sehr ins Gewicht wie die zuvor genannten.

Viel drastischer wirkt sich dagegen aus, daß Quake 2 nur mit einer Palette von 256 Farben arbeitet. Offenbar hat sich id Software aus Performance-Gründen dazu entschlossen, dies stellt allerdings eine große Einschränkung beim Erstellen von Texturen dar. Im Vergleich mit dem Spiel „Unreal“, das mit 32.000 Farben arbeitet, zeigt sich, daß man durch mehr Farben eine realitätsgetreuere Darstellung erreichen kann.

Unreal ist auch noch in weiteren Punkten Quake 2 technisch überlegen, wird aber wahrscheinlich seinerseits von „Quake 3: Arena“, dem geplanten Nachfolger von Quake 2, überboten werden. Mehr dazu findet sich in Kapitel 5.3.

Neben diesen gerade beschriebenen Einschränkungen gibt es aber auch noch eine zweite Art. Hier ist man an interne Grenzen gebunden, die die Engine nicht überschreiten kann. Ein Beispiel dafür ist die Ausdehnung eines Levels, hier liegt das Maximum bei -4096 bis +4096 Längeneinheiten (jeweils in X-, Y- und Z-Richtung)[3]. Für unser Projekt ist dies irrelevant, die Ausdehnung beträgt hier 2552 x 2072 x 2784 Einheiten, stößt also bei weitem nicht an diese Grenzen.

Anders sieht das bei den darzustellenden Polygonen aus, hier überschreiten wir die empfohlenen Grenzen deutlich. Mit dem Konsolenkommando `r_speeds 1` kann man Informationen zur aktuellen Belastung der Engine anzeigen lassen. Eine typische Ausgabe lautet im GL-Modus z.B. `557 wpoly  506 epoly  13 tex  0 lmaps`. Die Zahl vor `wpoly` zeigt hier an, daß momentan 557 Polygone gezeichnet werden müssen. In verschiedenen Tutorials über Leveldesign wird geraten, den Level so zu konstruieren, daß dieser Wert im Schnitt bei 600 bis 700 liegt und 1000 nicht überschreitet. In unserem Level liegt der Spitzenwert allerdings bei 2900 (vor der Verwendung von QVIS3). Dies ist auch der Grund dafür, daß unser Level nicht mit dem Software- Renderer darzustellen ist. Er ist damit überfordert und

[3] QBSP3 schneidet alles ab, was darüber hinaus geht.

zeigt viele Flächen untexturiert. Außerdem verschwinden Entities wie Türen oder Knöpfe. Mit einem GL-Renderer verläuft die Darstellung dagegen problemlos.

Eine andere Begrenzung wirkt sich allerdings stärker auf unser Projekt aus. Wie wir durch Versuche festgestellt haben, kann die Quake 2-Engine leider nur 128 Entities gleichzeitig darstellen. Entsprechend mußten wir die Anzahl der Bücher beschränken, die der Benutzer im Recherche-Raum sehen kann. Für die drei Regale sind jeweils drei Entities für die Regalböden erforderlich, drei für die Ziffern, zwei für die Knöpfe und eines für die Beschriftung. Dies sind bereits 27 Entities. Dazu kommen ein Model für den Computer und zwei für die Waffe des Spielers. Weiterhin werden noch zwölf Entities für die Knöpfe auf der Auswahltafel, zwei für die Legenden sowie zwei für die Tür benötigt. Damit verbleiben für die Bücher noch 82 Entities. Da im Maximalfall zwei Buchreihen pro Regal sichtbar sind, können maximal dreizehn Bücher pro Reihe dargestellt werden, also maximal 26 Bücher je Regal. Dies stellt eine starke Einschränkung dar, denn gerade bei großen Ergebnismengen wäre es wünschenswert, mehr Bücher auf einmal darzustellen. Leider ist dieses Verhalten jedoch fest im Programmcode implementiert. Grund ist die Beschränkung der Paketlänge des Servers: Ab einer gewissen Zahl an sichtbaren Entities genügen die gegebenen 1400 Byte nicht mehr, um alle notwendigen Informationen zu übertragen. Deshalb stellen 128 gleichzeitig sichtbare Entities ein nicht überwindbares Maximum dar.

Ein Blick auf die im Quellcode enthaltene Datei `qfiles.h` offenbart weitere Obergrenzen: Die Gesamtzahl an Models in einem Level ist auf 1024 begrenzt. Für Brushes liegt das Maximum bei 8192, für Entities bei 2048. In unserem Level liegen diese Werte bei 1398 und 383[4], also weit vom maximal Möglichen entfernt. Man muß jedoch bedenken, daß es sich bei den gegebenen Grenzen um theoretische Maximalwerte handelt. Es ist anzunehmen, daß die Quake 2-Engine schon früher an ihre Grenzen stößt.

[4] zuzüglich der dynamisch zur Laufzeit erzeugten Entities

4 Konzeption und Implementierung eines eigenen Levels

Im vorigen Kapitel wurden der Aufbau und die Struktur von Quake 2, die grundlegenden Einflußmöglichkeiten auf das Spiel sowie die Beschränkungen vorgestellt, die zu beachten sind. In diesem Kapitel soll nun die prinzipielle Vorgehensweise bei der Erstellung eines eigenen Levels aufgezeigt werden und zwar am Beispiel der Modellierung der beiden unteren Stockwerke des Gebäudes der Universitätsbibliothek Karlsruhe einschließlich der Schnittstelle zum Benutzeragenten des UniCats-Projektes.

Das Kernstück eines jeden Levels ist die BSP-Datei, welche Informationen bezüglich der Brushes (Ort, Abmessung, Texturen und deren besonderen Eigenschaften) und der Entities (Ort, Initialisierungswerte der Variablen) enthält. Um diese zu erzeugen, genügen die in den Original-Levels benutzten – und damit im Spiel schon enthaltenen – Texturen, Models und Entities. Möchte man aber, wie in unserem Fall, einen individuelleren Level konzipieren, so müssen noch einige Elemente bereitgestellt und teilweise auch implementiert werden. In diesem Kapitel werden nun die einzelnen Arbeitsschritte sowie die dafür von uns benutzten Werkzeuge kurz vorgestellt; hierbei erscheinen programmspezifische Vorgehensweisen in der Form Ort/Dialogelement: weitere Optionen bzw. Menü/Untermenü/Menüpunkt: weitere Optionen. Für die Modellierung der 3D-Welt dient der Level-Editor *Worldcraft*, für die Texturerstellung die Bildbearbeitungssoftware *Adobe Photoshop* sowie das speziell für Quake 2-Texturen entwickelte Programm *Wally*. Die Quelltexterweiterungen werden mit Hilfe des Entwicklungswerkzeugs *Microsoft Visual C++* durchgeführt und die Models mit Hilfe des 3D-Modellierungsprogramms *3D Studio Max* sowie dem speziell für Quake 2-Models entwickelten Programm *Quake 2 Model Editor* konstruiert. Die jeweiligen Versionsnummern bzw. die Internetadressen zu den einzelnen Programmen sind Anhang C zu entnehmen.

4.1 Modellierung der Bibliothek: Teil 1

Um ein reales Gebäude modellieren zu können, werden sowohl Daten bezüglich des Grundrisses benötigt, als auch ein geeigneter Maßstab für dessen Umsetzung in das Modell. Damit man einen ersten Eindruck der Dimensionen in einem Quake 2-Level erhält, sei erwähnt, daß die Spielfigur 32 Einheiten (im folgenden u für units), also 32 u, breit und dick sowie 56 u hoch ist; desweiteren kann der Spieler bis zu 225 u weit und bis zu 42,5 u hoch springen, er kann Stufen bis zu einer Höhe von 17 u ohne zu springen überwinden, er kann bis zu 275 u in die Tiefe fallen, ohne verletzt zu werden, und ab 36 u Abstand zwischen zwei Brushes hindurchfallen. Diese Eckdaten sollte man im Hinterkopf behalten, vor allem beim Erstellen von Treppen, Türen, … Im Fall der Bibliothek (Abb. 4.1) ergibt sich in etwa ein Maßstab von 75 u pro Meter Länge und Breite sowie von 60 u pro Meter Höhe. Dieser ist durch den initialen Wert 100 u pro Meter und anschließendes Testen des optischen Eindrucks verschieden skalierter Level entstanden.

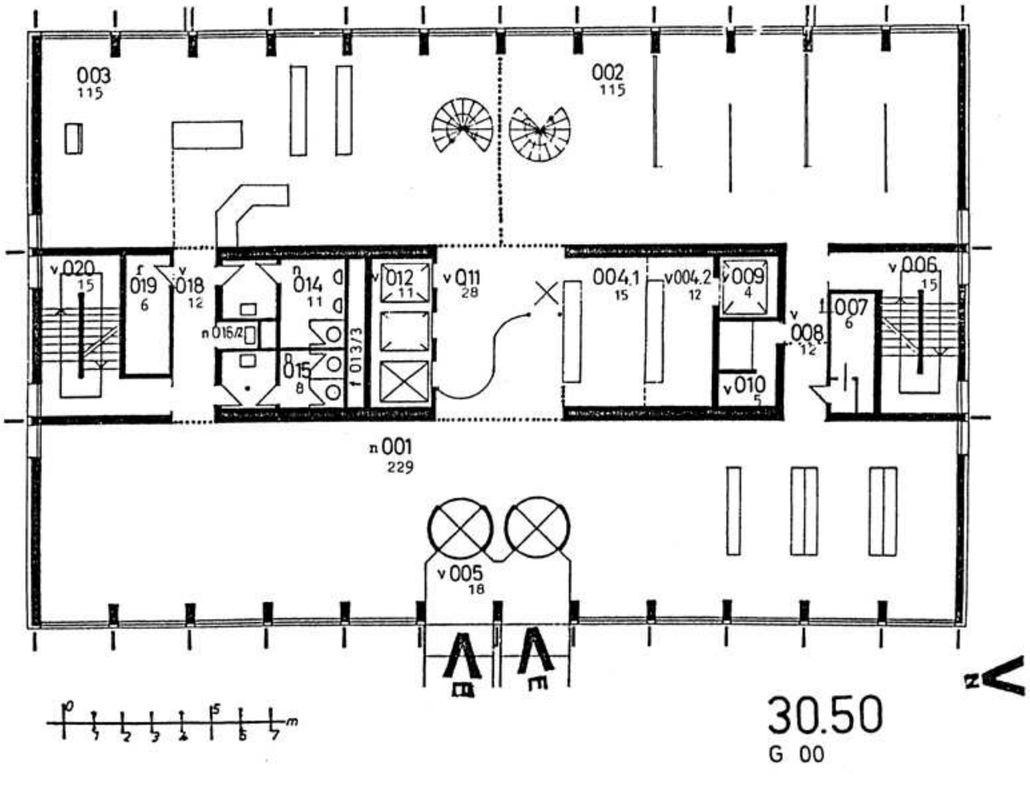


Abbildung 4.1: Grundriß des Erdgeschosses der Universitätsbibliothek Karlsruhe

Nun wird der Level Brush für Brush, Entity für Entity aufgebaut. In Worldcraft stehen vier Ansichten zur Verfügung, wie sie von den meisten 3D-Design-Programmen verwendet werden; im Uhrzeigersinn links oben beginnend: 3D-Ansicht, Aufsicht, Seiten- und Frontansicht (Abb. 4.2). Mit Hilfe von Maptools Bar/Block Tool werden die Brushes erstellt; hierzu wird der Grundtyp (Quader block, Keil wedge, Zylinder cylinder oder Dorn spike) und eventuell dessen Oberflächenanzahl (bei Dorn und Zylinder) bei Object Bar/Objects bzw.

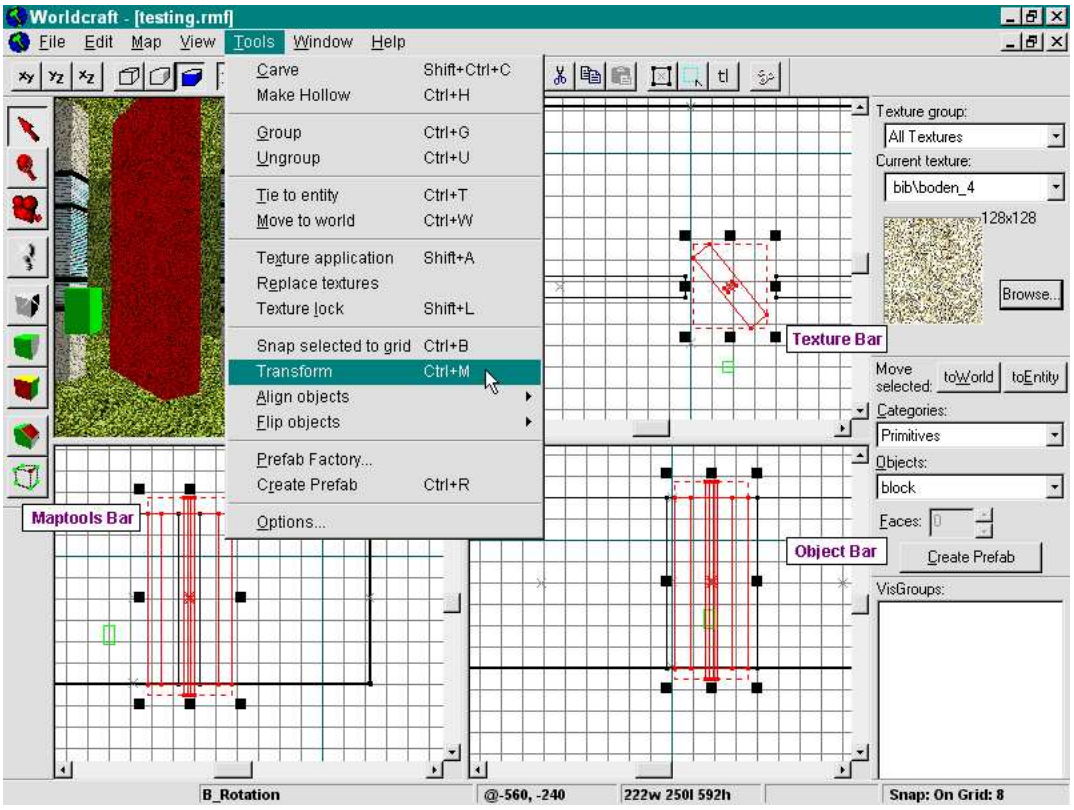


Abbildung 4.2: Level-Editor Worldcraft

Object Bar/Faces eingestellt. Es wird eine initiale Textur für die Oberflächen ausgewählt (Texture Bar/Current texture) und anschließend die Größe des Objektes in den 2D-Ansichten festgelegt. Der hierzu verwendete, den Brush umschließende Quader rastet an einem Gitter (grid) ein, dessen Weite zwischen 2 u und 512 u in Zweierpotenzen variiert werden kann. Im weiteren steht eine Vielzahl an Möglichkeiten bereit, um die Geometrie der entstandenen Brushes weiterzubearbeiten: Tools/Transform/Scale (skalieren)[1], Tools/Transform/Rotate (drehen)[2], scheren[3], Tools/Flip Objects (spiegeln), Tools/Transform/Move (verschieben)[4], Tools/Hollow (aushöhlen), Maptools Bar/Clipping Tool (ab- und durchschneiden), Maptools Bar/Vertex Tool (Eckpunkte oder Kanten der einzelnen Objekte manipulieren) und Tools/Carve (Brushes voneinander subtrahieren, also z.B. in einen Quader ein zylinderförmiges Loch schneiden, indem ein Zylinder in einen Quader geschoben wird und das Zylinder-

[1] auch nach Mausklick auf Objekt in einer der 2D-Ansichten durch Ziehen der Ecken
[2] auch in 2D-Ansicht nach doppeltem Mausklick auf Objekt
[3] in 2D-Ansicht nach dreifachem Mausklick auf Objekt
[4] auch Drag & Drop in 2D-Ansicht

volumen ausgeschnitten wird). Bei Anwendung der letzten beiden Werkzeuge ist Vorsicht angesagt. Mit dem *Vertex Tool* lassen sich ungültige Brushes erzeugen (es sind nur konvexe Brushes erlaubt). Durch das *Carving* wird das verbleibende Objekt eventuell in sehr viele kleine Brushes unterteilt (damit keine ungültigen Brushes entstehen). Dies kann aber zu Fehlern bei der Ausführung von QBSP3 führen. Mit diesen Mitteln werden die Böden, Wände, Fenster, Türen und sonstigen Elemente Stück für Stück aus einzelnen Brushes zu Räumen und komplexen Objekten zusammengesetzt. Dabei sollte vor allem durch sorgfältiges Arbeiten sichergestellt werden, daß im Level keine leaks (Lecks) entstehen. Dies sind offene Stellen vom umgebenden Freiraum direkt in die Räumlichkeiten des Levels, die im Nachhinein oftmals schwer zu finden sind. QBSP3 kann sonst den Level nicht komplett bearbeiten (bricht mit Fehlermeldung „ **** leaked **** “ ab).

Ebenso wie die Geometrie der Brushes kann auch deren Textur bearbeitet werden, um aber hierbei auf eigene Texturen zurückgreifen zu können, müssen diese zuerst einmal erstellt werden. Abbildung 4.3 zeigt den Level, bevor die eigenen Texturen Verwendung finden.

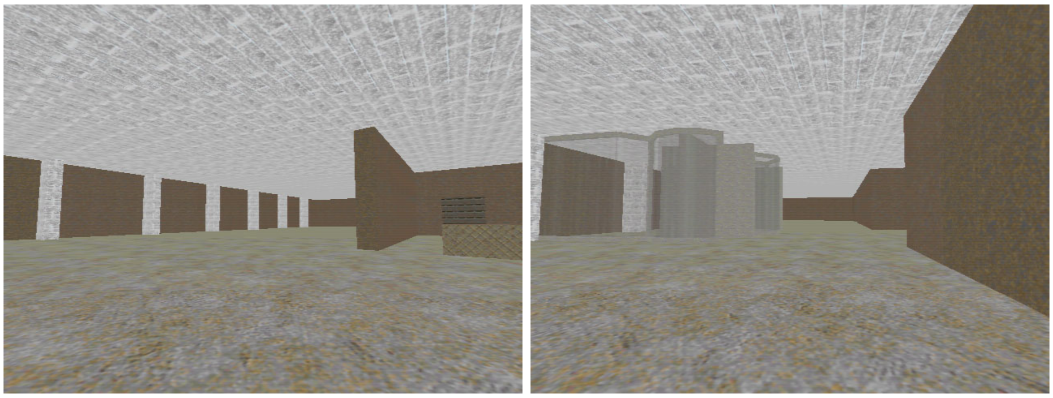


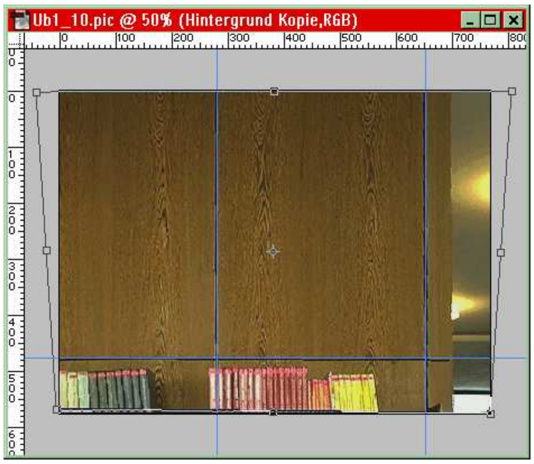
Abbildung 4.3: Level ohne eigene Texturen

4.2 Erstellen von Texturen

Texturen spielen eine entscheidende Rolle für das Erscheinungsbild eines Levels, denn sie bestimmen das Aussehen aller Objekte. Um eigene Texturen zu erstellen, muß zuerst eine Pixelgrafik mit den 256 Farben der Quake 2-Farbtabelle erzeugt werden, die mit Quake 2-spezifischen Werkzeugen in das intern verwendete WAL-Format umgewandelt werden kann. Unser Vorgehen dabei beschreibt sich wie folgt:

Von den gewünschten Oberflächen werden mit einer Kamera Fotos aufgenommen, wobei auf die Entfernung zur Struktur zu achten ist (nicht zu nah, aber auch nicht zu weit entfernt). Da dies aber zu Anfang nur schwer einschätzbar ist, nimmt man am besten mehrere Bilder aus unterschiedlichen Entfernungen auf, um später auf verschiedene Fotos zurückgreifen zu können (Daumenregel: aus „normaler“ Blickentfernung zum Objekt fotografieren – den Boden also aus Standhöhe, . . .).

(a) Originalbild



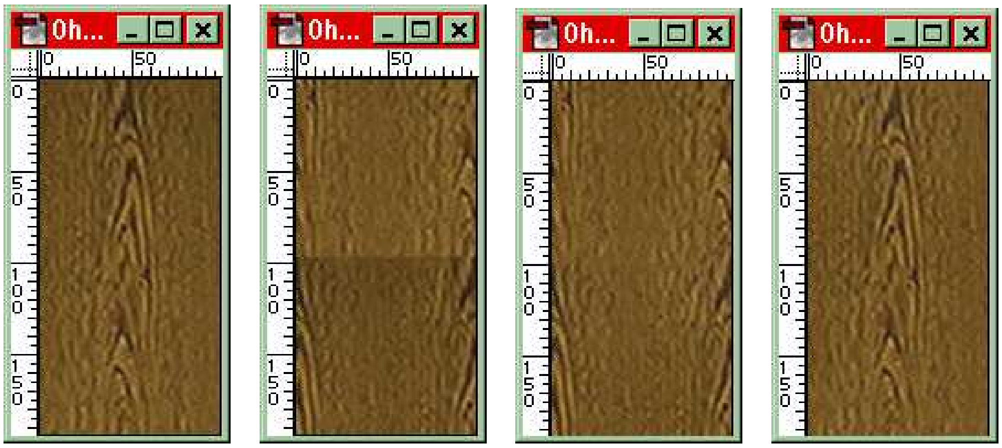
(b) beim Entzerren

(c) nach Entzerren

(d) Bereichsauswahl

Abbildung 4.4: Entzerren des Bildes

Danach wird das Bildmaterial digitalisiert, z.B. durch Einscannen, und anschließend mit einem Grafikprogramm (z.B. Adobe Photoshop) weiterbearbeitet (Abb. 4.4 a). Durch das Fotografieren entstandene, unerwünschte Verzerrungseffekte werden mit Bearbeiten/Transformieren/Verzerren kompensiert (Abb. 4.4 b, Abb. 4.4 c). Der Teil des Bildes, der schließlich als Textur dienen soll, wird markiert, kopiert und in ein neues Bild von der Größe der Auswahl eingefügt (Abb. 4.4 d, Abb. 4.5 a).



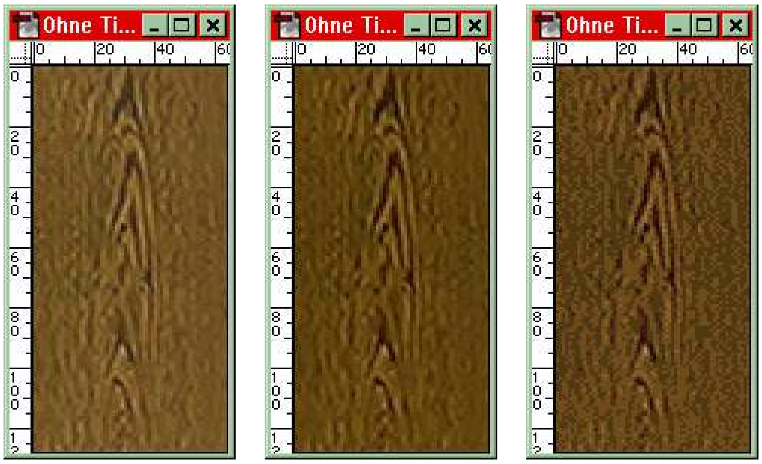
(a) (b) (c) (d)

Abbildung 4.5: Korrektur der Kanten

Jetzt muß dafür gesorgt werden, daß beim Aneinanderlegen der Textur keine Ränder und sichtbare Übergänge entstehen. Dazu wird das Bild horizontal sowie vertikal um die Hälfte der Bildbreite bzw. -höhe verschoben und die entstehenden Leerräume mit den aus dem Bild herausgeschobenen Teilen aufgefüllt (Abb. 4.5 b) (Filter/Sonstige Filter/Verschiebungseffekt: Durch verschobenen Bereich auffüllen). Die nun erkennbaren Übergänge können von Hand retuschiert (Abb. 4.5 c) und das Bild wieder in die Ausgangsposition zurückverschoben werden (Abb. 4.5 d).

Anschließend wird die Bildgröße angepaßt (Abb. 4.6 a) (Bild/Bildgröße... Beachte: Höhe und Breite müssen Vielfache von 16 Pixel sein), bei Bedarf die Helligkeits- und Farbeinstellungen korrigiert (Abb. 4.6 b) (Bild/Einstellungen/) und das Bild mit Bild/Modus/Indizierte Farben: Eigene Farbtabelle laden an die Quake 2-Farbpalette angepaßt (Abb. 4.6 c) (Die Quake 2-Farbpalette für Photoshop ist z.B. bei Wally als `q2pal.act` enthalten; am besten ersetzt man auch gleich die Farbtabelle in der Farbfelderanzeige von Photoshop mit dieser Farbpalette. Beachte: Die Quake 2-Farbpalette enthält auch einige Farbfelder mit spezieller Funktion wie z.B. Farben, die im Dunkeln leuchten oder „durchsichtig“ sind, also im Spiel nicht dargestellt werden).

Abschließend wird das Bild im PCX-Format abgespeichert und mit Hilfe von Wally die WAL-Datei erzeugt. Hierzu wird in Wally mit File/Convert... die PCX-Datei geladen, dann mit Edit/Remip die vier verschiedengroßen Bilder der Textur erstellt und zuletzt mit File/Save



(a) (b) (c)

Abbildung 4.6: Größen-, Helligkeits- und Palettenanpassung

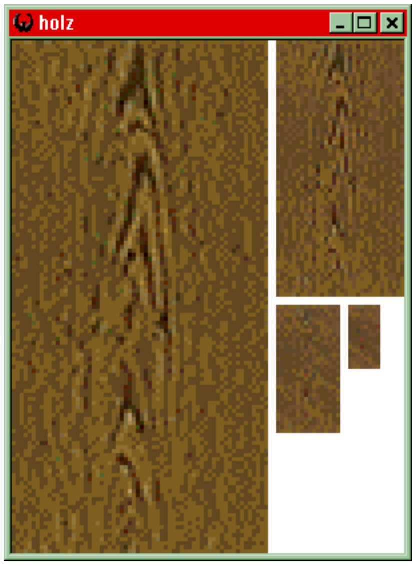


Abbildung 4.7: Textur in Wally nach Remip

die WAL-Datei gespeichert (Abb. 4.7) (Achtung: Um in Worldcraft keine Probleme mit den Texturbezeichnungen zu bekommen, sollte der Dateiname der Texturen nur aus Kleinbuchstaben bestehen).

Nun kann das Aussehen der Textur im Spiel getestet werden. Mit Wally können auch weitere Parameter der WAL-Datei angepaßt sowie Texturen allgemein editiert werden. So kann man z.B. spezielle Textur-Flags voreinstellen, Einschußlöcher hinzufügen, den Namen der Folgetextur für im Spiel animierte Texturen angeben, Helligkeits-, Blend- und Rauscheffekte anwenden sowie grundlegende Operationen durchführen (Rotieren, Spiegeln, Zeichnen, Malen, Auffüllen etc.). Bei Bedarf können im Bildbearbeitungsprogramm so lange Änderungen vorgenommen werden, bis das Ergebnis zufriedenstellend ist.

4.3 Modellierung der Bibliothek: Teil 2

Nachdem alle benötigten Texturen erstellt sind, können die Wände, Fenster, Schränke und anderen Objekte, die aus Brushes bestehen, nun in Worldcraft mit der jeweiligen Textur versehen werden. Dazu existieren folgende Möglichkeiten: Maptools Bar/Apply current texture weist sämtlichen Oberflächen der markierten Objekte die momentan ausgewählte Textur zu; Maptools Bar/Toggle texture application öffnet ein Fenster zur spezielleren Texturbearbeitung (Abb. 4.8). Damit können an einer einzelnen oder auch an mehreren Oberflächen gleichzeitig texturspezifische Änderungen vorgenommen werden: Texturen und Oberflächen auswählen, Werte der voreingestellten Flags ändern, Texturen mit oder ohne Flags der Auswahl zuweisen, Texturen der Auswahl skalieren, verschieben oder rotieren. Das Ergebnis solcher Anstrengungen kann in Abbildung 4.9 begutachtet werden.

Im weiteren müssen noch die Entities festgelegt werden. Point Entities werden mit Hilfe von Maptools Bar/Entity Tool in den Level eingefügt; hierzu wird die betreffende Point Entity-Klasse über Object Bar/Objects ausgewählt und in den 2D-Ansichten plaziert. Die Point Entities werden im Editor als Quader dargestellt, so daß sie später wie die Brushes jederzeit auswählbar und versetzbar sind. Um Solid Entities, die ja gleichzeitig auch Brushes sind, zu erhalten, markiert man die entsprechenden Brushes und bindet sie mit Tools/Tie to entity bzw. Object Bar/toEntity an ein Entity. Nun erscheint das Fenster mit den Objekteigenschaften, welches für jedes Entity, also auch für Point Entities, jederzeit über Edit/Properties erreichbar ist. In dessen Registerkarte Class Info lassen sich die Klasse des Entities über Class einstellen und die Werte der jeweiligen Variablen im Bereich Keyvalues eintragen; die Flag-Variablen werden in der Registerkarte Flags gesetzt (Abb. 4.10). Hierbei bietet Worldcraft eine sogenannte Smart Edit-Funktionalität an, bei der anstatt der originalen Variablennamen i.d.R. leichter verständliche Variablennamen angezeigt werden und so dem Benutzer die Eingabe erleichtern. Desweiteren sind in einigen Variablen schon Initialisierungswerte eingetragen oder man kann ihnen durch Pulldown-Menüs Werte zuweisen, anstatt sie einzutippen.

Das Zusammenwirken und Verhalten von Entities soll nun kurz am Beispiel einer func_door (Tür) verdeutlicht werden, wobei hier nur die Smart Edit-Variablennamen Verwendung finden (Abb. 4.10). Mit dem Solid Entity func_door wird eine Tür realisiert, die öffnet, indem sie sich um (ihre Breite − Lip remaining after move) in die von Angle angegebene Richtung

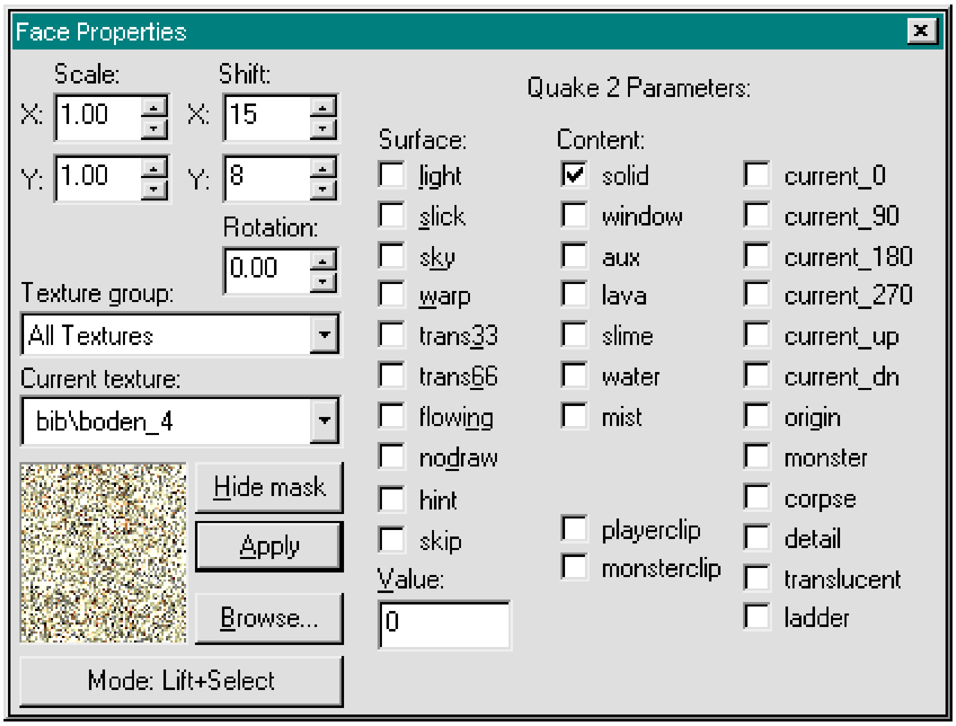


Abbildung 4.8: Texturbearbeitung in Worldcraft

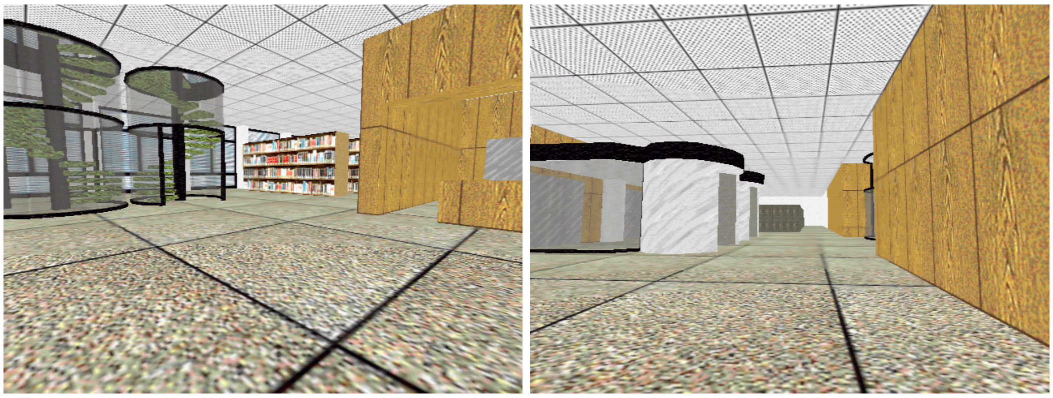


Abbildung 4.9: Level mit eigenen Texturen und zusätzlichen Brushes (vgl. Abb. 4.3)

mit der Geschwindigkeit Speed verschiebt, wenn sich ihr ein Spieler oder Monster nähert. Entfernt sich derjenige wieder von der Tür, so schließt sie nach Wait before close Sekunden. Wird ihr ein Name gegeben, so öffnet sie nicht automatisch bei Annäherung, sondern wartet, bis sie von einem anderen Entity ausgelöst wird (sie ist triggerable). Versucht ein Spieler

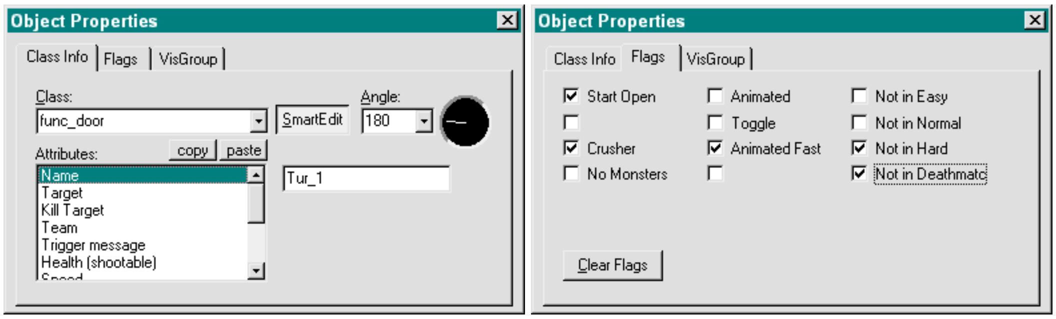


Abbildung 4.10: Entity-Objekteigenschaften

diese Tür „normal“ zu öffnen, so wird die Trigger message auf dessen Bildschirm ausgegeben (z.B. „Sie müssen erst den Knopf drücken!“). Diese Tür schließt ebenfalls Wait before close Sekunden später; sollte nun aber noch der Spieler oder ein Monster sie blockieren, so wird diesem Damage when blocked Schaden zugefügt und die Tür öffnet wieder, um nach Wait before close Sekunden erneut zu schließen. Steht die Variable Sounds auf Audible, so gibt die Tür bei der Bewegung Geräusche von sich, steht sie auf Silent, dann nicht. Wird der Tür in Health (shootable) eine Anzahl „Gesundheitspunkte“ übergeben, so kann sie nur durch mehr oder minder langen Beschuß geöffnet werden. Bekommt sie einen Team-Namen, so wird sie mit allen Türen desselben Team-Namens zu einer gemeinsamen Einheit verbunden; alle diese Türen öffnen dann gleichzeitig. Alle Entities, deren eigener Name dem in Target bzw. Kill Target angegebenen entsprechen, werden von der Tür beim Öffnen ausgelöst (sie werden ge-triggered) bzw. aus dem Level entfernt (z.B. irgendwelche Wände func_wall). Bei den Flag-Variablen sind bei allen Entities vier standardmäßig besetzt mit Not in Easy, Not in Normal, Not in Hard und Not in Deathmatch, welche ausdrücken, in welchen der verschieden Spiel-Schwierigkeitsgraden/Modi das jeweilige Entity vorhanden ist und in welchen nicht. Die restlichen acht Flags sind von der jeweiligen Entity-Klasse abhängig. In der Klasse func_door sind das folgende: Start Open: die Tür ist zu Beginn im offenen Zustand; Crusher: die Tür öffnet sich nicht wieder, wenn sie beim Schließen blockiert wird, sondern fügt dem Blockierenden solange Schaden zu, bis dieser nicht mehr im Weg ist (entweder von selbst weggeht oder „zerquetscht“ ist); No Monsters: Monster können die Tür nicht öffnen; Animated: die Textur der Tür wechselt automatisch mit 2 Hz zur jeweiligen Folgetextur (siehe Kapitel 4.2); Animated Fast: dto. mit 10 Hz; Toggle: die Tür wartet im geschlossenen *und* offenen Zustand auf Auslösung durch ein anderes Entity (nur sinnvoll bei Angabe von Name).

Möchte man in einem Level neben den beim Spiel beinhalteten Entities noch Objekte mit eigener Funktionalität benutzen, wie z.B. komplexe Fahrstühle, Bücher, oder sogar eine Kommunikationsverbindung zu anderen Rechnern aufbauen, so muß die Spielelogik an die individuellen Bedürfnisse angepaßt werden.

4.4 Erweiterungen des Quelltextes

Der frei erhältliche Quellcode der game dll (siehe [9]) kann mit Hilfe geeigneter Werkzeuge erweitert und/oder verändert und somit zu einer individuellen `gamex86.dll` kompiliert werden. game dll entspricht hierbei einer Verallgemeinerung der Bezeichnung `gamex86.dll`: auf Windows-Plattformen für die x86-Prozessorarchitektur wird die game dll zur `gamex86.dll`-Datei kompilert; auf Windows-Plattformen für die Alpha-Prozessorarchitektur heißt die resultierende Datei `gameaxp.dll`; die auf x86-Linux portierte game dll wird wiederum zur Datei `gamex86.so` kompiliert. Hierzu benutzen die id-Programmierer die Entwicklungsumgebung Microsoft Visual C++ v5.0, so daß bei den Quelldateien auch die dazugehörige Projektdatei enthalten ist. Somit ist es auch unerfahrenen Programmierern möglich, in kürzester Zeit eine lauffähige Version der game dll zu erzeugen. Selbstverständlich können auch andere Entwicklungswerkzeuge benutzt werden (insbesondere auch kostenlos erhältliche); weitergehende Informationen hierüber sind auf den Internetseiten zur Quake 2-DLL-Programmierung zu erhalten (Anhang D).

Bei der Arbeit mit dem Quelltext müssen zwei grundlegende Tatsachen berücksichtigt werden. Zum einen bauen die angestrebten Änderungen auf rund 44.000 Zeilen C-Quellcode auf, bei dem (bis auf die Kommentare im Quelltext) keine Dokumentation enthalten ist. Zum anderen ist der Quellcode der game engine nicht erhältlich, was den Einfluß auf die Funktionen der Engine auf die im DLL-Quellcode definierte Schnittstelle beschränkt und das Debuggen der eigenen Erweiterungen verhindert. Jedoch wird auf den oben bereits erwähnten Internetseiten umfangreiche Hilfestellung angeboten, es sind dort z.B. etliche Tutorials zu verschiedenen Programmaspekten oder auch Programm-informationen allgemeiner Art erhältlich. Desweiteren wird in Anhang B.3 versucht, eine allgemeine Einführung in bedeutende Teile des Original-DLL-Quellcodes zu geben. Dort werden auch die im Zuge dieser Arbeit entstandenen Quelltext-Erweiterungen genauer erläutert, auf welche im folgenden kurz eingegangen wird.

4.4.1 Objekte der Bibliothekumgebung

Die in der Universitätsbibliothek Karlsruhe vorhandenen Fahrstühle sowie die Drehtüren aus dem Eingangsbereich lassen sich nicht oder nur unzureichend mit vorhandenen Entities modellieren. Ein realer Fahrstuhl hat Knöpfe und Türen innerhalb des fahrenden Gehäuses, so daß diese sowohl als funktionsfähige Einzelentities Knopf und Tür als auch als Teilentities des sich bewegenden Fahrstuhlteils fungieren müßten. Es ist jedoch unmöglich, einem Brush mehrere Entities zuzuordnen; somit wäre es mit vorhandenen Entities nicht möglich, einen realistischen Fahrstuhl zu modellieren. Bei den Drehtüren besteht das Problem darin, daß das schon vorhandene Entity func_rotating sich entweder von selbst oder gar nicht dreht, wobei die Drehtür sich eigentlich auf Drücken des Benutzers hin in Bewegung setzen soll. Also bleibt nur der Ausweg über die Entwicklung eigener Entities.

Abbildung 4.11: Fahrstuhl-Entity

Die implementierten Entity-Klassen, mit deren Hilfe man einen Fahrstuhl (Abb. 4.11) realisieren kann, beschreiben sich wie folgt:

B_Fahrstuhl: das fahrende Fahrstuhl-Gehäuse, welches die ganze Fahrstuhllogik beinhaltet

B_Fahrstuhl_Tuer: zwei Außentüren je Stockwerk sowie zwei Innentüren im Gehäuse; die Innentüren öffnen und schließen bei Fahrtende am jeweiligen Stockwerk zusammen mit den dortigen Außentüren

B_Fahrstuhl_Knopf: im Innern des Fahrstuhl-Gehäuses für jedes Stockwerk ein Innenknopf und auf jedem Stockwerk ein Außenknopf; durch deren Betätigung nimmt der Fahrstuhl einen Fahrauftrag entgegen, den er zu geeigneter Zeit ausführt (ist er bereits unterwegs und fährt gerade an diesem Stockwerk vorbei, so hält er an; steht er gerade, so beginnt er, sich in Richtung dieses Stockwerks zu bewegen)

B_Fahrstuhl_Stock: je Stockwerk ein Point Entity; diese markieren die Haltepunkte des Fahrstuhls

Die neue Entity-Klasse B_Rotation ist eine Erweiterung der originalen func_rotating. Dient hier die Z-Achse als Drehachse und wird das Flag antriebslos gesetzt, so dreht sich das betreffende Entity nicht mehr von alleine, sondern muß vom Spieler angestoßen werden, damit es für kurze Zeit in Drehbewegung versetzt wird (wie eben bei einer Drehtür). Wird zusätzlich das Flag eine Richtung (sowie Reverse) gesetzt, so ist eine Drehung nur gegen (mit) dem Uhrzeigersinn möglich. Mit Name versehen kann die Drehbarkeit an- und abgeschaltet werden mittels Auslösen durch ein anderes Entity (wie bei einem Drehkreuz mit Sperrfunktionalität). Desweiteren kann mit B_Rotation eine Rotation um mehrere Koordinatenachsen „gleichzeitig“ realisiert werden (mit relativen Geschwindigkeitsangaben, z.B. während zwei Umdrehungen um die X-Achse zehn mal um die Z-Achse).

4.4.2 Funktionale Schnittstelle

Wenn für die soeben angeführten Objekte, Fahrstuhl und Drehtür, auch eine Modellierung mit den Original-Entities möglich wäre, so ist dies für die Realisierung der Literaturrechercheschnittstelle ganz ausgeschlossen. Deren Funktionalität liegt weit außerhalb der Möglichkeiten existierender Entities, so daß eine eigene Implementierung vonnöten ist.

Abbildung 4.12: Eingabemaske und zugehörige Hilfe

Damit der Benutzer eine Suchanfrage überhaupt abschicken kann, muß er zuerst seine Suchparameter eingeben können, d.h. was und wo er suchen will. Hierfür steht eine Eingabemaske zu Verfügung. Diese erscheint nach Berührung des Entities B_Computer (einem Model, das einen Computer darstellt) durch den Benutzer. Die Eingabemaske umfaßt fünf textuelle Eingabefelder (Freitext, Autor, Titel, ISBN, Jahr) sowie ein Auswahlfeld pro aufgeführtem Katalog (Abb. 4.12).

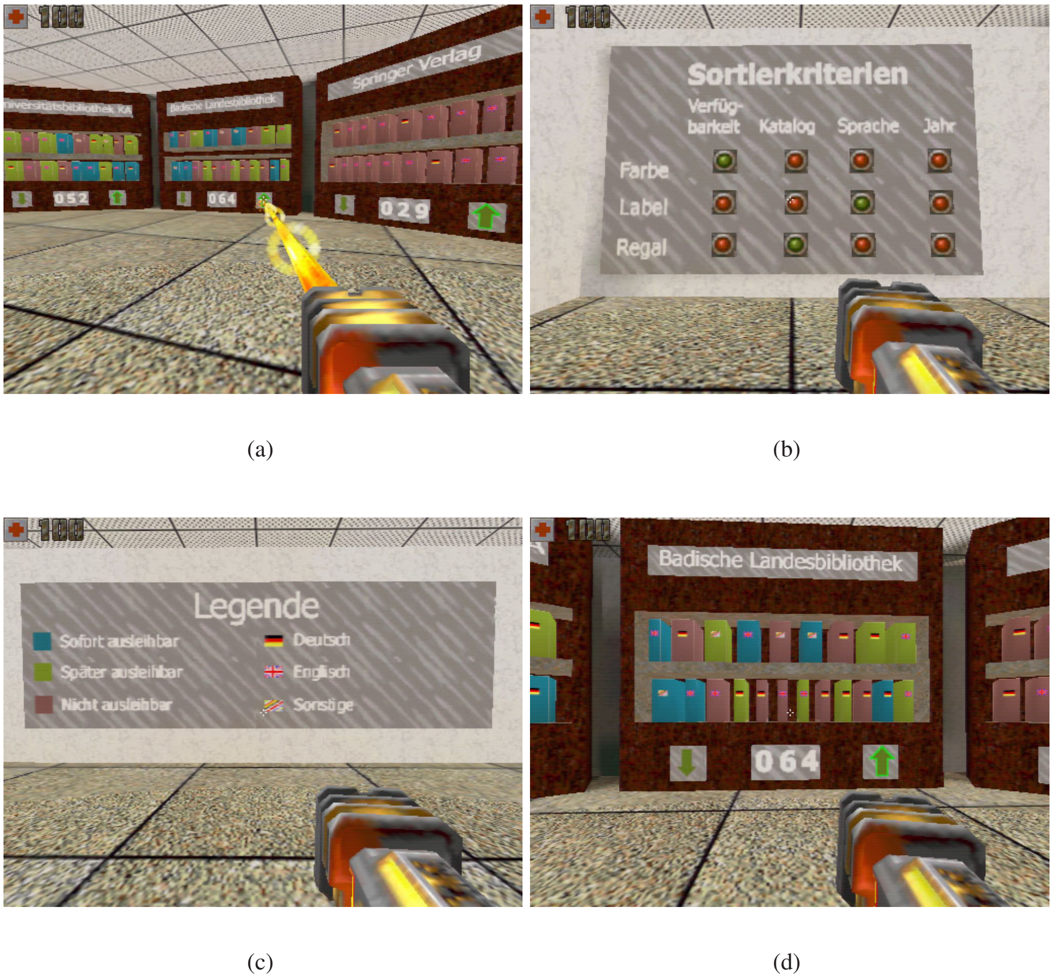
Damit der Benutzer jetzt seine Eingaben über die Tastatur durchführen kann, muß zuerst von der bisherigen Tastaturbelegung (Steuerung der Spielfigur) auf die deutsche Belegung (Buchstaben, Shift, Löschen, ...) umgeschaltet werden. Dies scheint auf direktem Weg leider nur in der game engine möglich, in der die Tastatursteuerzeichen interpretiert werden. Auf dem Umweg über Quake 2-Konsolenbefehle zur Bindung von Tasten an bestimmte Befehle kann dies aber ebenso erreicht werden. Ein weiteres Problem stellt der Quake 2-Zeichensatz dar, der nur den Standard-ASCII-Zeichensatz ohne länderspezifische Sonderzeichen unterstützt. Da Quake 2 die Zeichen aber aus einer Grafikdatei `conchars.pcx` bezieht, ist es möglich, „anscheinend unbenutzte“ Zeichen durch die deutschen Sonderzeichen (ä, ö, ü, Ä, Ö, Ü, ß) sowie einem Ersatzzeichen für andere länderspezifischen Zeichen (wie z.B. œ, å, õ, ò) zu überschreiben (Abb. 4.13). Nun muß die interne Darstellung der Sonderzeichen nur durch den Oktalwert der Stelle ihres Auftretens in der Grafikdatei ersetzt werden.

Da sich an der grafischen Darstellung der Eingabemaske von Frame zu Frame nur sehr wenig ändert, ist es sinnvoll, wenn auch etwas aufwendiger, diese mit Hilfe der Quake 2-Statusbar-Funktionalität zu realisieren. In dieser wird „normalerweise“ ständig der aktuelle

Abbildung 4.13: Die veränderte Datei `conchars.pcx`

Spielerstatus (Gesundheitspunkte, Rüstungspunkte, aktuelle Waffe, restliche Munition, ...) auf dem Bildschirm ausgegeben, so daß hierfür eine effiziente Client-Server-Kommunikation zur Verfügung steht. Nachdem der Benutzer mit Suchen seine Suchanfrage bestätigt, wird, um das restliche Programm nicht zu blockieren, ein Thread zur TCP-Kommunikation gestartet. Dieser wandelt die Suchanfrage in die Übertragungssyntax an den Benutzeragenten um und stellt eine TCP-Verbindung zu diesem her; er übermittelt die Anfrage und empfängt das Ergebnis, welches in eine Bücherstruktur eingelesen und zur Darstellung aufbereitet, inbesondere in verschiedene Listen sortiert wird.

Die Darstellung der Suchergebnisse erfolgt – wie in einer Bibliothek auch nicht anders zu erwarten – über Bücher, die in Regalen stehen (Abb. 4.14 a). Die Bücher können wahlweise nach vier Kriterien sortiert werden: Katalogzugehörigkeit (Universitätsbibliothek Karlsruhe, Badische Landesbibliothek, Springer Verlag), Verfügbarkeit (sofort ausleihbar, mittelfristig ausleihbar, nicht ausleihbar), Sprache (englisch, deutsch, sonstige), Erscheinungsjahr (bis 1989, 1990–1995, ab 1996). Zur grafischen Darstellung dieser Kriterien stehen drei unabhängige Medien zur Verfügung: getrennt stehende Regale, die Buchfarbe sowie das Buchlabel auf dem Buchrücken. Zusätzlich wird die Seitenzahl der Bücher durch ihre Dicke symbolisiert (weniger als 200 Seiten: dünn, 200 bis 600 Seiten: mittel, mehr als 600 Seiten: dick). Hierbei kann der Benutzer über eine Auswahltafel entscheiden, welches Kriterium durch welches Medium dargestellt werden soll (Abb. 4.14 b) (also z.B. Katalogzugehörigkeit durch verschiedene Regale, Verfügbarkeit durch die Buchfarbe und die Sprache durch das Buchlabel).



(a) (b) (c) (d)

Abbildung 4.14: Regale, Auswahltafel und Legenden

Die eigens hierfür definierten Entity-Klassen können wie folgt beschrieben werden (siehe dazu auch Abb. 4.14):

B_Buch: ein Model eines Buches, das in drei verschiedenen Dicken und mit etlichen unterschiedlichen Texturen verfügbar ist

B_Regal: ein Anzeigemodul, dessen Textur die Ausprägung des momentan für die Regaldarstellung gewählten Kriteriums beschreibt (z.B. „sofort ausleihbar“); gleichzeitig dient es noch als zentrales Regal-Entity, welches unter anderem auch die Bücherliste des Regals verwaltet

B_Regal_Boden: die als „Kasten“ (mit Decke, Boden und Seiten) realisierten Zwischenböden des Regals, auf denen ein Teil der Bücher aus der diesem Regal zugeordneten Bücherliste stehen. Dabei bietet das Regal Platz für vier solcher Böden: die zwei mittleren Plätze sind für den Benutzer sichtbar, die zwei äußeren durch das Regalgehäuse (einfache Brushes) verdeckt (dort werden die Bücher auf dem Boden erzeugt bzw. entfernt). Da für den Transport der Bücher nur 3 Böden benötigt werden, sind im Level-Editor nur die untersten drei zu modellieren.

B_Regal_Begrenzung: Point Entities, die sich am obersten und untersten Platz für die Zwischenböden (für den Benutzer nicht sichtbarer Bereich) jeweils am rechten und linken Rand befinden, um den Bereich zu definieren, in dem die Bücher erstellt werden

B_Regal_Anzeige: drei Ziffern, die die Gesamtzahl der auf dem Regal befindlichen Bücher grafisch darstellen

B_Regal_Knopf: zwei Knöpfe, die das Regal nach oben bzw. unten weiterschalten. Beim Betätigen des Knopfes nach oben (für den Knopf nach unten gilt entspechendes) werden die drei Zwischenböden nacheinander um eine „Stufe“ nach oben bewegt. Ist dabei der obere Zwischenboden im oberen nicht sichtbaren Bereich des Regalgehäuses verschwunden, werden dessen Bücher aus dem Level entfernt und anschließend die in der Bücherliste nächsten Bücher auf dem unteren Zwischenboden erzeugt (dieser befindet sich in diesem Moment noch im unteren nicht sichtbaren Bereich des Regalgehäuses). Mit Hilfe dieses Mechanismus wird es möglich, mehrere hundert Bücher in einem Regal unterzubringen, obwohl nur etwa 25 Stück gleichzeitig auf den zwei sichtbaren Zwischenböden zu sehen sind (hierbei begrenzt nicht nur der Platz die Anzahl der sichtbaren Bücher, sondern die Tatsache, daß von Quake 2 anscheinend maximal 128 sichtbare Entities gleichzeitig dargestellt werden können). Die Texturen der Knöpfe stellen Pfeile nach oben bzw. unten dar, je nach Richtung der Regalbewegung, die sie auslösen; sind in einer Richtung noch weitere Bücher in der Bücherliste vorhanden, so leuchtet der jeweilige Pfeil.

B_UniCats_Auswahlknopf: die zwölf Schalter der Auswahltafel für die Zuordnung der Kriterien zu den Medien Regal, Buchfarbe, Buchlabel

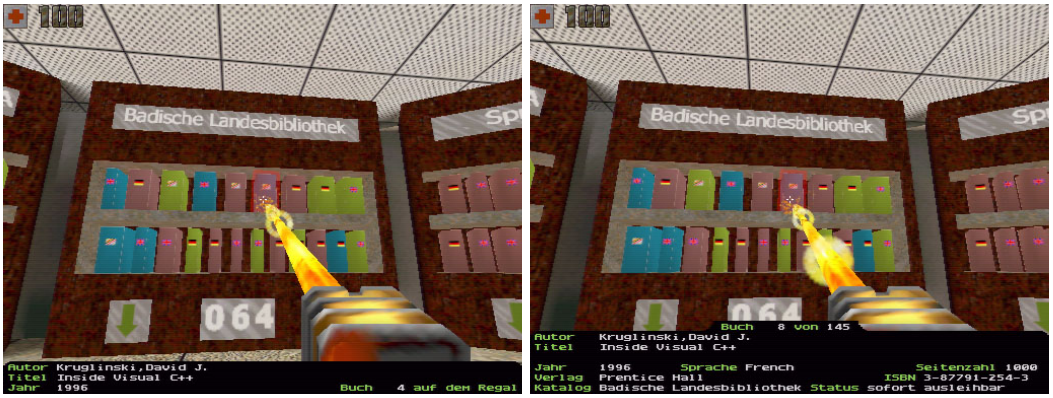
B_UniCats_Legende: zwei Legenden, die dem Benutzer erläutern, durch welche Farbe bzw. welches Label der Bücher die Ausprägungen der jeweils gewählten Kriterien symbolisiert werden

B_UniCats_Computer: ein Model eines Computers, um bei Berührung die Eingabemaske zu aktivieren

B_UniCats_Schleusentuer: spezielle Türen, um eine Art Schleuse zu realisieren. Das ganze UniCats-System wird in einem durch eine Schleuse (diese Türen inklusive areaportal-Entities) abgetrennten Raum realisiert, da die vielen hier auftretenden Entities nahe an die Darstellungsgrenzen von Quake 2 heranreichen und nur mit Hilfe einer Schleuse gewährleistet werden kann, daß diese Grenzen im Betrieb nicht überschritten werden.

B_UniCats_Schleusentrigger: Diese nicht sichtbaren Solid-Entities stellen die Auslöser für die Schleusentüren dar, läuft der Benutzer in deren „virtuellen“ Bereich, werden die Schleusentüren betätigt. Zusätzlich wird hiermit die An- und Abmeldung eines Benutzers an das UniCats-System geregelt (nur der am System angemeldete Benutzer darf das System benutzen und steuern)

B_UniCats: Ein Point Entity, das die zentrale Rolle des kompletten Ein- und Ausgabesystems übernimmt, also die Regale, die Auswahlknöpfe, die Legenden, den Computer, die Schleusen sowie die Bücherlisten verwaltet



(a) (b)

Abbildung 4.15: kurze und ausführliche Buchinformationsanzeige

Um die hier auftretenden Knöpfe zu bedienen und vor allem, um neben den bisher grafisch dargestellten, auch die restlichen Bücherinformationen abzurufen, dient der Laserpointer (eigentlich eine Waffe eines Quake 2-Zusatzpaketes mit weiteren Levels und Entities). „Feuert“ man mit ihm auf die Knöpfe, so schalten diese; „schießt“ man auf ein Buch, so wird dieses ausgewählt und durch rotes Leuchten hervorgehoben. Sollte ein identisches Buch noch einmal im selben oder einem anderen Regal vorhanden sein (z.B. ein Exemplar vom „Springer Verlag“ und eines von der „Badischen Landesbibliothek“ oder eines in deutsch und eines in englisch), so leuchtet dieses grün, sofern es gerade als Buch auf den sichtbaren Zwischenböden steht; falls dem nicht so ist, es also „nur“ in der Bücherliste eines Regals aufgeführt wird, blinkt der entsprechende Pfeil am Regal. Die Daten des hervorgehobenen Buches (Autor, Titel, . . .) können am unteren Bildschirmrand abgelesen werden. Das System bietet dazu zwei Modi an, zwischen denen man hin- und herschalten kann, indem man noch einmal auf ein ausgewähltes Buch (rot leuchtend) „schießt“ (Abb. 4.15):

kurze Buchinformation:

1. Zeile: Autor
2. Zeile: Titel
3. Zeile: Jahr, Buchnummer xxx im jeweiligen Regal

ausführliche Information:

1. Zeile: Buchnummer yyy von insgesamt zzz Büchern
2. Zeile: Autor
3. und 4. Zeile: Titel
5. Zeile: Jahr, Sprache, Seitenzahl
6. Zeile: Verlag, ISBN
7. Zeile: Katalog, Verfügbarkeitsstatus

4.5 Modellierung der Bibliothek: Teil 3

Um die selbst implementierten Entities in Worldcraft nutzen zu können, wird der Entity-Klassenname in den Objekteigenschaften per Hand eingetragen und sukzessive jede benötigte Variable mit ihrem Wert eingefügt (Add-Knopf), überflüssige werden gelöscht (Delete-Knopf) (Abb. 4.10). Soll aber der Smart Edit-Modus mitbenutzt werden, muß die Worldcraft eigene Konfigurationsdatei (Dateiendung .fgd) angepaßt oder eine neue dieser Art erstellt werden. In diesen FGD-Konfigurationsdateien werden alle Entity-Klassen in einer speziellen, leicht zu lernenden Syntax aufgelistet unter Angabe des DLL-internen Variablennamens, des in Worldcraft zu lesenden Smart Edit-Variablennamens und des Wertebereichs dieser Variablen. Bei Bedarf können auch Standardwerte oder ein Pulldown-Menü definiert werden. Nur die in FGD-Dateien definierten Entities kann man direkt in der Objektliste an der Object Bar oder der Class-Liste in den Objekteigenschaften anwählen. Für die Arbeit mit Worldcraft empfiehlt es sich deshalb, seine eigenen Entities ebenfalls in einer FGD-Datei zu definieren und diese in der diesbezüglichen Liste in der allgemeinen Wordcraft-Konfiguration hinzuzufügen.

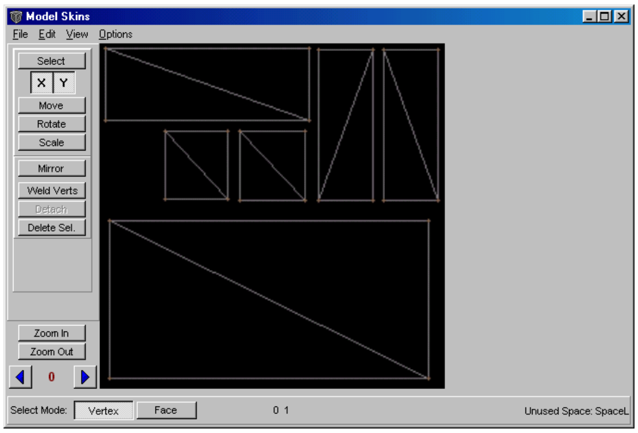
Diese kann über Tools/Options angepaßt werden. Dort werden auf sechs Registerkarten Einstellungen zu den Bereichen General, 2D Views, 3D Views, Textures, Game Configuration und Build Programs vorgenommen. In Game Configuration und Build Programs sind vor allem die Pfade zu den Hilfsprogrammen QBSP3, QRAD3, QVIS3 und zu den Dateien `quake2.fgd` und `quake2.pal` anzupassen, falls Worldcraft nicht im Standardverzeichnis installiert wurde. Desweiteren sind die Pfade zu Quake 2 und den Ausgabeverzeichnissen der RMF-, MAP- und BSP-Dateien sowie zu den Verzeichnisbäumen mit den Texturen einzutragen. Zur korrekten Ausführung von QRAD3 muß auch noch `colormap.pcx` vom Worldcraft-Verzeichnis in das Verzeichnis $x\backslash$pics kopiert werden, wenn das BSP-Ziel-Verzeichnis $x\backslash$maps heißt.

Da schon einige Male von selbsterstellten Models und Geräuschen die Rede war, widmen sich die folgenden zwei Abschnitte dieser Thematik.

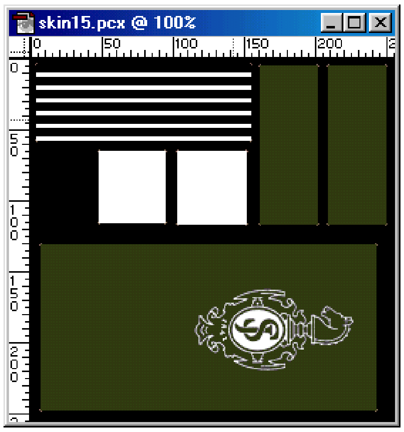
4.6 Konstruktion der Models

Um komplexe detailliertere ständig wiederkehrende Objekte zu modellieren, ist der Aufbau durch einzelne Brushes zu ineffizient, da Brushes nur konvex sein dürfen und die Texturen für jede Oberfläche einzeln definiert werden müssen. Weiterhin existieren nur sehr begrenzte Möglichkeiten, Brushes zu animieren (z.B. die Textur ändern oder den Brush als Ganzes verschieben). Zu diesem Zweck existieren die Models, die in Quake 2 als eine Einheit behandelt werden. Models bestehen aus einem Drahtgittermodell (nicht notwendigerweise konvex) und einer Textur als Oberflächenüberzug – die Skin (eine PCX-Grafikdatei, keine Quake 2-Textur im obigen Sinn). Damit die Quake 2-Engine weiß, wie die Skin über das Model zu legen ist, d.h. welche Ausschnitte der Grafikdatei auf welche Teiloberfläche des Models zu legen ist, wird die 2D-Projektion aller Teiloberflächen des Models gespeichert – der sogenannte Mesh. Dazu werden die Projektionen als 2D-Drahtgitter auf einer mit der Skingöße übereinstimmenden Fläche nicht überlappend angeordnet. Desweiteren kann der Detaillierungsgrad verschiedener Teiloberflächen des Models individuell gesteuert werden, indem deren 2D-Drahtgitter auf dem Mesh unterschiedlich skaliert werden (Abb. 4.16) (z.B. benutzten wir etwa die Hälfte des ganzen Meshs eines Buches für den Buchrücken, obwohl dieser geometrisch gesehen sicherlich nicht die Hälfte der gesamten Buchoberfläche ausmacht). Ein Model kann animiert werden, indem mehrere „Schnappschüsse“ des Drahtgittermodells als sogenannte Frames des Models gespeichert werden (bis zu 512 Frames je Model). Ebenso ist man nicht an nur eine Skin gebunden, sondern kann bis zu 32 Skins für ein Model angeben, jedoch besitzt jedes Model, egal wieviele Frames oder Skins es hat, nur einen Mesh. Welcher Frame bzw. welche Skin des Models im Moment ausgewählt ist, wird in der game dll festgelegt.

Die Konstruktion eines Models beginnt damit, das Drahtgittermodel des gewünschten Objektes zu modellieren. Dazu verwendeten wir das 3D-Modellierungswerkzeug 3D Studio Max (Abb. 4.17). Aufgrund der Komplexität und der Fülle an Möglichkeiten, die dieses Programm bietet, um 3D-Objekte zu erzeugen und zu manipulieren, kann hier unsere Vorgehensweise nur sehr grob erläutert werden. Es sei an dieser Stelle nur soviel gesagt: Auf der Befehlspalette/Erzeugen wählt man einen Grundkörper aus (z.B. Box) und legt durch diverse Mausklicks in den 2D-Ansichten dessen Dimensionen und Position fest. Anschließend kann das Objekt beliebig manipuliert werden (z.B. durch Verschieben, Skalieren, Drehen etc.). So wird das gewünschte Drahtgittermodell Schritt für Schritt aus einzelnen Grundkörpern aufgebaut und am Ende mit File/Export im 3D-Studio-Format (Dateiendung `.3ds`) abgespeichert (hat man mehrere Frames, also ein animiertes Drahtgittermodell, so werden diese einzeln gespeichert).



(a)



(b)

Abbildung 4.16: Buch-Mesh im Quake 2 Model Editor und Buch-Skin in Photoshop

Diese Datei kann man nun im Quake 2 Model Editor über File/Import/Import Append Verts... zur weiteren Bearbeitung importieren (die weiteren Frames werden mit File/Import/Import Append Frames... als neue Frames angehängt). Dort wird nun der Mesh erzeugt, was ein wenig trickreich ist und einiger Übung bedarf, bis das gewünschte Ergebnis erzielt werden kann. Damit bei der anschließenden Projektion des 3D-Drahtgittermodells auf ein 2D-Gitternetz nicht einige Oberflächen zu Linien verkümmern, sollte man zuerst das Model im Raum (in den 2D-Ansichten) schräg stellen; wenn man jede Oberfläche einzeln im Mesh übernehmen möchte, ist dies aber nicht nötig. Je nach Komplexität des Models werden die folgenden Schritte für die einzelnen Teilobjekte nacheinander durchgeführt. Dazu wechselt man mit Hilfe des entsprechenden Knopfes auf der Statusleiste am unteren Fensterrand in den Face-Modus (im Vertex-Modus werden immer nur einzelne Eckpunkte des Drahtgittermodells ausgewählt, im Face-Modus dagegen immer ein Dreieck aus 3 Punkten; somit besteht z.B. eine Vierecksfläche aus zwei Dreiecken). Dort markiert man ein Teilobjekt, indem man alle Dreiecke der Teilobjektoberflächen nacheinander durch einen Mausklick auswählt. Anschließend wechselt man mit View/Skins... in das Skin-Fenster, wo man ebenfalls den Face-Modus aktiviert, falls dieser noch nicht ausgewählt ist. Nun kann man mit Edit/Get Position from Model... die Oberflächen des ausgewählten Teilobjektes auf eine (zu Beginn noch schwarze) Skin projizieren lassen; man hat hier die Auswahl über die Projektionsrichtung: von vorne, hinten, links, rechts, unten oder oben. Steht das Model schräg im Raum, so genügt es, jeweils zwei korrespondierende Richtungen (also z.B. erst unten und im nächsten Schritt oben) zu wählen, um alle Oberflächen zu beinhalten, ansonsten muß diese Prozedur unter Umständen von allen Richtungen aus durchgeführt werden. Nach jeder dieser Rich-

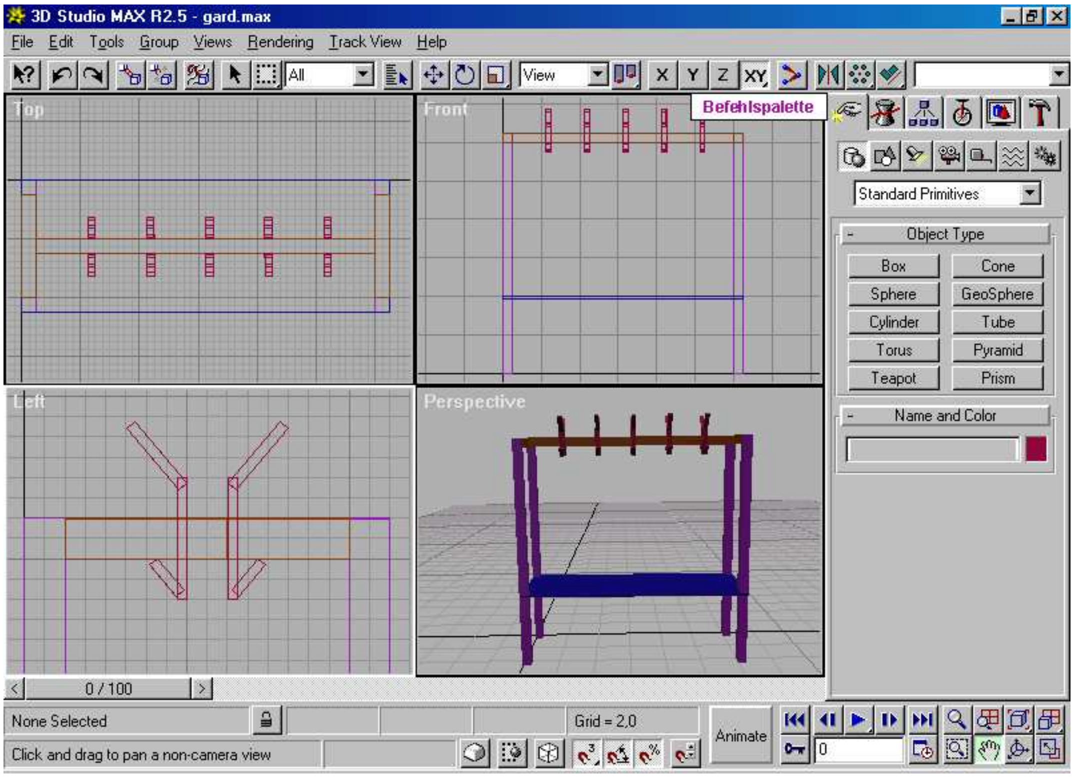


Abbildung 4.17: Kleiderständer in 3D Studio Max

tungen wird das Teil-Gitternetz auf der Skin entsprechend manipuliert und plaziert (mit den Werkzeugknöpfen Move, Rotate, Scale etc. am linken Fensterrand), dabei kann es z.B. auch sinnvoll sein, im Vertex-Modus einzelne Ecken des Teil-Gitternetzes zu verschieben, um eine sinnvolle Fläche zu erhalten. Hat man auf diese Weise nun den Mesh aller Oberflächen dieses Teilobjektes erzeugt, so verfährt man mit dem nächsten Teilobjekt entsprechend, bis alle Oberflächen des Models auf dem Mesh abgebildet sind (Abb. 4.16 a). Ist der Mesh soweit angeordnet, kann man (immer noch im Skin-Fenster) mit File/Export/Current Image to PCX... das Gitternetz in einer PCX-Datei speichern. Diese PCX-Datei lädt man in ein Grafikprogramm (z.B. Photoshop) und malt entsprechend der Aufteilung durch die Gitternetzlinien die Textur der Oberflächen aus (dabei spielen die nicht vom Gitternetz eingeschlossenen Teile keine Rolle, so daß man die Übergänge nicht exakt nachzeichnen muß) (Abb. 4.16 b). Ist die Skin fertig (und in einer Datei gespeichert), kann man im Quake 2 Model Editor (wieder im Skin-Fenster) mit File/Import/Skin Image... die entstandene PCX-Datei als Skin für dieses Model importieren. Möchte man noch weitere Skins (die auf die gleiche Art und Weise gemalt werden) importieren, so muß man zuerst über Edit/Add New Skin... eine neue Skin anhängen bzw. irgendwo in die Reihe der bestehenden Skins einordnen (übrigens kann man mit den blauen Pfeiltasten zwischen den verschiedenen Skins wechseln; mit den entsprechenden Pfeiltasten im Model-Fenster wird zwischen den Frames gewechselt).

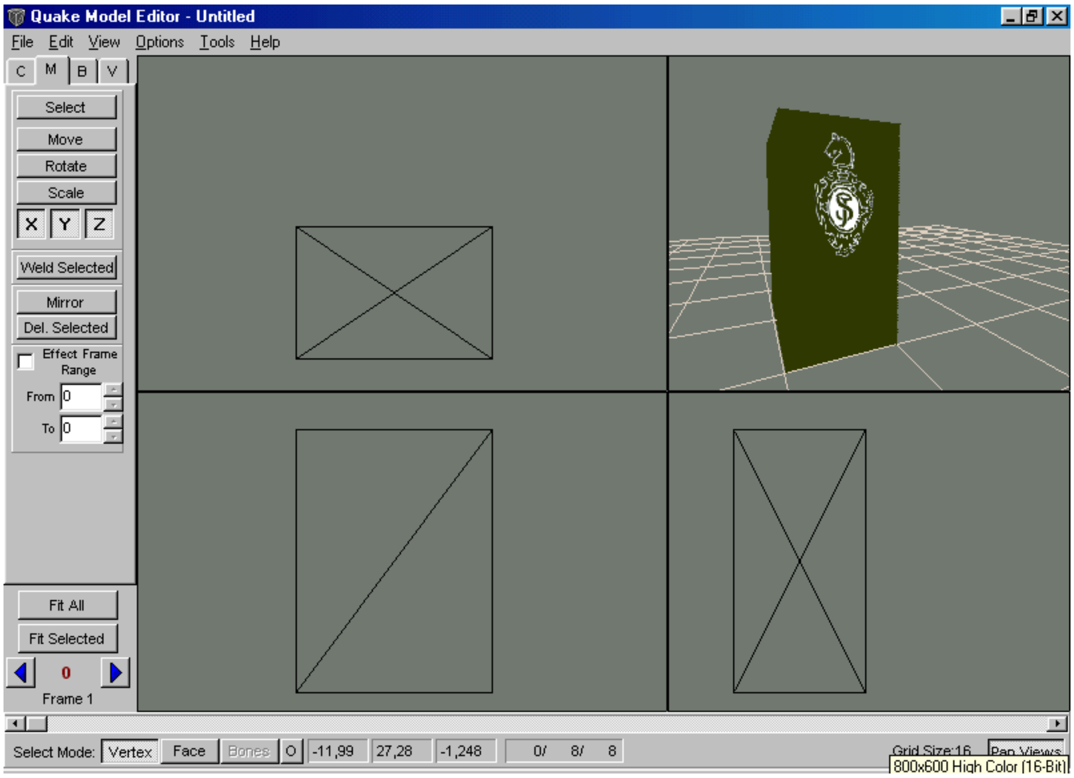


Abbildung 4.18: Buch-Model im Quake 2 Model Editor

Ist man nun mit seinem Werk soweit zufrieden (Abb. 4.18), speichert man das Model im Quake 2-Model-Format ab, indem man im Model-Fenster File/Export/Export to MD2... anwählt (nachdem man den Dateinamen des Models gewählt und bestätigt hat, wird man leider genötigt, für jede Skin nocheinmal die entsprechende PCX-Datei im Dateisystem auszuwählen).

Abschließend sei noch bemerkt, daß man die ganze Arbeit selbstverständlich auch komplett mit dem Quake 2 Model Editor erledigen kann, jedoch scheint es mir (zumindest in der von uns verwendeten Version) dort erheblich komplizierter und umständlicher, ein entsprechendes Drahtgittermodell zu erstellen. Nach getaner Arbeit entdeckten wir außerdem, daß ein Plugin für 3D Studio Max existiert, mit dem man direkt im Quake 2-Model-Format (Dateiendung `.md2`) abspeichern kann. Somit kann man auch ganz auf die Verwendung des Quake 2 Model Editors verzichten, was einem die Arbeit wahrscheinlich um einiges erleichtert (dazu müßte man sich nur noch aneignen, wie man in 3D Studio Max die Objekte texturiert ..., aber zum Thema 3D Studio Max ist ja reichlich Lesestoff erhältlich ...).

4.7 Digitalisierung der Geräusche

Um in einer 3D-Welt dem Benutzer einen realistischen Eindruck zu vermitteln, sind neben dem Aussehen der Umgebung auch Geräusche von Bedeutung. Dazu sind in Quake 2 schon eine Vielzahl an Geräuschen vorhanden (z.B. für Schritte, Türen, Waffen etc.), so daß man sich nur um außergewöhnliche oder individuelle Geräusche kümmern muß. Bei real existierenden Räumlichkeiten sind dies z.B. Hintergrundgeräusche sowie Geräusche von speziellen Objekten (z.B. eines Fahrstuhls, eines Computers, ...). Um diese in die 3D-Welt von Quake 2 integrieren zu können, müssen sie in digitaler Form im WAV-Format vorliegen. Die benötigten Geräusche werden mit einem Mikrofon auf Band aufgenommen und anschließend über den Line-In Eingang einer Soundkarte mit Hilfe geeigneter Software (meist bei Soundkarte mitgeliefert) digitalisiert und im WAV-Format gespeichert bzw. später in dieses umgewandelt.

4.8 Modellierung der Bibliothek: Teil 4

Um nun die eigenen Models und Geräusche in einen Level einzubauen, gibt es drei grundlegende Möglichkeiten. Es existieren Entities, bei denen man im Level-Editor die Möglichkeit hat, den Pfad einer Geräusch- oder Model-Datei anzugeben. Damit wäre das Problem in einfachster Weise gelöst; dazu muß aber erst einmal das richtige Entity existieren, was vor allem für Models i.d.R. nicht gegeben ist, so daß dann doch eine der beiden folgenden Möglichkeiten Anwendung findet. Die schnelle, aber „unsaubere“ Methode besteht darin, schon vorhandene Models und Geräusche durch die eigenen zu ersetzen. Hierzu muß man die ursprünglichen Dateien nicht löschen, sondern kann seine eigenen Dateien mit gleichem Namen und fast gleichem Verzeichnispfad speichern. Dabei gilt aber ebenfalls, daß man ein geeignetes Entity finden muß, dessen Geräusch oder Model man ersetzen kann (zusätzlich muß man noch wissen, welche Geräusch- bzw. Modeldatei dieses Entity überhaupt benutzt). Zum Ersetzen einer Datei erst einmal ein kleiner Exkurs zur Dateiablage in Quake 2:

Um den Speicherverschnitt bei den vielen u.U. kleinen Dateien (Texturen, Geräusche, Grafiken etc.) möglichst klein zu halten und auch des schnelleren Zugriffs wegen werden viele Dateien des (Original-)Spiels in einer einzigen PAK-Datei zusammengefaßt, der Datei `baseq2/pak0.pak`. In PAK-Dateien exisiert eine „normale“ Dateistruktur, auf welche man mit speziellen Programmen auch zugreifen kann, so z.B. mit dem Freeware Programm *PakExplorer*. Wird Quake 2 gestartet, so wird die resultierende Dateistruktur für Dateizugriffe folgendermaßen zusammengestellt: Es wird versucht, nacheinander die Dateien `baseq2/pak0.pak`, `baseq2/pak1.pak`, ..., `baseq2/pak9.pak` zu durchsuchen, falls die jeweilige Datei überhaupt vorhanden ist. Sollte dabei in einer nachfolgenden PAK-Datei in der dort gespeicherten Dateistruktur eine Datei vorhanden sein, die mit gleichem Namen und gleichem Pfad schon in einer vorhergehenden PAK-Datei existiert, so wird die schon vorhandene „virtuell überschrieben“, d.h. für Quake 2 ist nur die zuletzt auftretende Datei diesen

Namens und Pfades bekannt. Nach dem Durchsuchen der PAK-Dateien wird noch die ab dem Verzeichnis `baseq2` real vorhandene Dateistruktur durchsucht. Sollte die Konsolenvariable game ein anderes Spielverzeichnis definieren (bei unserer Erweiterung also z.B. unicats), wird anschließend gleichermaßen mit diesem Verzeichnis verfahren, also wiederum die PAK-Dateien nacheinander durchsucht und zum Schluß noch die Verzeichnisstrukur selbst (bei uns also `unicats/pak0.pak`, ..., `unicats/pak9.pak` und `unicats/*`). Somit kann man also sehr leicht schon vorhandene Geräusche oder Models ersetzen (dies gilt ebenso für die `gamex86.dll`, wobei diese nicht in einer PAK-Datei liegen darf, da sie vom Betriebssystem geladen werden muß; unsere `gamex86.dll` liegt im Verzeichnis `unicats` und findet dann Verwendung, wenn die Konsolenvariable game auf unicats gestellt wird). Dank dieses Mechanismus kann man Quake 2 einfach und ohne großes „Datei-Wirrwarr“ erweitern. Die etwas aufwendigere Methode, seine eigenen Models und Geräusche einzubauen, besteht darin, den game dll-Quelltext entsprechend zu erweitern. Diese Vorgehensweise bedeutet zwar mehr Arbeit, jedoch kann man somit ein genau auf das Problem zugeschnittes Entity erstellen, welches sich am Ende auch wie gewünscht verhält. Bei animierten Models kommt man i.d.R. ohne diese Methode nicht aus, da z.B. jede Waffe eine unterschiedliche Anzahl an Frames und eine speziell auf das Model ausgerichtete Animation hat.

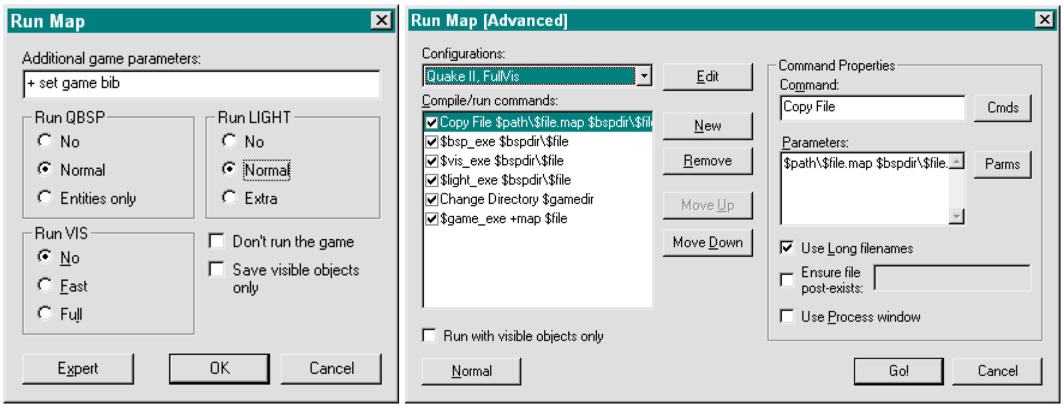
Sind nun alle Brushes und Entities mitsamt ihren Models und Geräuschen in den Level eingefügt, so steht der Erzeugung der BSP-Datei nichts mehr im Weg. Über File/Run öffnet man das Dialogfenster Run Map (Abb. 4.19 a). Hier werden die Einstellungen für die Hilfsprogramme vorgenommen:

BSP-Programm Erzeugung der vom Spiel benutzten Level-Datei; dabei wird die interne Struktur der 3D-Welt erzeugt, der BSP-Baum.

VIS-Programm Hier wird ermittelt, welche Bereiche der 3D-Welt von welchen Bereichen aus nicht sichtbar sind, dadurch bleibt der Engine im Spiel einige Arbeit erspart, da eventuell ganze Bereiche erst gar nicht berechnet werden müssen.

RAD-Programm Die statischen Lichteffekte werden berechnet.

Im folgenden werden in Klammern auch Angaben zur Rechendauer der einzelnen Programme gemacht. Diese Angaben beziehen sich auf unser fertiges Bibliotheksmodell; die von uns dafür benutzte Hardware war ein Celeron 300A MHz mit 128 MB RAM. Im Standarddialogfenster (Abb. 4.19 a) können die unter Tools/Options/Build Programs eingetragenen Hilfsprogramme folgendermaßen benutzt werden (die folgenden Zeitangaben beziehen sich auf die Originalprogramme von id Software, QBSP3, QVIS3 und QRAD3): Run QBSP wird i.d.R. mit der Option Normal gestartet (22 s). Wurden seit der letzten Kompilierung des Levels nur Entities verändert, so kann der Prozeß mit Entities only beschleunigt werden (1 s). Run VIS ist während der Entwicklungsphase des Levels nicht unbedingt nötig, also normalerweise Option No oder die Option Fast (101 s). Hingegen empfiehlt es sich ganz am Ende des Level-Design-Prozesses, die Option Full zu verwenden. Run LIGHT sollte, solange noch keine Lichter vorhanden sind, nicht angewandt werden (Option No), da der Level sonst



(a) Standard (b) Expert

Abbildung 4.19: Dialogfenster zur Level-Kompilierung

komplett dunkel ist. Um die Lichtwirkung zu testen, sollte man Run LIGHT vom Experten-Modus aus mit den beschleunigenden Optionen -bounce 2 (das Licht wird maximal an zwei Oberflächen reflektiert) und -chop 128 (der Level wird zur Lichtberechnung in 128x128 u große Stücke unterteilt) starten (26 min 13 s). Am Ende kann dann die Option Normal verwendet werden oder auch Extra, um eine „weichere“ Wirkung der Lichteffekte zu erhalten (dauert aber ungefähr viermal so lang wie Normal). Möchte man individuellere Einstellungen vornehmen, d.h. noch zusätzliche Optionen an die Hilfsprogramme übergeben, so verwendet man den Experten-Modus (Abb. 4.19 b) oder startet die Hilfsprogramme gleich in der Kommandozeile (dies ist vor allem beim Aufruf des VIS-Programms der Fortschrittsanzeige wegen zu empfehlen). Neben den Originalprogrammen existieren (da zu diesen der Quelltext auch frei verfügbar ist) eine Reihe an modifizierten und teilweise schnelleren Versionen. Wir benutzten daher eine Variante des VIS-Programmes, welche mit der Option -fast nur 86 s benötigte und ohne weitere Optionen (entspricht Full im Standarddialogfenster) ungefähr 9 h 50 min. Weiterhin benutzten wir eine Variante des RAD-Programms, ArghRad, mit den Zeiten 23 min 33 s (unter den Optionen -bounce 2 -chop 128) bzw. etwa 2 h (ohne weitere Optionen, entspricht Normal im Standarddialogfenster).

Um die Laufzeit von VIS-Programmen zu verkürzen, können alle Brushes, die Details des Levels darstellen, d.h. die nicht oder nur sehr begrenzten Einfluß auf die Sicht des Spielers haben, als solche markiert werden, indem das Flag detail seiner Texturen gesetzt wird (Abb. 4.8 auf S. 39) (z.B. könnte man in unserem Level die Wendeltreppe oder die Tische als Details markieren, jedoch Decken und Wände auf keinen Fall; mit den Bücherregalen und Schließfächern sollte man deren Einfluß vielleicht erst einmal testen). Wie oben bereits erwähnt, verbessert ein VIS-Programm i.d.R. die Performanz im Spiel, da die Engine weniger zu berechnen hat. Dies kann man noch mit Hilfe weiterer Mittel zusätzlich unterstützen. Da wären zum einen die von uns auch benutzten Areaportals und zum anderen spezielle Brushes mit gesetztem hint-Flag. Von VIS-Programmen werden Entities überhaupt nicht berücksichtigt, da sie im Spiel (zumindest theoretisch) beweglich sind und deshalb keine verläßlichen Aussagen über die Sichtverhältnisse im Spiel gemacht werden können. Da nun aber Türen i.d.R. geschlossen sind und sonst voneinander getrennte Bereiche des Levels verbinden, wäre es sinnvoll, wenn die Engine bei geschlossener Tür auch weiß, daß sie den Raum hinter der Tür nicht zu berücksichtigen braucht, da er sowieso nicht gesehen werden kann. Hier kommt das Entity areaportal ins Spiel: Ein Areaportal läßt im geschlossenen Zustand weder Licht noch Geräusche passieren, verhält sich also wie eine Wand; im offenen Zustand verhält es sich jedoch so, als wäre es überhaupt nicht vorhanden. I.d.R. wird ein Areaportal in der Mitte einer Tür mit dem „Türrahmen überlappend" plaziert und durch entsprechende Name- und Target-Einträge von der Tür beim Öffnen und Schließen automatisch gesteuert. Dagegen zeigen hint-Brushes keinen direkten Einfluß im Spiel, da sie in der BSP-Datei gar nicht mehr vorhanden sind. Sie geben dem BSP-Programm lediglich Hinweise darauf, wie es die 3D-Welt (in den obersten Ebenen des BSP-Baumes) in verschiedene Bereiche aufteilen soll. Werden hint-Brushes nun an sinnvollen Stellen plaziert, so findet das VIS-Programm schon in den obersten Ebenen des BSP-Baums Knoten, die voneinander nicht sichtbare Bereiche trennen, so daß die Engine später u.U. recht große Teilbäume des BSP-Baums als vom momentanen Punkt aus nicht sichtbar „überspringen" kann.

Zur näheren Erläuterung von Worldcraft und allen anderen Aspekten zum Thema Level-Design sei nochmals auf diesbetreffende Internetseiten und dort zu findende Tutorials verwiesen (siehe Anhang D).

5 Zusammenfassung, Bewertung, Ausblick

5.1 Zusammenfassung

Die gesteckten Ziele (siehe Kapitel 1.1) wurden erreicht: Nach einer Einarbeitung in Quake 2 und diverse Hilfsprogramme erstellten wir einen Quake 2-Level, der eine Modellierung (eines Teils) der Universitätsbibliothek Karlsruhe darstellt. Der Benutzer kann sich in diesen Räumlichkeiten frei bewegen und mit seiner Umwelt interagieren. Um ein hohes Maß an Realismus zu erreichen, orientierten wir uns bei der Gestaltung des Levels am Grundriß der Bibliothek, außerdem verwendeten wir eigene, auf Fotos basierende Texturen und erstellten eigene interaktive Objekte wie Fahrstühle und Drehtüren. Letzters gelang durch Änderungen und Erweiterungen der Quake 2-Spielelogik, die im C-Quellcode erhältlich ist. Dadurch war es auch möglich, in den Level eine Benutzerschnittstelle für das UniCats-Projekt einzubetten, die aus zwei Teilen besteht: Die Anfrageschnittstelle nimmt eine Anfrage des Benutzers entgegen und sendet sie über TCP/IP an eine externe Komponente, die die Aufgaben des Benutzeragenten übernimmt, die von unserer Arbeit nicht abgedeckt werden, wie z.B. die Kommunikation mit Tradern und Wrappern. Diese externe Komponente sendet dann das Suchergebnis zurück, und der zweite Teil unserer Benutzerschnittstelle, die Ergebnispräsentation, sorgt für eine dreidimensionale Darstellung der gefundenen Bücher. Für die Visualisierung von buchspezifischen Informationen stehen bei uns verschiedene Möglichkeiten zur Verfügung, unter anderem die Verwendung von verschiedenen Farben für den Bucheinband oder die Plazierung verschiedener Labels auf den Buchrücken. Welche Informationen durch die verschiedenen Visualisierungsmittel dargestellt werden sollen, kann der Benutzer innerhalb gewisser Grenzen dynamisch festlegen. Weiterhin besteht auch die Möglichkeit, sich alle relevanten Informationen eines Buches textuell anzeigen zu lassen.

Um die bei der Arbeit mit Quake 2 gewonnen Erfahrungen festzuhalten, haben wir die einzelnen Arbeitsschritte dokumentiert. In Kapitel 4 wird auf Texturen, Models und den Level eingegangen, Anhang B.3 beinhaltet eine umfassende Dokumentation des Quelltextes für die Spielelogik. Wir hoffen, damit den Einstieg für Personen zu erleichtern, die ein ähnliches Projekt verwirklichen oder aber unseres weiterentwickeln wollen.

5.2 Bewertung

5.2.1 Eignung

Es hat sich gezeigt, daß Quake 2 zur 3D-Modellierung relativ gut geeignet ist. Die 3D-Engine ist leistungsfähig und die Navigation einfach und praxiserprobt. Allerdings ist man gewissen Einschränkungen unterworfen, unter anderem deshalb, weil Quake 2 ja eigentlich ein Computerspiel ist und kein universelles Modellierungswerkzeug. Dementsprechend ist es gut geeignet, um Räume etc. darzustellen und weniger geeignet für abstrakte Szenen, die z.B. aus Kegelbäumen bestehen (siehe Abbildung 2.8). Die darzustellenden Räume sollten außerdem nicht allzu groß sein, da sonst zu viele Objekte gleichzeitig sichtbar sind und die Engine dies nicht unbedingt verkraftet. Unsere Modellierung der Universitätsbibliothek stellt in dieser Hinsicht schon fast das maximal Machbare dar. Einer detailgetreuen realistischen Abbildung stehen außerdem die in Kapitel 3.3 genannten Einschränkungen (keine gekrümmten Flächen, keine Spiegelungen, keine beweglichen transparenten Objekte, ...) im Weg. Trotzdem läßt sich mit den gegebenen Mitteln einiges erreichen.

Das Einbringen eigener Funktionalitäten erfolgt problemlos durch Programmierung in C. Bezüglich der Informationsverarbeitung gibt es also praktisch keine Grenzen. Das einzige Problem ist die Informationsdarstellung, hier ist man auf die Möglichkeiten angewiesen, die die Engine bietet. Die (zweidimensionale) Darstellung von Text wird zum Beispiel nur sehr eingeschränkt unterstützt, und auch die Zahl an gleichzeitig darstellbaren dreidimensionalen Objekten ist begrenzt. Einen weiteren limitierenden Faktor stellt der zeitliche Aufwand dar, auf den im folgenden Abschnitt genauer eingegangen wird.

5.2.2 Zeitlicher Aufwand

In diesem Abschnitt soll eine Übersicht gegeben werden, wie aufwendig es war, die Universitätsbibliothek zu modellieren. Dabei unterscheiden wir zwischen der Einarbeitungszeit und der Entwicklungszeit. Die angebenen Zeitdauern sind keine exakten Werte, sondern stellen mehr oder weniger grobe Schätzungen dar.

Für die Einarbeitung in den Level-Editor Worldcraft benötigten wir ca. 10 Stunden. Hier muß man nicht nur den Umgang mit den einzelnen Werkzeugen, sonderen auch die Quake 2-Architektur verstehen lernen, insbesondere was Brushes, Entities und das Kompilieren der Level betrifft. Die eigentliche Konstruktion der Bibliothek dauerte ungefähr 250 Stunden. Hierbei erwiesen sich besonders die detaillierten Objekte als zeitaufwendig, wie z.B. die Wendeltreppen oder die geschwungene Glas-Abtrennung.

Das Erstellen von Texturen ist weniger komplex und erfordert deshalb eine geringere Einarbeitungszeit. Man benötigt etwa 5 Stunden, um den nötigen Umgang mit Photoshop zu erlernen. Da wir das Programm Wally fast nur zum Konvertieren von PCX-Bildern in das WAL-Format genutzt haben, fiel hier die Einarbeitungszeit entsprechend geringer aus. Mit ca. einer Stunde pro Textur muß man dann für die Erstellung rechnen. Bei einfachen

Texturen liegt man natürlich unter diesem Wert, dafür sind andere umso zeitaufwendiger, wie in unserem Fall die Textur der Legendentafel. Außerdem sehen die Texturen leider in Photoshop bzw. Wally oft ganz anders aus als unter Quake 2, sowohl was die Farbtöne als auch die Helligkeit betrifft. Da hilft meist nur langwieriges Ausprobieren nach dem „Trial and Error"-Prinzip.

Die Konstruktion von Models ist jedoch ungleich komplexer. Allein für die Einarbeitung in 3D Studio MAX benötigt man ca. 20 Stunden nur um die wichtigsten Grundlagen zu verstehen. Das Erstellen eines einfachen Models schlägt dann mit ein bis drei Stunden zu Buche. Für komplexe Models mit Animation (auf die wir verzichtet haben) benötigt man sicherlich ein oder zwei Tage.

Den höchsten Schwierigkeitsgrad weist allerdings die Erweiterung des DLL-Quelltextes auf. Man benötigt ca. 50 bis 70 Stunden für die Einarbeitung, was vor allem daran liegt, daß keine offizielle Dokumentation vorliegt. Man findet im Internet zwar vereinzelte Erklärungen, diese sind aber alle unvollständig und enthalten teilweise auch nur Vermutungen und Halbwahrheiten. Für die von uns geschaffenen, teilweise recht umfangreichen Erweiterungen waren ungefähr 400 Stunden Arbeitszeit notwendig, also der Großteil der insgesamt für das Projekt benötigten Zeit.

Tabelle 5.1 faßt die hier genannten Zahlen zusammen.

	Komplexität	*Einarbeitung*	*Entwicklung*
Level	mittel	10 h	250 h
Texturen	gering – mittel	5 h	1 h pro Textur
Models	mittel – hoch	20 h	1 – 3 h pro Model
Spielelogik	hoch	50 – 70 h	400 h

Tabelle 5.1: Aufwand für die 3D-Modellierung mit Quake 2

5.3 Ausblick

Nachdem nun eine einsatzbereite Benutzerschnittstelle entstanden ist, erscheint es sinnvoll, diese auch zu testen, d.h. es sollten Benutzerstudien bezüglich der Bedienungsfreundlichkeit durchgeführt werden. Anhand von Testpersonen kann so überpfüft werden, ob unsere Benutzerschnittstelle den selbstgestellten Anforderungen (siehe Kapitel 2.1) genügt. Auch ein Vergleich wäre denkbar, indem man z.B. bei der Evaluation noch ein textuelles bzw. ein auf VRML basierendes System mit einbezieht.

Falls man sich nach den Benutzerstudien entschließt, unser System weiterzuentwickeln, so bieten sich hierfür mehrere Ansatzpunkte. Zum einen ist eine Erweiterung der Funktionalität denkbar, so daß ein Benutzer nicht nur nach Büchern suchen, sondern diese auch gleich ausleihen oder vormerken kann. Auch die Recherche selbst ließe sich verbessern, z.B.

indem man es dem Benutzer gestattet, die Grenzen für die Einteilung der Bücher nach Erscheinungsjahr selbst zu wählen, denn während im Informatik-Bereich ein Buch von 1990 heutzutage mit großer Wahrscheinlichkeit schon veraltet ist, kann in anderen Gebieten ein Buch dieses Alters noch als hochaktuell gelten.

Auf der anderen Seite könnte man versuchen, die 3D-Modellierung der Universitätsbibliothek noch realistischer zu gestalten, damit die Attraktivität für die Benutzer wächst. Um die Bibliothek mit Leben zu füllen, wäre es denkbar, Models zu erschaffen, die z.B. Studenten oder Bibliothekare darstellen, und in den Level einzusetzen. Auch eine passende Geräuschkulisse würde den Realismus erhöhen. Mit einer anderen Engine, z.B. der des Spiels Unreal, wäre es möglich, auch Objekte wie Glastüren und spiegelnde Oberflächen in den Level einzubauen. Die nächste Stufe bildet dann die Engine von Quake 3: Arena, das noch in der ersten Hälfte dieses Jahres erscheinen soll. Sie arbeitet mit 24 Bit Farbtiefe und bietet unter anderem gekrümmte Oberflächen, Spiegel und volumetrischen Nebel, womit noch größerer Realismus möglich wäre. Es läßt sich jedoch zum jetzigen Zeitpunkt noch nicht sagen, ob auch mit Quake 3: Arena derart umfangreiche Funktionalitätserweiterungen möglich sein werden, wie wir sie für dieses Projekt realisiert haben.

An dieser Stelle sei noch angemerkt, daß es bei id Software üblich ist, nach einer gewissen Zeit den kompletten Quellcode für ihre Spiele zu veröffentlichen. So wurde z.B. der Quellcode für Doom im Dezember 1997, also rund vier Jahre nach dessen Erscheinen, der Allgemeinheit zur Verfügung gestellt. Wenn man für Quake 2 von einem ähnlichen Zeitraum ausgeht, so ergibt sich Ende 2001 als Termin. Durch den Zugriff auf den Quellcode der Engine ließen sich sicherlich viele der genannten Einschränkungen beseitigen und neue Funktionalitäten realisieren. Auf der anderen Seite ist allerdings fraglich, ob dies im Jahr 2001 überhaupt noch aktuell ist, denn niemand kann vorhersagen, mit welchen Arten von Benutzerschnittstellen dann gearbeitet werden wird.

Anhang

A Glossar

Brush: Brushes sind konvexe Polyeder und stellen die elementaren Bausteine eines Levels dar. Obwohl es in einigen Editoren Grundtypen gibt (Quader, Pyramide, etc.), ist ihre Form nicht auf diese beschränkt. Den einzelnen Seiten eines Brushes können unterschiedliche Texturen zugewiesen werden.

BSP: In diesem Format müssen Level vorliegen, damit sie von Quake gelesen werden können. BSP-Dateien entstehen aus MAP-Dateien durch Kompilieren mit QBSP3.

Engine: Die Engine eines 3D-Spiels stellt sozusagen das Herzstück des Spiels dar. Sie ist zuständig für Grafikdarstellung, Soundwiedergabe und Netzwerkfunktionalität und ist damit weitgehend unabhängig vom eigentlichen Spielinhalt. id Software lizensiert seine Engines an andere Software-Firmen, die dann darauf aufbauend eigene Spiele entwickeln. So basieren z.B. die Spiele Hexen2, Sin und Half-Life alle auf der Quake- bzw. Quake 2-Engine.

Entity: Entities sind die aktiven Objekte in einem Level, z.B. Monster, Türen, Teleporter usw. Es gibt verschiedene Typen von Entities, siehe dazu Kapitel 3.2.4.

Konsole: Durch Drücken der ^-Taste (~ auf einer amerikanischen Tastatur) während des Spiels erhält man eine Konsole, mit der man Variablen verändern und Befehle ausführen kann. `map base1` bewirkt zum Beispiel, daß der Level base1.bsp gestartet wird.

Level: Die virtuelle Welt, die man bei 3D-Spielen wie Quake 2 durchwandern muß, ist in logische Abschnitte aufgeteilt, die Levels oder Maps genannt werden. Ein Level besteht aus Brushes und Entities und wird mit einem Level-Editor wie z.B. Worldcraft entworfen. Vor der Benutzung muß er erst noch kompiliert werden.

Map: Im weiteren Sinn bedeutet Map genau das gleiche wie Level. Im engeren Sinn ist die Map-Datei gemeint, die mit einem Editor bearbeitet wird und noch nicht kompiliert wurde.

Model: Models sind dreidimensionale Objekte, mit denen der Spieler interagieren kann. Sie bestehen aus einem Drahtgittermodell und einer Textur („Skin“), die darübergelegt wird. Manche Models sind animiert (z.B. Monster), andere sind statisch (z.B. Munitionsboxen).

Teleporter: Teleporter sind Entities, die bei Aktivierung (z.B. durch Berühren) einen Spieler (oder auch ein Monster) an eine andere, vorher spezifizierte Stelle im Level versetzen.

Trigger: Trigger sind besondere Entities, die entweder durch Berührung oder durch Beschuß ausgelöst werden und dann bestimmte Ereignisse veranlassen. Zum Beispiel wäre es möglich, in der Mitte eines Raumes einen unsichtbaren Trigger zu plazieren, der bei Berührung durch den Spieler bewirkt, daß das Licht gedämpft und mehrere Monster in den Raum teleportiert werden.

QBSP3: Ein Compiler von id Software, der aus MAP-Dateien BSP-Dateien erzeugt.

QRAD3: QRAD3 berechnet die Beleuchtungsverhältnisse in einem Level und speichert diese Informationen in der BSP-Datei.

QVIS3: Dieses Programm nimmt Vorausberechnungen bezüglich der Sichtbarkeit in einem Level vor, um die Engine während der Laufzeit davon zu entlasten.

RMF: Das RMF-Format wird von Worldcraft benutzt. Es enthält gegenüber dem MAP-Format zusätzliche Informationen über einen Level, z.B. die Kamerapositionen im Editor.

Wally: Ein kostenloses Programm zum Erstellen und Bearbeiten von Texturen im WAL-Format.

Worldcraft: Ein Editor zum Erstellen von Quake-, Quake 2-, und Hexen 2-Levels.

B Dokumentation

B.1 Installation und Bedienung

Systemvoraussetzungen

Auf der Hardware-Seite sind mindestens ein Rechner der Pentium 200 MMX-Klasse, 64 MB Arbeitsspeicher und eine 3D-Beschleunigerkarte (z.B. Voodoo-, Voodoo2- oder Riva TNT-Chipsatz) erforderlich. Weiterhin benötigt man Quake 2 in der Version 3.20. Das dafür nötige Update erhält man z.B. unter `ftp://sunsite.doc.ic.ac.uk/Mirrors/ftp.cdrom.com/pub/idgames2/idstuff/quake2/`.

Installation

Falls noch nicht geschehen, sollte zuerst das Update für Quake 2 Version 3.20 installiert werden. Danach entpackt man Datei `unicats.zip` in das Quake 2-Verzeichnis. Es wird automatisch ein Unterverzeichnis namens UniCats erstellt. Im Quake 2-Verzeichnis wird eine Batch-Datei (`unicats.bat`) erstellt, mit der die Modifikation gestartet wird.

Start

Der Start erfolgt entweder über die Batch-Datei `unicats.bat` oder über den Kommandozeilenaufruf `quake2 -game unicats +map unicats`. Das Arbeitsverzeichnis muß das Quake 2-Verzeichnis sein, damit die Beispieldaten im Offline-Modus korrekt gelesen werden können.

Linke Maustaste	Waffe benutzen
Mittlere Maustaste	Vorwärtsbewegung
Rechte Maustaste	Vorwärtsbewegung
Mausbewegung nach links	Drehung nach links
Mausbewegung nach rechts	Drehung nach rechts
Mausbewegung nach oben	Blick nach oben richten
Mausbewegung nach unten	Blick nach unten richten

Tabelle B.1: Belegung der Maus

Steuerung

Es wurden verschiedene Konfigurationen vordefiniert, aus denen man sich eine geeignete Konfiguration zusammenstellen kann: Siehe Tabelle B.1, B.2 und B.3. Alternativ kann man sich auch seine eigene Belegung einstellen, allerdings sollte dies nicht innerhalb des Spiels geschehen, sondern durch Änderung der Datei `autoexec.cfg`. Hier ist das verwendete Schema einzuhalten.

1	Drehung nach links
2	Rückwärtsbewegung
3	Drehung nach rechts
4	Seitwärtsbewegung nach links
5	Vorwärtsbewegung
6	Seitwärtsbewegung nach rechts
7	Blick nach unten richten
8	Blick zentrieren
9	Blick nach oben richten
0	Ducken
Enter	Springen
+	Waffe benutzen

Tabelle B.2: Belegung des Ziffernblocks

$\rightarrow$	Seitwärtsbewegung nach rechts
$\leftarrow$	Seitwärtsbewegung nach links
$\uparrow$	Vorwärtsbewegung
$\downarrow$	Rückwärtsbewegung
Bild $\uparrow$	Blick nach oben richten
Bild $\downarrow$	Blick nach unten richten
Strg	Springen
Alt	Ducken
Leertaste	Waffe benutzen
ß	Zu Laserpointer wechseln
a	Seitwärtsbewegung nach links
d	Seitwärtsbewegung nach rechts
w	Springen
s	Rückwärtsbewegung

Tabelle B.3: Belegung sonstiger Tasten

Bedienung der Literaturrecherche

Anmeldung an das System erfolgt automatisch durch Betreten des Rechercheraumes. Um eine Anfrage zu formulieren, berührt man den Computer, woraufhin eine Eingabemaske erscheint. Dort stehen folgende Tasten zur Steuerung zur Verfügung:

Tab	Wechselt zum nächsten Feld
Umschalten+Tab	Wechselt zum vorhergehenden Feld
$\downarrow$	Wechselt zum nächsten Feld
$\uparrow$	Wechselt zum vorhergehenden Feld
Pos1	Bewegt den Cursor zum Anfang des Eingabefeldes
Ende	Bewegt den Cursor zum Ende des Eingabefeldes
$\leftarrow$	Bewegt den Cursor im Eingabefeld ein Zeichen nach links, bei der Katalogauswahl ein Feld nach links
$\rightarrow$	Bewegt den Cursor im Eingabefeld ein Zeichen nach rechts, bei der Katalogauswahl ein Feld nach rechts
Return	Bestätigung des ausgewählten Feldes

Die restlichen Tasten werden wie üblich zur Texteingabe verwendet.

Syntax der Anfrage:
(a, b, c stehen für beliebige Zeichenketten, die nicht „␣“ oder „+“ enthalten).

a␣b	Suche nach ‚a‘ ODER ‚b‘
a+b	Suche nach ‚a‘ UND ‚b‘
(a␣b)+c	Logische Klammerung
“a␣b”	Suche nach feststehendem Ausdruck, hier ‚a␣b‘

Nach dem Abschicken der Anfrage über das Feld „Suchen“ erscheinen entsprechend dem Ergebnis Bücher auf den Regalen (In der Standardeinstellung wird das Ergebnis aus der Datei `Suchergebnis.txt` eingelesen). Einige Informationen der Bücher werden durch verschiedene Stilmittel visualisiert. Von uns berücksichtigte Informationen sind die Verfügbarkeit (mit den Ausprägungen „sofort ausleihbar“, „später ausleihbar“, „nicht ausleihbar“), die Katalogzugehörigkeit („Universitätsbibliothek KA“, „Badische Landesbibliothek“, „Springer Verlag“), die Sprache („Deutsch“, „Englisch“, „Sonstige“) und das Erscheinungsjahr („ab 1996“, „1990 bis 1995“, „bis 1989“). Zur Visualisierung stehen zur Verfügung die Zuordnung zu einem Regal, die Farbe des Bucheinbandes und das Label auf dem Buchrücken. Auf einer Auswahltafel kann man die Zuordnungen von Informationen zu Stilmitteln frei kombinieren, indem man mit dem Laserpointer die entsprechenden Knöpfe auswählt. Nach einer Verzögerung von drei Sekunden erfolgt dann die Umsortierung der Bücher. Die Zuordnung der einzelnen Regale, Farben und Labels zu den jeweiligen Ausprägungen der Buchinformationen kann an der oberen Anzeige der Regale bzw. an der Legendentafel abgelesen werden. Auf den Regalen ist unten jeweils eine Anzeige angebracht, die die Gesamtzahl der Bücher auf diesem Regal darstellt. Da meist nicht alle Bücher eines Regals gleichzeitig dargestellt werden können, existieren rechts und links Schalter, um das Regal hoch bzw. runter fahren zu lassen. Sie werden durch den Laserpointer aktiviert. Allerdings kann ein Schalter nur betätigt werden, wenn er aufleuchtet.

Schießt man auf ein Buch, so wird dieses ausgewählt. Am unteren Bildschirmrand erscheint daraufhin eine Anzeige mit den Kurzinformationen bezüglich dieses Buches (Autor, Titel, Jahr sowie die Nummer des Buches auf dem Regal). Wird ein bereits angewähltes Buch wieder getroffen, so wird zur ausführlichen Anzeige gewechselt (Buchnummer bezüglich aller Bücher, Autor, Titel, Jahr, Sprache, Seitenzahl, Verlag, ISBN, Katalog, Verfügbarkeitsstatus). Zwischen diesen beiden Anzeigemodi kann auf diese Weise ständig gewechselt werden. Das ausgewählte Buch leuchtet rot, alle identischen Bücher leuchten grün, sofern sie gerade sichtbar sind[1]. Wenn sie nicht sichtbar sind, so blinken alle Schalter, die aktiviert werden müssen, um die zur Zeit nicht sichtbaren Bücher erscheinen zu lassen. Identisch sind zwei Bücher z.B., wenn sie in verschiedenen Sprachen vorliegen oder in unterschiedlichen Katalogen vorhanden sind[2].

Wenn man den Raum verläßt, wird man vom System abgemeldet und alle Bücher verschwinden.

Konsole-Variablen zur Steuerung des Online-Betriebs:

`Online` (Default ist `0`): Falls auf 1 gesetzt, wird versucht, Anfrage und Ergebnis über eine TCP-Verbindung zu senden bzw. zu empfangen.

`server_host_address` (Default ist `192.0.0.1`): IP-Adresse des Servers, der die Anfrage entgegennimmt

`server_tcp_port` (Default ist `44444`): Nummer des Ports, an dem der Server auf Anfragen wartet

B.2 Konsolen-Kommandos und -Variablen

Die Konsolen-Kommandos und -Variablen von Quake 2 erlauben es, viele wichtige Befehle schnell auszuführen und intern verwendete Variablen zu verändern. Insgesamt gibt es 143 Konsolen-Kommandos und 243 Konsolen-Variablen. Es soll deshalb an dieser Stelle nur eine Auswahl dieser Befehle erklärt werden. Eine ausführliche Übersicht, die auch Grundlage dieses Abschnitts war, findet man bei [10].

Um auf die Kommandos bzw. Variablen zuzugreifen, gibt es mehrere Möglichkeiten:

[1] Manchmal leuchten angewählte Bücher nicht auf. Die Ursache dafür ist uns unbekannt.

[2] In der Datei Suchergebnis.txt wurde die Identität zweier Bücher allerdings meistens willkürlich festgelegt.

- Direkt über die Konsole: Hier hat man den direktesten und umfassendsten Zugriff. Die Konsole wird aufgerufen durch die Taste links neben der 1 (Bei amerikanischem Tastaturlayout ist dies die ~-Tase, bei deutschem die ^-Taste). Außerdem ist sie über das Menü zugänglich (Options/Go to Console).

- Über das Menü: Hier kann man auf einen kleinen Teil der Variablen Einfluß nehmen (z.B. über Options/Effect Volume auf die Variable `s_volume`).

- Über entsprechend belegte Tasten: Die Escape-Taste führt zum Beispiel das Kommando `togglemenu` aus, daß zwischen Menü und Spiel umschaltet.

- Über die DLL-Datei (siehe Seite 121).

- Beim Programmstart können (durch das ‚+'-Zeichen gekennzeichnet) Kommandos an die Konsole übergeben werden. Beispiel: `quake2 +game unicats +map bib100`

- Beim Programmstart werden außerdem die beiden Scriptdateien `config.cfg` und `autoexec.cfg` ausgeführt, in denen hauptsächlich Variablen gesetzt werden, die jedoch auch Kommandos enthalten können.

Einige Kommandos (die sogenannten „action commands") bestehen aus korrespondierenden Paaren, die mit ‚+' bzw. ‚-' beginnen (z.B. `+attack`/`-attack`). Das ‚+'-Kommando startet dabei die Ausführung, das ‚-'-Kommando beendet sie. Wenn eine Taste mit einem solchen Kommando belegt ist, dann wird das ‚+'-Kommando beim Drücken und das ‚-'-Kommando beim Loslassen der Taste ausgeführt.

Variablen, die mit `sw_` beginnen, haben nur für den Software-Modus (siehe Abschnitt 3.2.1) Gültigkeit, analog dazu gelten mit `gl_` beginnende Variablen nur für den GL-Modus.

B.2.1 Konsolen-Kommandos

`+attack` Feuern der aktuellen Waffe.

`+back` Rückwärtsbewegung.

`+forward` Vorwärtsbewegung.

`+klook` Die Tasten, die mit `+forward` und `+back` belegt sind, fungieren wie `+lookup` und `+lookdown`.

`+left` Drehung nach links.

`+lookdown` Nach unten schauen.

`+lookup` Nach oben schauen.

+mlook Mausbewegung nach vorne bewirkt +lookup und Mausbewegung nach hinten bewirkt +lookdown. Durch negative Werte für m_pitch kann dieses Verhalten umgekehrt werden.

+movedown Ducken bzw. Abwärtsbewegung (in Wasser und auf Leitern).

+moveleft Seitwärtsbewegung nach links.

+moveright Seitwärtsbewegung nach rechts.

+moveup Springen bzw. Aufwärtsbewegung (in Wasser und auf Leitern).

+right Drehung nach rechts.

+speed Schnelle Fortbewegung (Rennen).

+strafe Die Tasten, die mit +left und +right belegt sind, fungieren wie +moveleft und +moveright.

alias Syntax: alias [<Name> "<Kommandos>"]
Dient zum Erzeugen einer Referenz auf ein Kommando oder auf eine Liste von Kommandos. Im zweiten Fall müssen die Kommandos durch Semikolons getrennt werden. Ein Aufruf ohne Parameter zeigt alle aktuellen Aliases an. Beispiel: alias jumpshot "+moveup;+attack;wait;-attack;moveup"

bind Syntax: bind <Taste> [<Kommandos>]
Weist einer Taste ein oder mehrere Kommandos zu. Dabei ist Taste im weiteren Sinne zu verstehen: Neben Keyboard- und Maus-Tasten können auch Joystick-Buttons oder ein Mausrad mit Funktionen versehen werden. Wird kein Kommando angegeben, so wird die aktuelle Belegung der entsprechenden Taste mitgeteilt. Siehe auch unbind und unbindall. Erlaubte Werte für <Taste> sind:
a-z, escape, f1-f12, pause, backspace, tab, semicolon, enter, shift, ctrl, alt, space, ins, home, pgup, del, end, pgdn, uparrow, downarrow, leftarrow, rightarrow, kp_slash, kp_minus, kp_plus, kp_enter, kp_5, kp_uparrow, kp_leftarrow, kp_rightarrow, kp_downarrow, kp_home, kp_end, kp_pgup, kp_pgdn, kp_ins, kp_del, mouse1-mouse3, aux1-aux32, joy1-joy4, mwheelup, mwheeldown
Beispiel: bind mouse1 +attack

bindlist Zeigt alle belegten Tasten zusammen mit ihrer Belegung an.

centerview Zentriert die Sicht des Spielers.

clear Löscht jeglichen Text von der Konsole.

cmdlist Gibt eine Liste aller Kommandos aus.

condump Syntax: condump <Dateiname>
Schreibt den Inhalt der Konsole in eine Textdatei.

connect Syntax: connect <IP-Adresse>[:<Port>]
Verbindet den Client mit dem angegebenen Server.

cvarlist Gibt eine Liste aller Variablen aus, wobei auch Flags und die aktuelle Belegung angezeigt werden.

demomap Syntax: demomap <Demoname>
Spielt eine Demo ab. Die Endung .dm2 muß angegeben werden. Siehe auch map.

disconnect Trennt die Verbindung zwischen Client und Server.

exec Syntax: exec <Dateiname>
Führt eine Scriptdatei aus. Scriptdateien haben üblicherweise die Endung .cfg oder .rc.

give Syntax: give <Item> [<Anzahl>]
Gibt dem Spieler das gewünschte Item. Der Parameter Anzahl ist nur bei Munition sinnvoll. Folgende Eingaben sind möglich: all, ammo, armor, health, weapons, shells, bullets, grenades, rockets, slugs, cells, combat armor, jacket armor, armor shard, power screen, power shield, shotgun, super shotgun, machinegun, chaingun, grenade launcher, rocket launcher, hyperblaster, bfg10k, adrenaline, ammo pack, bandolier, environment suit, invulnerability, quad damage, rebreather, silencer, airstrike marker, ancient head, blue key, commander's head, data cd, data spinner, power cube, pyramid key, red key, security pass. all steht für alle Items, ammo für volle Munition, armor für volle Rüstung, health vor volle Gesundheit und weapons für alle Waffen. Beispiele: give shotgun, give shells 20, give all.

god Schaltet den „God-Mode“ (Unverwundbarkeit) an und aus.

gun_model Syntax: gun_model [<Dateiname>]
Dieser Befehl dient zusammen mit den nächsten beiden Befehlen zum Testen von Waffen-Models. Jeder Frame der Animation kann so einzeln betrachtet werden. Wird der Befehl ohne Parameter aufgerufen, so wird zum normalen Waffen-Model zurückgekehrt. Beispiel: gun_model weapons/v_shotg.

gun_next Zeigt den nächsten Frame der Animation des aktuellen Waffen-Models an.

gun_prev Zeigt den vorhergehenden Frame der Animation des aktuellen Waffen-Models an.

invdrop Dient zum Ablegen des aktuellen Items (Waffen, Quad-Damage etc.).

`inven` Zeigt das Inventar an.

`invnext` Wählt das nächste Item aus dem Inventar des Spielers aus.

`invnextp` Wählt das nächste Powerup aus dem Inventar aus.

`invnextw` Wählt die nächste Waffe aus dem Inventar aus.

`invprev` Wählt das vorhergehende Item aus dem Inventar aus.

`invprevp` Wählt das vorhergehende Powerup aus dem Inventar aus.

`invprevw` Wählt die vorhergehende Waffe aus dem Inventar aus.

`invuse` Benutzt das angewählte Item.

`kill` Der Spieler begeht Selbstmord. Sinnvoll ist dies z.B., wenn man im Single Player Spiel an einer Stelle feststeckt und von vorne beginnen möchte.

`map` Syntax: `map <Dateiname>`
Die eigentliche Bedeutung dieses Befehls ist das Laden von Maps. Es können jedoch auch Demos abgespielt und Bilder angezeigt werden. In diesen Fällen muß die Dateiendung jedoch angegeben werden. Beispiele: `map base1`, `map demo1.dm2`, `map colormap.pcx`.

`messagemode` Fordert zur Eingabe einer Nachricht auf, die an alle Spieler auf dem Server geschickt wird.

`messagemode2` Fordert zur Eingabe einer Nachricht auf, die an alle Teammitglieder auf dem Server geschickt wird.

`noclip` Schaltet den „No clipping"-Modus an bzw. aus. In diesem Modus ist der Spieler sozusagen körperlos und kann sich durch Wände, Objekte und andere Spieler bewegen.

`notarget` Schaltet den „No Target"-Modus an bzw. aus. Wenn dieser Modus aktiv ist, so ist der Spieler für die Monster unsichtbar.

`quit` Beendet Quake 2.

`record` Syntax: `record <Demoname>`
Startet das Aufnehmen einer Demo. Zum Beenden dient der Befehl `stop`. Die Demos werden in das `demos`-Verzeichnis des aktuellen `game`-Verzeichnisses geschrieben (also standardmäßig in das `quake2/baseq2/demos`-Verzeichnis).

`screenshot` Erzeugt einen Screenshot des aktuellen Bildes. Im Software-Modus wird das

PCX-Format benutzt, ansonsten das TGA-Format.

sizedown Verkleinert das angezeigte Bild. Die gewünschte Größe kann auch direkt über `viewsize` eingestellt werden.

sizeup Vergrößert das angezeigte Bild. Die gewünschte Größe kann auch direkt über `viewsize` eingestellt werden.

stop Stoppt die Aufnahme einer Demo. Siehe auch `record` und `demomap`.

timerefresh Syntax: `timerefresh <Parameter>`
Dieser Test dient zum Testen der fps-Rate (Frames per Second). Es wird eine 360°-Drehung ausgeführt und anschließend ein fps-Wert angezeigt. Falls ein Parameter angegeben wird, so wird die Berechnung nur im Speicher ausgeführt.

toggleconsole Zeigt bzw. entfernt die Konsole.

unbind Syntax: `unbind <Taste>`
Hebt eine mit `bind` gemachte Belegung wieder auf.

unbindall Hebt die Belegung aller Tasten auf. Ausgenommen sind lediglich die Escape-Taste und die Taste links von der 1, ^ bzw. ~ (je nach Tastatur).

use Syntax: `use <Item>`
Dient zum Benutzen eines Items. Folgende Eingaben sind möglich: bfg10k, blaster, chaingun, environment suit, grapple, grenade launcher, grenades, hyperblaster, invulnerability, machine gun, power screen, power shield, quad damage, railgun, rebreather, rocket launcher, shotgun, silencer, super shotgun.
Beispiel: `use rocket launcher`

weaplast Schaltet zur zuletzt benutzten Waffe um.

weapnext Schaltet zur nächsten Waffe im Inventar um.

weapprev Schaltet zur vorhergehenden Waffe im Inventar um.

B.2.2 Konsolen-Variablen

Hinter den Variablennamen ist in Klammern der Default-Wert angegeben.

cd_nocd (0) Schaltet die Verwendung von CD-Musik an und aus.

cheats (0) Legt fest, ob Cheat-Codes (z.B. `god`, `noclip`, `notarget`) erlaubt sind.

cl_entities (1) Legt fest, ob Entities dargestellt werden oder nicht.

cl_footsteps (1) Legt fest, ob beim Gehen Schritte zu hören sind.

cl_forwardspeed (200) Die maximale Geschwindigkeit für Vorwärts- und Rückwärtsbewegung.

cl_gun (1) Legt fest, ob die Waffe des Spielers angezeigt wird.

cl_lights (1) Schaltet das „dynamic lighting“ ein bzw. aus.

cl_pitchspeed (150) Die Geschwindigkeit, mit der der Spieler nach oben und unten schaut (nur bei Verwendung des Keyboards).

cl_run (0) Legt fest, ob sich der Spieler immer schnell bewegt.

cl_sidespeed (200) Die maximale Geschwindigkeit für Seitwärtsbewegung.

cl_upspeed (200) Die maximale Geschwindigkeit für Auf- und Abwärtsbewegung.

cl_yawspeed (140) Die Geschwindigkeit, mit der sich der Spieler dreht (nur bei Verwendung des Keyboards).

crosshair (1) Legt das Aussehen des Fadenkreuzes fest:
0: kein Fadenkreuz
1: Kreuz
2: Punkt
3: Winkel

debuggraph (0) Schaltet den Debuggraphen an bzw. aus. Siehe auch Seite 122.

freelook (0) Wenn Freelook aktiviert ist, dann haben die Tasten für Vorwärts-/Rückwärtsbewegung die Funktion nach oben/unten schauen.

game Legt das Verzeichnis fest, in dem zusätzlich zum `baseq2`-Verzeichnis nach Spieldaten (Maps, Texturen, Models, etc.) gesucht wird.

gl_dynamic (1) Schaltet das „dynamic lighting“ an bzw. aus. Bei Verwendung des Software-Renderers wird dies über `cl_lights` gesteuert. Default ist 1.

gl_ext_palettedtexture (1) Legt fest, ob Texturen mit eigener Farbpalette verwendet werden können.

gl_mode (3) Legt die Bildschirmauflösung bei Verwenung eines GL-Renderers fest. Die erlaubten Werte liegen zwischen 0 (320 x 200) und 9 (1600 x 1200). Default ist 3 (640 x 480).

gl_texturemode (`GL_LINEAR_MIPMAP_NEAREST`) Legt fest, wie die Texturen gezeich-

net werden. Zur Auswahl stehen
1. `GL_NEAREST`: Verwendung von „Nearest Neighbor“-Interpolation
2. `GL_LINEAR`: Verwendung von linearer Interpolation
3. `GL_NEAREST_MIPMAP_NEAREST`: Wie 1., zzgl. Mipmapping für bilineare Hardware
4. `GL_LINEAR_MIPMAP_NEAREST`: Wie 2., zzgl. Mipmapping für bilineare Hardware
5. `GL_NEAREST_MIPMAP_LINEAR`: Wie 1., zzgl. Mipmapping für trilineare Hardware
6. `GL_LINEAR_MIPMAP_LINEAR`: Wie 2., zzgl. Mipmapping für trilineare Hardware

hand (0) Legt fest, wo sich die Waffe des Spielers befindet. Mögliche Werte:
0: Rechts
1: Links
2: In der Mitte (in diesem Fall wird die Waffe nicht dargestellt)

intensity (2) Bestimmt die Helligkeit der OpenGL-Darstellung.

lookspring (1) Wenn diese Variable auf 1 gesetzt ist, wird nach der Benutzung von `+mlook` der Bildschirm automatisch zentriert.

lookstrafe (0) Wenn diese Variable auf 1 gesetzt ist, haben während der Benutzung von `+mlook` Tasten mit der Belegung `+left` bzw. `+right` die Funktion `+moveleft` bzw. `+moveright`.

m_filter (0) Wenn diese Variable auf 1 gesetzt ist, werden die von der Maus erhaltenen Eingaben interpoliert. Dies führt zu einer glatteren Bewegung, vergrößert allerdings die Verzögerung zwischen Mausbewegung und Spielerbewegung.

m_forward (1) Die Empfindlichkeit der Maus für Vorwärts- und Rückwärtsbewegung.

m_pitch (0.022) Die Empfindlichkeit der Maus für nach oben/unten schauen.

m_side (0.8) Die Empfindlichkeit der Maus für Seitwärtsbewegung.

m_yaw (0.022) Die Empfindlichkeit der Maus für nach links/rechts drehen.

name (`Player`) Der Name des Spielers.

r_drawentities (1) Legt fest, ob Entities dargestellt werden.

r_drawworld (1) Legt fest, ob Wände etc. dargestellt werden.

r_dspeeds (0) Wenn diese Variable auf 1 gesetzt ist, wird die Render-Geschwindigkeit (in Millisekunden) angezeigt.

r_fullbright (0) Schaltet zwischen normaler und maximaler Beleuchtung um.

r_novis (0) Schaltet die Verwendung der in der BSP-Datei gespeicherten VIS-Tabellen aus bzw. an.

r_speeds (0) Wenn diese Variable auf 1 gesetzt ist, werden Render-spezifische Statistiken angezeigt.

s_volume (0.7) Die Lautstärke für die Soundeffekte.

scr_centertime (2.5) Die Anzeigedauer in Sekunden für die in der Bildschirmmitte angezeigten Nachrichten.

scr_conspeed (3) Die Geschwindigkeit, mit der die Konsole erscheint bzw. verschwindet.

sensitivity (3) Die Empfindlichkeit der Maus.

skill (0) Legt den Schwierigkeitsgrad fest. Mögliche Werte:
0: easy (Einfach)
1: medium (Mittel)
2: hard (Schwer)
3: hard+ (Sehr schwer)

sv_gravity (800) Legt die Schwerkraft innerhalb des Spiels fest.

sw_drawflat (0) Wenn diese Variable auf 1 gesetzt ist, werden keine Texturen gezeichnet.

sw_draworder (0) Wenn diese Variable auf 1 gesetzt ist, werden die Wände in umgekehrter Reihenfolge (also von vorne nach hinten) gezeichnet.

sw_mode (0) Legt die Bildschirmauflösung für den Software-Modus fest. Die erlaubten Werte liegen zwischen 0 (320 x 480) und 9 (1600 x 1200).

timedemo (0) Wenn diese Variable auf 1 gesetzt ist, wird eine mit `demomap` oder `map` abgespielte Demo mit maximaler Geschwindigkeit angezeigt. Nach dem Ablauf erhält man einen fps-Wert (Frames per Second), der Aufschluß über die Geschwindigkeit des verwendeten Systems gibt.

timescale (1) Legt das Verhältnis von der spielinternen zur wirklichen Zeit fest.

viewsize (100) Legt die Bildgröße fest. Siehe auch `sizedown` und `sizeup`.

B.3 DLL-Quelltext

In diesem Abschnitt wird versucht, zuerst den offiziellen Quelltext zur game dll näher zu erläutern, anschließend werden die eigenen Erweiterungen vorgestellt. Die verschiedenen Spielmöglichkeiten werden wie folgt abgekürzt: DM steht für Deathmatch (hier spielen mehrere Spieler gegeneinander), CTF steht für Capture the Flag (hier spielen mehrere Teams gegeneinander), Coop für Cooperative (hier spielen mehrere Spieler miteinander). Die Ausführungen bezüglich des offiziellen Quellcodes basieren ausschließlich auf Erfahrungen aus der Arbeit mit diesem, so daß die Richtigkeit der folgenden Ausführungen nicht garantiert werden kann. Sie spiegeln lediglich die Erkenntnisse – teilweise auch nur Vermutungen – wider, auf deren Basis die Erweiterungen erfolgt sind (bei Unsicherheiten wird ein ? der Erklärung vorangestellt, sofern überhaupt ein Erklärungsversuch erfolgt). Das folgende soll dem besseren Verständnis und der leichteren Einarbeitung in die Materie dienen. Es wird versucht, die „wichtigen“ Strukturen und Funktionen detailliert zu beschreiben sowie Interaktionen zu erfassen. Somit kann das Ganze auch als einfache „Referenz“ verstanden werden.

Inhaltsübersicht

B.3.1 Aufbau

Der Quellcode ist auf 74 Dateien mit einer Gesamtgröße von rund 1 MB verteilt. Diese werden nun im Einzelnen kurz vorgestellt. Die zwei Dateien mit vorangestelltem q_ (quake) sind eigentlich Teil des game engine Quellcodes, die neunzehn mit vorangestelltem g_ (game) sind die zentralen Dateien der game dll; die fünf Dateien mit p_ (player) beeinhalten speziellen Code für die Spieler-Entities und die siebenundvierzig mit m_ (monster) beginnenden Dateien speziellen Code für die Monster-Entities.

game.h Deklaration der vom Server aus sichtbaren Informationen der game dll – wird auch von der game engine gelesen ⇒ nicht ändern!

q_shared.h Deklaration von programmweiten Informationen – diese werden von *allen* Programmmodulen (also auch den game engine Modulen) zu Beginn gelesen ⇒ nicht ändern!

q_shared.c Allgemeine, von *allen* Programmmodulen benutzbare Funktionen ⇒ nicht ändern!

g_local.h Lokale Deklarationen zur game dll – wird zu Beginn aller *.c Dateien (außer q_shared.c) gelesen

m_*.h Deklaration der einzelnen Frames der Figuren (Monster und Spieler) – wurden vom Modelling-Programm erstellt ⇒ nicht ändern!

m_*.c Die jeweiligen Monster Entities, deren Animation und Verhalten

g_ai.c Die künstliche Intelligenz der Monster

g_chase.c Funktionalität für Zuschauermodus im DM; man steuert sozusagen eine „Kamera“ durch den Level und kann diese auch anderen Spielern automatisch folgen lassen, so daß man deren Sicht auf die 3D-Welt auf seinem Bildschirm betrachten kann

g_cmds.c Bearbeitung der an der Konsole eingegebenen Client-Befehle (i.d.R. mit cmd␣ eingeleitet)

g_combat.c Behandlung von Kampfhandlungen

g_func.c Definition sich bewegender Entities: Aufzüge (func_plat), sich drehende Entities (func_rotating), Knöpfe (func_button), Türen (func_door, func_door_rotating), sich bewegendes Wasser (func_water), Plattformen (func_train), Förderbänder (func_conveyor) und andere (trigger_elevator, func_timer, func_door_secret, func_killbox)

g_items.c Definition der Items wie Erste-Hilfe-Pakete (health), Munition (ammo), Rüstung (armor), Waffen, etc.

g_main.c Schnittstellenfunktionen und -variablen zur game engine

g_misc.c Verschiedene Entities wie einfache Wände (func_wall), Explosives (func_explosive, misc_explobox), Gegenstände (func_object), virtuelle Punkte z.B. für Plattformen, Monster (path_corner, point_combat), Lichter (light, info_null, info_notnull) und andere (func_areaportal, misc_blackhole, misc_eastertank, misc_easterchick, misc_easterchick2, monster_commander_body, misc_banner, misc_deadsoldier, misc_bigviper, misc_viper, misc_viper_bomb, light_mine1, light_mine2, misc_gib_arm, misc_satellite_dish, misc_strogg_ship, misc_gib_leg, misc_gib_head, target_character, target_string, func_clock, misc_teleporter)

g_monster.c Waffen der Monster

g_phys.c Behandlung der physikalischen Eigenschaften von Entities: Bewegung, Kollision, Schwerkraft, etc.

g_save.c Funktionen zum Laden und Speichern von Spielständen und Levels

g_spawn.c Funktionen zur Erzeugung der Entities (insbesondere der ganzen Welt)

g_svcmds.c Bearbeitung der an der Konsole eingegebenen Server-Befehle (mit sv␣ eingeleitet)

g_target.c Target-Entities (werden von anderen Entities ausgelöst und interagieren mit der Welt) wie Lautsprecher (target_speaker), Geheimnisse (target_secret), Laser (target_laser) und andere (target_temp_entity, target_help, target_goal, target_explosion, target_changelevel, target_splash, target_spawner, target_blaster, target_crosslevel_trigger, target_crosslevel_target, target_lightramp, target_earthquake)

g_trigger.c Trigger-Entities (lösen andere Entities oder Ereignisse aus; werden selbst z.B. durch Annäherung, Beschuß, Berührung oder andere Entities ausgelöst) wie Zähler (trigger_counter), Gebrauch eines Schlüssels (trigger_key) und andere (trigger_multiple, trigger_once, trigger_relay, trigger_always, trigger_push, trigger_hurt, trigger_gravity, trigger_monsterjump)

g_turret.c Turret-Entity (Geschützturm) (turret_base, turret_breach, turret_driver)

g_utils.c Verschiedene Hilfsfunktionen für die game dll

g_weapon.c Verhalten der Waffen

p_client.c Spieler-Client-Server Interaktion (Verbinden, Erzeugen, etc.)

p_hud.c (Head Up Display) Behandlung der über der 3D-Welt liegenden Grafik wie Statusbar, Hilfeanzeige, etc.

`p_trail.c` Player-Trail Funktionalität: Liste mit Entities, die Punkte darstellen, an denen der Spieler zuletzt war; wird von den Monstern benutzt, um die Spieler zu verfolgen

`p_view.c` Berechnungen zur Sicht des Spielers auf die 3D-Welt (Blickrichtung, Darstellung der aktivierten Waffe, Effekte wie Gehbewegung, Waffenrückstoß, Rückstoß von Explosionen, Farbschleier über das Bild legen, etc.)

`p_weapon.c` Waffen der Spieler

B.3.2 Zentrale Datenstrukturen und Funktionen

Erläuterungen zur folgenden Darstellung: Funktionen, die im anschließenden Abschnitt erklärt werden, werden folgendermaßen dargestellt:

Rückgabewert Funktionsname(Parameter)

Variablen sehen folgendermaßen aus:
Typ Variablenname Erläuterungen

Konstanten hingegen:
Konstante (Wert der Konstante) Erläuterungen

Wird im Text auf eine Funktion verwiesen, so werden die Parameter i.d.R. weggelassen, die leeren Klammern werden jedoch zur besseren Unterscheidung von Variablen hinzugefügt (Funktionsname() ⇔ Variablenname). Bei Funktionen ohne Parameter wird ausdrücklich void als Parameter angegeben. Konstanten werden oft als Flags benutzt, d.h. sie stellen ein Bit z.B. von einer Integer-Variablen dar, so daß i.d.R. mit Bit-Operatoren auf diesen Variablen gearbeitet wird. Die Werte der Konstanten entsprechen dann Zweierpotenzen.

Einfache Datentypen und Konstanten

byte	unsigned char
qboolean	false(0), true(1)
vec_t	float
vec3_t	vec_t[3]
vec5_t	vec_t[5]

Orientierungskomponenten eines Vektors vec3_t:

PITCH	(0) Drehung um y-Achse (hoch/runter)
YAW	(1) Drehung um z-Achse (links/rechts)
ROLL	(2) Drehung um x-Achse (nach links/rechts kippen)

Richtungskomponenten eines Vektors vec3_t:

0	x-Koordinate
1	y-Koordinate
2	z-Koordinate
vec3_origin	(0,0,0) Welt-Ursprung
FRAMETIME	(0.1) Zeitintervall der Server-Frames, $\frac{1}{10}$ s
M_PI	(3.141...) π
MAX_QPATH	(64) maximale Länge eines Quake 2-game-Pfadnamens
MAX_OSPATH	(128) maximale Länge eines Dateisystem-Pfadnamens
MAX_CLIENTS	(256) Maximum an Clients
MAX_EDICTS	(1024) Maximum an Entities
MAX_LIGHTSTYLES	(256) Maximum an Lichtstilen
MAX_MODELS	(256) Maximum an Models
MAX_SOUNDS	(256) Maximum an Geräuschen
MAX_IMAGES	(256) Maximum an Bildern
MAX_ITEMS	(256) Maximum an Items
MAX_GENERAL	(2*MAX_CLIENTS) maximale Anzahl an generellen Configstrings
MAX_INFO_KEY MAX_INFO_VALUE	} (64) maximale Länge der Schlüssel (Name) und Werte einer Cvar im client→pers.userinfo Info-String (S. 137)
MAX_INFO_STRING	(512) maximale Länge des client→pers.userinfo Info-Strings

Wichtige game dll-Variablen

gitem_t	itemlist	Liste aller existierenden Items in g_items.c, dort werden auch alle Parameter der Struktur gitem_t eingetragen
game_locals_t	game	lokale Daten bezüglich der game dll
level_locals_t	level	lokale Daten bezüglich des momentanen Levels
edict_t	*g_edicts	Zeiger auf die Datenfelder aller Entities im Level (Index [0]: Welt, Index [1–game.maxclients]: Clients (Spieler), Index [game.maxclients+1–game.maxentities−1]: restliche Entities)
edict_t	*world	als (&g_edicts[0]) definiert
spawn_temp_t	st	temporärer Speicher für einige Entity-Feld-Werte beim Laden der Entities zu Beginn eines Levels
game_import_t	gi	von der game engine gelieferte Funktionen
game_export_t	globals	von der game dll exportierte Funktionen

Konsolenvariablen (Cvars) der game dll

cvar_t		
char	*name	Namensstring
char	*string	Wert bei String-Cvars
char	*latched_string	Stringwert-Zwischenspeicher für CVAR_LATCH-Cvars
int	flags	Flags der Cvar:
CVAR_ARCHIVE		(1) die Cvar wird mit in die Datei `vars.rc` gespeichert
CVAR_USERINFO		(2) die Cvar wird zum String client→pers.userinfo hinzugefügt (wird in der Funktion ClientThink() als Parameter an die game dll übergeben)
CVAR_SERVERINFO		(4) die Cvar wird zum String serverinfo hinzugefügt (im Code der game engine)
CVAR_NOSET		(8) die Cvar kann von der Konsole aus nicht verändert werden, dies ist nur von der Kommandozeile beim Programmstart aus möglich
CVAR_LATCH		(16) Änderungen werden bis zum nächsten Server-Start zwischengespeichert und haben erst dann Gültigkeit
qboolean	modified	wird beim Ändern der Cvar gesetzt
float	value	Wert bei Zahlen-Cvars
cvar_t	*next	Zeiger auf die nächste Cvar

In game dll definierte Cvars

Das Folgende bezieht sich immer auf die Werte der Cvars cvar→value bzw. cvar→string, nicht auf die Zeiger vom Typ cvar_t *.

cvar_t	*deathmatch	auf 1, wenn dies ein DM-Spiel ist
cvar_t	*dmflags	Flags im DM:

DF_NO_HEALTH	(1) keine Erste-Hilfe-Items im Level
DF_NO_ITEMS	(2) keine Powerup-Items im Level
DF_WEAPONS_STAY	(4) die Waffe bleibt nach dem Aufnehmen durch einen Spieler für andere weiter vorhanden; hat der Spieler schon eine Waffe dieser Sorte, kann er sie nicht aufnehmen
DF_NO_FALLING	(8) Spieler erleiden durch Herunterfallen keinen Schaden
DF_INSTANT_ITEMS	(16) Powerup-Items werden sofort benutzt und können nicht mehr bis zu ihrer Verwendung umhergetragen werden
DF_SAME_LEVEL	(32) Nach Erreichen des timelimit oder des fraglimit wird im gleichen Level weitergespielt
DF_SKINTEAMS	(64) Spieler mit gleicher Skin sind im selben Team
DF_MODELTEAMS	(128) Spieler mit dem gleichen Model (*und* gleicher Skin) sind im gleichen Team
DF_NO_FRIENDLY_FIRE	(256) Spieler des gleichen Teams können sich keinen Schaden zufügen
DF_SPAWN_FARTHEST	(512) Spieler werden möglichst weit entfernt von allen anderen Spielern eingesetzt
DF_FORCE_RESPAWN	(1024) Spieler werden nach dem Tod sofort wieder eingesetzt und können somit nicht mehr solange in der Intermission bleiben wie sie wollen
DF_NO_ARMOR	(2048) keine Rüstung im Level
DF_ALLOW_EXIT	(4096) den Spielern ist das Verlassen des Levels durch „normale“ Level-Ausgänge erlaubt
DF_INFINITE_AMMO	(8192) Spieler haben unbegrenzte Munition, keine Munition und keine BFG im Level
DF_QUAD_DROP	(16384) Hat ein Spieler Quad Damage aktiviert und stirbt, läßt er auch das Quad Damage Item fallen, so daß es ein anderer sofort aufnehmen und benutzen kann (es bleibt aber nur für die (theoretische) Restlaufzeit liegen)

DF_FIXED_FOV	(32768) Spieler können fov (Cvar: field of view – Breite des Sichtfeldes des Spielers) nicht ändern, bleibt immer 90
cvar_t *coop	auf 1, wenn dies ein Coop-Spiel ist
cvar_t *skill	Schwierigkeitsgrad im Spiel:
0	easy, einfach
1	medium, mittel
2	hard, schwer
3	nightmare, am schwersten
cvar_t *fraglimit cvar_t *timelimit	Hat ein Spieler im DM fraglimit oft andere Spieler getötet oder läuft der Level schon timelimit lange, so wird in den nächsten Level gewechselt.
cvar_t *password	Paßwort, um am Spiel auf einem Server teilnehmen zu können
cvar_t *spectator_password	Paßwort, um im Spiel in den Zuschauermodus zu wechseln
cvar_t *needpass	Wenn ein Paßwort (s.o.) geändert wird, wird needpass gesetzt, damit die Engine Bescheid weiß und die Passwörter der Clients erneut abgefragt werden können.
cvar_t *maxclients	Maximale Anzahl an Clients in einem Spiel ($\leq$ 256)
cvar_t *maxentities	Maximale Anzahl Entities in einem Spiel ($\leq$ 1024)
cvar_t *maxspectators	Maximale Anzahl an Clients im Zuschauermodus in einem Spiel
cvar_t *g_select_empty	Zeigt an, ob man eine Waffe aktivieren kann, für die man keine Munition hat
cvar_t *dedicated	Zeigt an, ob der Server alleinstehend ist (ansonsten läuft auf dem Server-Rechner noch ein Client mit)
cvar_t *filterban	siehe IP-Filtering (S. 132)
cvar_t *sv_maxvelocity	maximale Bewegungsgeschwindigkeit der Objekte
cvar_t *sv_gravity	Gravitation im Spiel
cvar_t *sv_rollspeed cvar_t *sv_rollangle	werden für die Berechnung der ROLL-Bewegung des Spieler-Models benötigt
cvar_t *run_pitch cvar_t *run_roll	PITCH und ROLL der Spielersicht beim Vorwärtsrennen
cvar_t *bob_pitch cvar_t *bob_roll cvar_t *bob_up	PITCH, ROLL und Aufwärtsbewegung der Spielersicht beim Vorwärtsgehen

cvar_t	*gun_x	Für die Waffenentwicklung vorhandene Möglichkeit, um die Lage der Waffe um diese Offsetwerte zu verändern
cvar_t	*gun_y	
cvar_t	*gun_z	
cvar_t	*flood_msg	Einstellungen für „Chat“-Nachrichten-Flußkontrolle
cvar_t	*flood_persecond	
cvar_t	*flood_waitdelay	
cvar_t	*sv_cheats	Zeigt an, ob im DM „Cheat“-Befehle erlaubt sind (God, Noclip, Notarget, Give)
cvar_t	*sv_maplist	Liste von „level.mapnames“, in der die Reihenfolge der Level festgelegt ist, so daß beim Erreichen von fraglimit oder timelimit der nächste Level geladen werden kann

Liste der Items

item_armor_body	Body Armor (Rüstung)
item_armor_combat	Combat Armor (Rüstung)
item_armor_jacket	Jacket Armor (Rüstung)
item_armor_shard	Shard Armor (Rüstung)
item_power_screen	Power Screen (Rüstung)
item_power_shield	Power Shield (Rüstung)
weapon_blaster	Blaster (Waffe)
weapon_shotgun	Shotgun (Schrotflinte)
weapon_supershotgun	Supershotgun (doppelläufige Schrotflinte)
weapon_machinegun	Machinegun (Maschinenpistole)
weapon_chaingun	Chaingun (Maschinengewehr)
ammo_grenades	Grenades (Granaten)
weapon_grenadelauncher	Grenadelauncher (Granatenwerfer)
weapon_rocketlauncher	Rocketlauncher (Raketenwerfer)
weapon_hyperblaster	Hyperblaster (Waffe)
weapon_railgun	Railgun (Waffe)
weapon_bfg	BFG 10k (Waffe, im folgenden nur noch BFG genannt)
ammo_shells	Shells (Patronen für Schrotflinten)
ammo_bullets	Bullets (Kugeln für Maschinengewehr und Maschinenpistole)
ammo_cells	Cells (Energiezellen für Hyperblaster, BFG, Power Screen und Power Shield)
ammo_rockets	Rockets (Raketen)
ammo_slugs	Slugs (Munition für Railgun)

item_quad	Quad Damage Powerup (vierfacher Schaden beim Gegner)
item_invulnerability	Invulnerability Powerup (Unverwundbarkeit)
item_silencer	Silencer Powerup (Schalldämpfer)
item_breather	Breather Powerup (Sauerstoffgerät)
item_enviro	Environmentsuit Powerup (Schutzanzug)
item_ancient_head	Ancient Head (erhöht maximale Gesundheit)
item_adrenaline	Adrenaline (setzt Gesundheit auf maximale Gesundheit)
item_bandolier	Bandolier (mehr Munition transportieren)
item_pack	Ammo Pack (dto., auch für Raketen und Granaten)
key_data_cd	Data CD (Schlüssel)
key_power_cube	Power Cube (Schlüssel)
key_pyramid	Pyramid Key (Schlüssel)
key_data_spinner	Data Spinner (Schlüssel)
key_pass	Security Pass (Schlüssel)
key_blue_key	Blue Key (blauer Schlüssel)
key_red_key	Red Key (roter Schlüssel)
key_commander_head	Commander's Head (Schlüssel)
key_airstrike	Airstrike Marker (Schlüssel)
item_health	normales Erste-Hilfe-Paket
item_health_small	kleines Erste-Hilfe-Paket
item_health_large	großes Erste-Hilfe-Paket
item_health_mega	Erste-Hilfe-Powerup

Strukturtyp game_locals_t

char	helpmessage1[512]	primäres Ziel in der Hilfeanzeige
char	helpmessage2[512]	sekundäres Ziel in der Hilfeanzeige
int	helpchanged	wird gesetzt, wenn der Hilfetext wechselt (Help-Icon leuchtet, Geräusch wird ausgegeben)
gclient_t	*clients	Zeiger auf die Client-Datenstrukturen (Index [0–game.maxclients−1])
char	spawnpoint[512]	String der Punkte, an denen die Clients in den Level eingesetzt werden
int	maxclients	Speicherung des Wertes der Cvars maxentities und maxclients für schnelleren Zugriff
int	maxentities	
int	serverflags	für Cross Level Trigger benutzt
int	num_items	aktuelle Anzahl an Items
qboolean	autosaved	Zeigt beim Laden eines Spielstandes an, ob der Spielstand automatisch gespeichert wurde

Strukturtyp level_locals_t

int	framenum	Anzahl der im Level vergangenen Frames (jeweils 0.1 Sekunden lang)
float	time	Levelzeit (=level.framenum$*$0.1)
char	level_name[MAX_QPATH]	deskriptiver Level-Name (Outer Base, . . .)
char	mapname[MAX_QPATH]	Server-Level-Name (base1, . . .)
char	nextmap[MAX_QPATH]	Nächster Level, wenn fraglimit oder timelimit im DM überschritten
float	intermissiontime	An diesem Zeitpunkt wurde die Intermission gestartet (Pause nach Beendigung einer Mission oder eines Levels im DM)
char	*changemap	Level, mit dem es nach der Intermission weitergeht
int	exitintermission	wenn Intermission verlassen wird auf 1
vec3_t	intermission_origin	Position des Spielers während der Intermission
vec3_t	intermission_angle	Orientierung des Spielers während der Intermission
edict_t	*sight_client	wird von Monstern zur Verfolgung der Clients benutzt
edict_t	*sight_entity	
int	sight_entity_framenum	
edict_t	*sound_entity	
int	sound_entity_framenum	
edict_t	*sound2_entity	
int	sound2_entity_framenum	
int	pic_health	Index des Erste-Hilfe-Icons
int	total_secrets	Gesamtzahl der Geheimnisse
int	found_secrets	Anzahl gefundener Geheimnisse
int	total_goals	Gesamtzahl der Missionsziele
int	found_goals	Anzahl erfüllter Missionsziele
int	total_monsters	Gesamtzahl der Monster
int	killed_monsters	Anzahl der getöteten Monster
edict_t	*current_entity	momentan von G_RunFrame() bearbeitetes Entity
int	body_que	Index für Body Que (für Körper von toten Spielern reservierte Entities)
int	power_cubes	für Power Cubes im Coop benutzt

Strukturtyp edict_t – Entity Struktur

Von den Strukturtypen edict_t und gclient_t existiert jeweils eine lange Client-Version (g_local.h) und eine kurze Server-Version (game.h), der Teil der Strukturtypen, den der Server benötigt. Die Teile der Strukturtypen, die auch der Server benutzt, sollten nicht geändert werden (auch nicht die Reihenfolge der Felder)! Viele Felder in edict_t werden vom Level-Editor gesetzt (und i.d.R. nicht mehr verändert); diese Felder sind mit LE: gekennzeichnet; natürlich können vom Level-Editor aus alle vorhandenen Felder (incl. die in spawn_temp_t) mit initialen Werten versehen werden.

edict_t

entity_state_t	s	Entity Statusinformationen, welche von den Clients für das Rendering benötigt werden; wird vom Server jeden Server-Frame als Update-Nachricht an die Clients übermittelt
gclient_t	*client	Zeiger auf die Client Struktur; NULL, wenn das Entity kein Spieler ist
qboolean	inuse	zeigt an, ob das Entity gerade in Gebrauch ist (also in der Welt existiert) oder nicht
int	linkcount	Zähler, der von gi.linkentity() erhöht wird (somit wird es möglich, festzustellen, ob sich das Entity bewegt hat)
link_t	area	hat mit der internen Darstellung der Welt und der dortigen Position des Entities zu tun
int	num_clusters	
int	clusternums[MAX_ENT_CLUSTERS]	
int	headnode	
int	areanum	
int	areanum2	
int	svflags	Serverflags:
SVF_NOCLIENT		(1) Entity wird nicht an die Clients geschickt
SVF_DEADMONSTER		(2) Entity wird bei Kollisionserkennung als CONTENTS_DEADMONSTER (S. 127) behandelt, i.d.R. tote Monster- und Spielerkörper
SVF_MONSTER		(4) dto. als CONTENTS_MONSTER, i.d.R. ist das Entity ein Monster
vec3_t	mins	Ausdehnung der Bounding Box (S. 136) des Entities relativ zu seinem Ursprung ent→s.origin in Koordinaten des Welt-Koordinatensystems
vec3_t	maxs	
vec3_t	absmin	wie mins und maxs, nur relativ zum Welt-Ursprung vec3_origin
vec3_t	absmax	

vec3_t	size	Größe des Entities; Vektor von mins nach maxs
solid_t	solid	Solidity des Entities (Enumeration solid_t):

SOLID_NOT	(0) keine Interaktion mit anderen Objekten (i.d.R. Point Entity (S. 26))
SOLID_TRIGGER	(1) wird nur berührt, wenn etwas nach einer Bewegung im Inneren ist (Solid Entity oder Point Entity)
SOLID_BBOX	(2) Bounding Box; wird an den Ecken berührt (i.d.R. Point Entity, z.B. ein Model)
SOLID_BSP	(3) BSP-Brush; wird an den Ecken berührt (i.d.R. Solid Entity)

int	clipmask	Objekttypen, die das Entity nicht durchqueren kann (MASK_$*$, CONTENTS_$*$) (S. 127)
edict_t	*owner	Besitzer des Entities, falls vorhanden (z.B. eine Kugel gehört dem, der geschossen hat)

Hier endet die kurze Server-Version.

int	movetype	Bewegungsart des Entities (Enumeration movetype_t):

MOVETYPE_NONE	(0) bewegt sich nie
MOVETYPE_NOCLIP	(1) bewegt sich, ohne zu interagieren (bewegt sich durch andere Dinge hindurch)
MOVETYPE_PUSH	(2) bewegt sich, versucht bei Box-Kontakt zu schieben
MOVETYPE_STOP	(3) bewegt sich, bleibt stehen bei Box-Kontakt
MOVETYPE_WALK	(4) gehen, der Gravitation ausgesetzt (bei Spielern eingesetzt)
MOVETYPE_STEP	(5) der Gravitation ausgesetzt, spezielle Eckenbehandlung (bei Monstern eingesetzt)
MOVETYPE_FLY	(6) fliegen, keine Gravitation (nicht eingesetzt)
MOVETYPE_TOSS	(7) bewegt sich nicht von selbst, nur Gravitations- und Stoßbewegungen (Gegenstände)
MOVETYPE_FLYMISSILE	(8) Geschosse von Waffen, keine Gravitation
MOVETYPE_BOUNCE	(9) abprallende Gegenstände mit Gravitation (Granaten)

int	spawnflags	LE: Flags; Bits 1-8 können individuell für jedes Entity festgelegt werden, die folgenden sind jedoch fest definiert:

SPAWNFLAG_NOT_EASY	(256) Entity erscheint nicht in Schwierigkeitsstufe easy
SPAWNFLAG_NOT_MEDIUM	(512) dto. nicht in medium
SPAWNFLAG_NOT_HARD	(1024) dto. nicht in hard
SPAWNFLAG_NOT_DEATHMATCH	(2048) dto. nicht in DM
SPAWNFLAG_NOT_COOP	(4096) dto. nicht in Coop

char	*model	Dateiname des Models mit Pfad
float	freetime	Zeitpunkt, an dem das Entity gelöscht wurde
char	*message	LE: Nachricht an den Spieler (z.B. beim Benutzen eines Entities)
char	*classname	Klasse des Entities (z.B. player, monster_medic, func_door, ...)
int	flags	verschiedene Flags:
FL_FLY		(1) fliegend
FL_SWIM		(2) schwimmend, kann nicht ertrinken
FL_IMMUNE_LASER		(4) wird von Lasern nicht verletzt
FL_INWATER		(8) im Wasser
FL_GODMODE		(16) God-Modus aktiviert (nichts kann einem schaden)
FL_NOTARGET		(32) wird von Monstern nicht verfolgt
FL_IMMUNE_SLIME		(64) wird von Säure nicht verletzt
FL_IMMUNE_LAVA		(128) wird von Lava nicht verletzt
FL_PARTIALGROUND		(256) Boden unter dem Monster ist verschwunden
FL_WATERJUMP		(512) Spieler springt im Wasser
FL_TEAMSLAVE		(1024) nicht der erste im Team
FL_NO_KNOCKBACK		(2048) Entity erfährt bei Verletzung keinen Rückschlag
FL_POWER_ARMOR		(4096) Power Armor (Powerup) ist aktiviert
FL_RESPAWN		(8192) Entity wartet darauf, wieder eingesetzt/erzeugt zu werden (bei Items)
float	timestamp	Zeitstempel; von einigen Entities benutzt, um eine Zeitinformation zu speichern
float	angle	LE: YAW des Entities (im Bereich: 0° – 360°), teilweise auch als Bewegungsrichtung benutzt (dann zusätzlich: -1 = hoch und -2 = runter)
char	*target	LE: Name des Entities, auf das dieses Entity „zielt"; das Zielentity wird bei Bedarf benutzt: Zielentity→use (z.B. damit ein Knopf weiß, welche Tür er beim Betätigen öffnen soll)
char	*targetname	LE: Name des Entities, damit ein anderes auf es zielen kann
char	*killtarget	LE: wie target, nur wird dieses nicht benutzt, sondern aus dem Level entfernt
char	*team	LE: Name des Teams, zu dem dieses Entities gehört; alle Entities eines Teams werden in einer Kette zusammengehängt (⇒ teamchain, teammaster)
char	*pathtarget	LE: Name eines Entities; dient verschiedenen Entities auf verschiedene Weise, z.B. stellt es für die Plattformen den nächsten Wegpunkt dar
char	*deathtarget	LE: Name des Entities, das benutzt wird, wenn dieses Entity stirbt
char	*combattarget	LE: Name des Zielpunkt-Entities (Klasse: point_combat), welches von Monstern anvisiert wird

float	yaw_speed	Drehgeschwindigkeit, mit der sich das Monster Richtung ideal_yaw dreht
float	ideal_yaw	optimale YAW Winkelkomponente für Monster
edict_t	*target_ent	Zielentity (z.B. von turret oder func_train benutzt)
int	mass	Entitymasse; z.B. für Rückstoß bei Explosionen und beim Schieben durch andere Entities (Pusher)
float	gravity	individueller Gravitations-Multiplikator, um einzelne Entities leichter/schwerer zu machen (normal 1)
edict_t	*goalentity	das Entity, auf das sich das Monster gerade zubewegt
edict_t	*movetarget	das Entity, zu dem das Monster geht, solange es nichts Besseres zu tun hat
int	health	Gesundheitszustand des Entities; sollte ausschließlich von der Funktion T_Damage() verringert werden
int	max_health	maximale Gesundheit (i.d.R. der Anfangswert von ent→health)
int	gib_health	Gesundheitszustand, ab dem das Entity in Stücke zerfällt (i.d.R. negativ)
int	deadflag	Todesstatus:
DEAD_NO		(0) lebendig
DEAD_DYING		(1) sterbend
DEAD_DEAD		(2) tot
DEAD_RESPAWNABLE		(3) wiedereinsetzbar
qboolean	show_hostile	wenn true, zeigt dieses Monster allen anderen seinen Feind
edict_t	*mynoise	Geräusch-Entities, welche die Spieler erzeugen
edict_t	*mynoise2	
int	noise_index	Geräusch-Indizes, werden mit gi.soundindex() (S. 126) gesetzt, entsprechendes Geräusch kann dann mit gi.sound() bzw. gi.positioned_sound() abgespielt werden
int	noise_index2	
float	volume	Lautstärke und Reichweite obiger Geräusche beim Abspielen
float	attenuation	
int	sounds	LE: welches Geräusch das Entity von sich gibt
int	count	allgemeine Zählervariable (oft auch im LE gesetzt)
vec3_t	move_origin	Position und Orientierung des Geschützturms, falls der Spieler auf einem solchen stehen sollte
vec3_t	move_angles	
int	light_level	„Helligkeit“ des Spielers, damit Monster prüfen können, ob sie den Spieler sehen; z.B. kann der Spieler in einer dunklen Ecke für die Monster „unsichtbar“ sein
int	style	Stilparameter, z.B. für Items, Areaportal-Nummer, …

int	groundentity_linkcount	letzter linkcount des groundentity, um feststellen zu können, ob dieses sich bewegt hat
float	powerarmor_time	Ab diesem Zeitpunkt wird der durchsichtige Power Armor Schild nicht mehr angezeigt.
float	air_finished	Zu diesem Zeitpunkt geht dem Spieler unter Wasser die Luft aus, so daß er (erneut) Schaden nimmt.
float	touch_debounce_time	Zeitpunkt (während einer Berührung), an dem manche Funktionen frühestens wieder ausgeführt werden; sonst: zu oft hintereinander
float	pain_debounce_time	Zeitpunkt, an dem manche Geräusche frühestens wieder gespielt werden (z.B. bei Schmerzen)
float	damage_debounce_time	Zeitpunkt, an dem Lava/Säure dem Spieler erneut Schaden zufügen kann
float	fly_sound_debounce_time	Zeitpunkt, an dem ein Windstoßgeräusch frühestens wieder gespielt werden darf; nur bei Entity trigger_push benutzt
float	last_move_time	Zeitpunkt, an dem das Erdbebengeräusch frühestens wieder gespielt wird; nur bei Entity target_earthquake benutzt
float	teleport_time	Zeitpunkt an dem ein Spieler-Geräusch-Entity zuletzt ein Geräusch erzeugt hat
int	watertype	Art der Flüssigkeit, in der sich das Entity gerade befindet (Wasser, Lava, Säure) (gi.pointcontents() S. 127)
int	waterlevel	Eintauchtiefe in die obige Flüssigkeit:
0		außerhalb jeglicher Flüssigkeit
1		bis zu den Füßen
2		bis zum Hals
3		ganz unter der Oberfläche
int	takedamage	Zeigt an, ob das Entity beschädigt werden kann (Enumeration damage_t):
DAMAGE_NO		(0) kann nicht beschädigt werden
DAMAGE_YES		(1) kann beschädigt werden
DAMAGE_AIM		(2) dto. und zielsuchende Waffen erkennen dieses Entity
int	dmg	Schaden, den dieses Entity anderen zufügt
int	radius_dmg	Grundschaden, der innerhalb dmg_radius zugefügt wird
float	dmg_radius	Radius, innerhalb dessen dieser Schuß (Rakete, BFG) Schaden anrichtet

int	viewheight	Höhe über dem Entity-Ursprung, wo sich die Augen befinden
float	speed	LE: Geschwindigkeit (nur bei sich bewegenden Entities)
float	accel	LE: Beschleunigung
float	decel	LE: Verzögerung
vec3_t	movedir	Bewegungsrichtung
vec3_t	pos1, pos2	zwei ausgezeichnete Positionen der Entities (z.B. bei Knöpfen, Plattformen)
vec3_t	velocity	momentaner Bewegungsvektor des Entities; wird im Server-Frame in die Entitybewegung umgesetzt; um ein Entitity zu bewegen, setzt man i.d.R. diesen Parameter nach seinen Wünschen oder benutzt die Bewegungshilfsfunktionen, die dies für einen machen; er gibt an, wie weit in welche Richtung das Entity sich in 1 s bewegt
vec3_t	avelocity	momentaner Drehungsvektor des Entities; entsprechende Umsetzung
float	nextthink	Zeitpunkt an dem das Entity (außer bei Spielern) zum nächsten Mal seine think-Funktion aufruft
void	(*prethink) (edict_t *ent)	
		Funktion, die vor allen anderen Aktionen dieses Entities bei jedem Server-Frame ausgeführt wird (bei Spielern nicht) (optional, diese Funktion muß nicht für jedes Entity definiert sein)
void	(*think) (edict_t *self)	
		Funktion, die im Zeitpunkt nextthink ausgeführt wird (optional)
void	(*blocked) (edict_t *self, edict_t *other)	
		Funktion, die aufgerufen wird, wenn ein Entity (mit ent→movetype MOVETYPE_PUSH oder MOVETYPE_STOP) bei seiner Bewegung blockiert wird (optional)
void	(*touch) (edict_t *self, edict_t *other, cplane_t *plane, csurface_t *surf)	
		Funktion, die aufgerufen wird, wenn sich zwei Entities berühren (optional)
void	(*use) (edict_t *self, edict_t *other, edict_t *activator)	
		Funktion, die aufgerufen wird, wenn ein anderes Entity dieses Entity benutzt (i.d.R. gilt: self→targetname = other→target) (optional)
void	(*pain) (edict_t *self, edict_t *other, float kick, int damage)	
		Diese Funktion wird aufgerufen, wenn ein anderes Entity diesem Entity Schaden zufügt (optional).
void	(*die) (edict_t *self, edict_t *inflictor, edict_t *attacker, int damage, vec3_t point)	
		Diese Funktion wird aufgerufen, wenn der zugefügte Schaden größer war, als es das Entity verkraften konnte, das Entity also stirbt (wenn das Entity Schaden nehmen kann, muß diese Funktion existieren).

char	*map	Dateiname der nächsten Map für das Entity target_changelevel
float	wait	Wartezeit, z.B. bevor Türen wieder schließen
float	delay	andere Entities werden um delay Sekunden verzögert benutzt (G_UseTargets() S. 139)
float	random	LE: definiert die Größe des Zeitintervalls, innerhalb welchem die ent→wait-Zeit beim Entity func_timer zufällig schwanken kann
edict_t	*chain	Nachfolgeentity in einer Kette (nur bei Items in einem Team verwendet)
edict_t	*enemy	Feind, mit dem gerade gekämpft wird
edict_t	*oldenemy	letzter Feind
edict_t	*activator	Entity, das dieses Entity aktiviert hat
edict_t	*teamchain	nächstes Entity im Team
edict_t	*teammaster	erstes Entity im Team
edict_t	*groundentity	Entity, auf dem dieses Entity gerade steht
gitem_t	*item	Zeiger auf die Item-Informationen
moveinfo_t	moveinfo	Bewegungsinformationen von Entities
monsterinfo_t	monsterinfo	nähere Informationen für Monster

entity_state_t

int	number	Index des Entites in g_edicts
vec3_t	origin	Position des Entity-Ursprungs bzgl. Welt-Ursprung vec3_origin
vec3_t	angles	Orientierung des Entities im Raum (PITCH, YAW und ROLL)
vec3_t	old_origin	letzte Entity-Position
int	modelindex	Index des Models, mit gi.modelindex() (S. 126) gesetzt
int	modelindex2	weitere Indizes für Models, z.B. Waffe
int	modelindex3	
int	modelindex4	
int	skinnum	Nummer der Skin des Models (0 bis maximal 31)
int	frame	momentaner Entity-Frame; bei Models bezieht sich dieser auf die im Model definierten Frames (0 bis maximal 511), bei Brushes hingegen auf die verschiedenen Folgetexturen (diese Frames sind Entity-spezifisch und haben nichts mit Server-Frames oder Client-Frames zu tun!)
unsigned int	effects	Flags, die dem Entity folgende Effekte zuweisen:
EF_ROTATE		(0x00000001) rotiert um z-Achse
EF_GIB		(0x00000002) hinterläßt rote Partikel-Spur (rote Spur)

EF_BLASTER	(0x00000008) strahlt gelblich-grünes Licht aus (gelb-grünes Licht), orange Spur
EF_ROCKET	(0x00000010) gelb-grünes Licht, gelbe und graue Spur
EF_GRENADE	(0x00000020) feine graue Spur
EF_HYPERBLASTER	(0x00000040) gelb-grünes Licht
EF_BFG	(0x00000080) grünes Licht
EF_COLOR_SHELL	(0x00000100) aktiviert Transparenz der Hüllen bei RF_COLOR_∗-Flags (s.u.)
EF_POWERSCREEN	(0x00000200)) grüne transparente Wand vor dem Entity
EF_ANIM01	(0x00000400) Jeden fünften Server-Frame wird zwischen den Entity-Frames 0 und 1 hin- und hergeschaltet.
EF_ANIM23	(0x00000800) dto. zwischen den Entity-Frames 2 und 3
EF_ANIM_ALL	(0x00001000) Jeden fünften Server-Frame wird der Entity-Frame um eins erhöht (nur bei Brushes möglich, da bei Models der letzte Frame nicht ausgezeichnet ist und der Server somit über das Ende der Model-Animation hinausläuft).
EF_ANIM_ALLFAST	(0x00002000) dto., aber Inkrementierung des Entity-Frames jeden Server-Frame
EF_FLIES	(0x00004000) schwarzer Partikel-Schwarm um Objekt
EF_QUAD	(0x00008000) blaue transparente Hülle um das Objekt herum
EF_PENT	(0x00010000) rote transparente Hülle um das Objekt herum
EF_TELEPORTER	(0x00020000) gelbe Partikel-Fontaine
EF_FLAG1	(0x00040000) rote Partikel-Fontaine
EF_FLAG2	(0x00080000) blaue Partikel-Fontaine
EF_IONRIPPER	(0x00100000) rot-weißes Licht, rote Spur
EF_GREENGIB	(0x00200000) gelb-orange-grüne Spur
EF_BLUEHYPERBLASTER	(0x00400000) blaues Licht
EF_SPINNINGLIGHTS	(0x00800000) rotiert um Z-Achse und leuchtet in eine Richtung rot (rotes Drehlicht)
EF_PLASMA	(0x01000000) rot-weißes Licht
EF_TRAP	(0x02000000) gelb-grünes pulsierendes Licht, orange Spur
EF_TRACKER	(0x04000000) „schwarzes“ Licht, schwarzer „Schweif“

EF_DOUBLE	(0x08000000) gelbe transparente Hülle um das Objekt
EF_SPHERETRANS	(0x10000000) Objekt ist transparent
EF_TAGTRAIL	(0x20000000) gelb-grünes Licht, gelbe Spur
EF_HALF_DAMAGE	(0x40000000) graue transparente Hülle um das Objekt herum
EF_TRACKERTRAIL	(0x80000000) schwarze Partikelkugel hinter Objekt

Diese Effekte werden in den Clients bearbeitet und haben permanenten Einfluß auf dieses Entity; ein Entity mit Effekten wird immer an die Clients geschickt, sogar wenn ent→s.modelindex = 0 ist.

int renderfx verschiedene Darstellungseffekte:

RF_MINLIGHT	(1) Dieses Entity gibt immer ein wenig Licht ab; es kann immer gesehen werden, sogar bei Dunkelheit.
RF_VIEWERMODEL	(2) Dieses Entity kann dieser Spieler nicht direkt sehen, sondern z.B. nur durch einen Spiegel (macht nur Sinn, wenn das Entity ein Spieler-Entity ist).
RF_WEAPONMODEL	(4) Dieses Entity wird nur von diesem Spieler gesehen (z.B. die ausgewählte Waffe).
RF_FULLBRIGHT	(8) Dieses Entity wird immer mit maximaler Lichtintensität dargestellt.
RF_DEPTHHACK	(16) Effekt für die aktivierte Waffe („view weapon Z crunching“)
RF_TRANSLUCENT	(32) Objekt ist transparent
RF_FRAMELERP	(64) ? wird nur bei Monstern benutzt
RF_BEAM	(128) Laser-Strahl-Effekt; ein Laser der Farbe ent→s.skinnum und der Breite ent→s.frame wird von ent→s.old_origin nach ent→s.origin gezeichnet (hexadezimale Angabe von vier Farben der Farbpalette, die für den Laser benutzt werden, z.B. vier Orangetöne 0xe0e1e2e3 oder nur ein einziger Blauton 0xb0b0b0b0, wobei eine Farbe – bei 256 möglichen – durch jeweils zwei Stellen der Zahl bestimmt ist)
RF_CUSTOMSKIN	(256) ent→s.skinnum bezeichnet den Index eines mit Hilfe gi.imageindex() (S. 126) bereits gecachten Bildes. Dies erlaubt (theoretisch) dynamisch erzeugte Skins auf den Models; anscheinend ist dies jedoch recht langsam und macht darüberhinaus den Server instabil; wird auch im game dll Code nirgendwo benutzt.
RF_GLOW	(512) Pulsierender Lichteffekt, für Items benutzt
RF_SHELL_RED	(1024) rote Hülle um Entity; die Hüllen können auch kombiniert werden und sind erst dann transparent, wenn EF_COLOR_SHELL gesetzt ist

RF_SHELL_GREEN	(2048) grüne Hülle
RF_SHELL_BLUE	(4096) blaue Hülle
RF_IR_VISIBLE	(32768) Objekt ist mit Hilfe eines Infrarot-Sichtgerätes auch im Dunkeln sichtbar (siehe auch RDF_IRGOOGLES S. 101)
RF_SHELL_DOUBLE	(65536) gelbe Hülle
RF_SHELL_HALF	(131072) graue Hülle
RF_USE_DISGUISE	(262144) der Spieler benutzt eine Verkleidung und wird von den Monstern nicht erkannt

int	solid	für Client-Vorhersage; wird von gi.linkentity() gesetzt
int	sound	(sich wiederholendes) Geräusch, das das Entity abgibt
int	event	Dies sind einmalige Ereignisse; sie finden in einem Frame statt, dann wird ent→s.event wieder zurückgesetzt. Es existieren folgende Möglichkeiten der Enumeration entity_event_t:

EV_NONE	(0) nichts
EV_ITEM_RESPAWN	(1) Geräusch beim Wiedererscheinen von Items
EV_FOOTSTEP	(2) Gehgeräusch
EV_FALLSHORT	(3) Geräusch beim Fallen, kurze Stecke
EV_FALL	(4) dto. mittlere Strecke
EV_FALLFAR	(5) dto. lange Strecke
EV_PLAYER_TELEPORT	(6) Geräusch, wenn ein Lebewesen teleportiert wird
EV_OTHER_TELEPORT	(7) Geräusch, wenn Gegenstände teleportiert werden (z.B. bei Entity train)

gclient_t

player_state_t	ps	Spieler Statusinformationen, welche für das Rendering einer Sicht von den Clients benötigt werden; wird vom Server jeden Server-Frame an die Clients übermittelt.
int	ping	Ping

Hier endet die kurze Server-Version.

client_persistant_t	pers	Daten, die über die Levels hinweg erhalten bleiben
client_respawn_t	resp	Daten, die beim Wiedereinsetzen nach dem Tod im DM erhalten bleiben
pmove_state_t	old_pmove	Zwischenspeicher des letzten client→ps.pmove
qboolean	showscores	Wenn auf true gesetzt, wird das Scoreboard (Punktestand im DM) angezeigt (⇒ STAT_LAYOUTS 1. Bit gesetzt).
qboolean	showinventory	Wenn auf true gesetzt, wird das Inventar angezeigt (⇒ STAT_LAYOUTS 2. Bit gesetzt).
qboolean	showhelpicon	unbenutzt

qboolean	showhelp	Wenn auf true gesetzt, wird die Hilfeanzeige zur Mission eingeblendet (⇒ STAT_LAYOUTS 1. Bit gesetzt; nicht im DM).
int	ammo_index	Index der Munition für die aktivierte Waffe (in der Itemliste)
int	buttons	die vom Spieler momentan gedrückten Knöpfe (siehe Client-Frame: ucmd→buttons S. 147)
int	oldbuttons	die zuletzt gedrückten Knöpfe
int	latched_buttons	die gerade neu gedrückten Knöpfe
qboolean	weapon_thunk	Anzeige, ob Waffenframe schon ausgeführt wurde
gitem_t	*newweapon	Diese Waffe wird gerade aktiviert.
int	damage_armor	Teilschäden dieses Frames
int	damage_parmor	
int	damage_blood	
int	damage_knockback	
vec3_t	damage_from	Richtung, in die der Rückstoß aufgrund des Schadens wirkt
float	killer_yaw	YAW auf eigenen Mörder, damit man diesen nach dem Tod ansieht
weaponstate_t	weaponstate	Waffenzustand:
WEAPON_READY		(0) bereit
WEAPON_ACTIVATING		(1) wird aktiviert
WEAPON_DROPPING		(2) wird deaktiviert
WEAPON_FIRING		(3) schießt gerade
vec3_t	kick_angles	Winkel und Richtung des Waffenrückstoßes
vec3_t	kick_origin	
float	v_dmg_roll	für Positionsänderungen der Sicht bei Schaden bzw. beim Fallen
float	v_dmg_pitch	
float	v_dmg_time	
float	fall_time	
float	fall_value	
float	damage_alpha	Durchsichtigkeit (Alpha) des Farbschleiers bei Schaden bzw. beim Aufnehmen von Items
float	bonus_alpha	
vec3_t	damage_blend	Farbe (RGB) des Farbschleiers
vec3_t	v_angle	Sicht- und Schußrichtung
float	bobtime	Ende des nächsten Schrittes, damit EV_FOOTSTEP synchron zur Bewegung ist
vec3_t	oldviewangles	Zwischenspeicher der letzten Werte von client→ps.viewangles bzw. ent→velocity
vec3_t	oldvelocity	

float	next_drown_time	nächster Zeitpunkt an dem die Luft (unter Wasser) ausgeht und ein Schaden entsteht
int	old_waterlevel	Zwischenspeicher des letzten ent→waterlevel
int	breather_sound	für Breather-Geräusch abwechselnd 1 und 0 (ein- und ausatmen)
int	machinegun_shots	Anzahl der Maschinenpistolenschüsse, damit die Waffe bei längerem Schießen nach oben gerissen wird
int	anim_end	letzter Frame der Spielermodel Animation
int	anim_priority	Animationspriorität:
ANIM_BASIC		(0) stehen, gehen
ANIM_WAVE		(1) winken
ANIM_JUMP		(2) springen
ANIM_PAIN		(3) Schmerzen erleiden
ANIM_ATTACK		(4) angreifen
ANIM_DEATH		(5) sterben
ANIM_REVERSE		(6) Frames laufen rückwärts ab
qboolean	anim_duck	wenn true, Animation in geducktem Zustand
qboolean	anim_run	wenn true, Animation in rennendem Zustand
float	quad_framenum	Framenummern bis wann die jeweiligen Powerups ihre Wirkung haben
float	invicible_framenum	
float	breather_framenum	
float	enviro_framenum	
qboolean	grenade_blew_up	wenn true, ist die Granate (dummerweise) schon in der Hand losgegangen
float	grenade_time	Zeitpunkt, an dem die Granate explodiert
int	silencer_shots	Anzahl der verbleibenden Schüsse mit dem Schalldämpfer
int	weapon_sound	Index des Geräusches, das die Waffe von sich gibt
float	pickup_msg_time	Zeitpunkt, bis zu welchem die Nachricht beim Aufnehmen eines Items noch eingeblendet wird
float	flood_locktill	Variablen zur Chat-Nachrichten-Flußkontrolle
float	flood_when[10]	
int	flood_whenhead	
float	respawn_time	Zeitpunkt, ab dem der Spieler frühestens wieder eingesetzt werden kann
edict_t	*chase_target	die Kamera folgt gerade diesem Spieler (im Zuschauermodus)
qboolean	update_chase	zeigt an, ob die Informationen bezüglich der Kameraführung aufgefrischt werden müssen

player_state_t

Diese Struktur wird jeden Server-Frame an den entsprechenden Client übermittelt.

pmove_state_t	pmove	Bewegungsstatus des Spielers, wird von den Clients für die Vorhersage benötigt; pmove wird zusätzlich jeden Client-Frame an den betreffenden Client geschickt; damit die Client-Vorhersage synchron bleibt, werden hier keine float-Typen verwendet (siehe auch gi.Pmove() S. 122)
vec3_t	viewangles	Orientierung des Spielers bezüglich des Welt-Koordinatensystems, mit Hilfe derer die Sichtrichtung des Spielers festgelegt wird
vec3_t	viewoffset	Offset der Sichtposition; wird zu client→ps.pmove.origin addiert, um die Sichthöhe zu bestimmen (Augenhöhe entspricht ja nicht unbedingt dem Ursprung des Models)
vec3_t	kick_angles	Orientierungsoffset; wird zu der Sichtrichtung addiert, um den endgültigen Rendering-Winkel zu erhalten; wird vom Waffen-Rückstoß oder von Schadens-Effekten gesetzt
vec3_t	gunangles	Die Orientierung der Waffe
vec3_t	gunoffset	Der Positionsoffset der Waffe bzgl. der Spieler-Position
int	gunindex	Model-Index der Waffe
int	gunframe	Entity-Frame der Waffe
float	blend[4]	Farbschleier, der letztendlich auf dem Bildschirm zu sehen ist; Werte entsprechen RGBA-Darstellung (Rot, Grün, Blau, Alpha – Durchsichtigkeit: 0 kein Effekt, 1 nur noch einfarbiger Bildschirm); z.B. bei Invulnerability, beim Aufnehmen von Items, …
float	fov	Field of View, horizontale Ausdehnung des Sichtfelds des Spielers in Grad; Standard: 90, aber mit Cvar fov von 1 bis 160 einstellbar.
int	rdflags	Rendering-Flags:
RDF_UNDERWATER		(1) Unterwassereffekt
RDF_NOWORLDMODEL		(2) für Spieler-Konfigurationsanzeige benutzt (nur Entities werden dargestellt)
RDF_IRGOOGLES		(4) Spieler hat Infrarot-Sichtgerät, d.h. er sieht im Dunkeln noch alle Objekte mit RF_IR_VISIBLE (S. 97)
RDF_UVGOOGLES		(8) nicht benutzt

short stats[MAX_STATS]	Statusvariablen mit folgenden Indizes:
STAT_HEALTH_ICON	(0) Index des Erste-Hilfe-Icons (gi.imageindex() S. 126)
STAT_HEALTH	(1) Gesundheitszustand des Spielers (ent→health)
STAT_AMMO_ICON	(2) Index des Munitions-Icons
STAT_AMMO	(3) verbleibende Munition
STAT_ARMOR_ICON	(4) Index des Rüstungs-Icons
STAT_ARMOR	(5) verbleibende Rüstungspunkte
STAT_SELECTED_ICON	(6) Icon des ausgewählten Items
STAT_PICKUP_ICON	(7) Icon des aufgenommenen Items
STAT_PICKUP_STRING	(8) Nummer des Configstrings (S. 124), welcher die Nachricht enthält, die beim Aufnehmen des Items ausgebeben wird (CS_ITEMS+x)
STAT_TIMER_ICON	(9) Index des Icons des laufenden Timers (bei Quad Damage, Invulnerability, . . .)
STAT_TIMER	(10) Restzeit des laufenden Timers
STAT_HELPICON	(11) Index des Hilfe-Icons
STAT_SELECTED_ITEM	(12) Index des ausgewählten Items in der itemlist
STAT_LAYOUTS	(13) Layout-Status (S. 114)
STAT_FRAGS	(14) Anzahl der Frags (ent→client→resp.score)
STAT_FLASHES	(15) Flags zum Aufleuchten des Hintergrundes unter Status-Nummern vom Gesundheits- (1) und Rüstungszustand (2)
STAT_CHASE	(16) Nummer des Configstrings, der die Skin des Spielers, dem man folgt, beinhaltet (CS_PLAYERSKINS+x)
STAT_SPECTATOR	(17) 1, wenn im Zuschauermodus, sonst 0
MAX_STATS	(32) maximale Anzahl an Statusvariablen

Statusbar

Hier bietet es sich an, die Funktionsweise der Statusbar näher zu erläutern. Die Statusbar ist eine Grafik, die mit Hilfe eines Statusbarprogramms erzeugt und über die 3D-Grafik des Spiels gelegt wird. Sie wird von der game dll während des Spiels mit Hilfe obiger Statusvariablen jedes Clients gesteuert, nachdem zu Beginn des Levels das Statusbarprogramm über den Configstring CS_STATUSBAR allen Clients mitgeteilt wurde (dieser Configstring darf hier ausnahmsweise größer als 64 Bytes sein, bei ungefähr 1280 Bytes ist aber auch hier die Grenze erreicht). Im folgenden ist [stat] ein Platzhalter für den Index einer obigen Statusvariable (den Integerwert benutzen, nicht den Konstantenname!), [zahl] steht für einen Zahlenwert, [text] für einen beliebigen Text und [datei] für einen Dateinamen ohne die Erweiterung .pcx, jedoch mit Pfad ab dem `pics`-Verzeichnis, falls sie nicht in diesem Verzeichnis liegt. Der Aufbau ähnelt dem eines Layouts (S. 114); dort sind auch einige grundlegende Dinge zur Grafikdarstellung beschrieben. Ein Statusbarprogramm besteht aus einfachen Kontrollanweisungen (diese können anscheinend nicht geschachtelt werden):

if␣[stat]␣	Das Programm wird bis zur nächsten endif-Anweisung ausgeführt, falls die Statusvariable mit Index [stat] ungleich 0 ist.
endif␣	endif-Anweisung
ifeq␣[stat]␣[zahl]␣	? steht so im Quelltext als Erklärung zum Statusbarprogramm, soll wohl bedeuten: wie if, falls die Statusvariable gleich [zahl] ist; zeigte aber in bisherigen Tests keinerlei Funktion!?
ifbit␣[stat]␣[zahl]␣	? hier gilt das selbe wie bei ifeq, nur daß dies wohl bedeuten soll: wie if, falls die Statusvariable bitweise und mit [zahl] verknüpft ungleich 0 ist?

... aus Positionsangaben:

xl␣[zahl]␣	alles Folgende wird in x-Richtung (von links nach rechts) [zahl] Pixel vom linken Bildschirmrand entfernt gezeichnet ([zahl] ist i.d.R. positiv)
xr␣[zahl]␣	dto. vom rechten Bildschirmrand entfernt ([zahl] ist i.d.R. negativ)
xv␣[zahl]␣	dto. von der „Bildmitte“
yb␣[zahl]␣	Alles Folgende wird in y-Richtung (von oben nach unten) [zahl] Pixel vom unteren Bildschirmrand entfernt gezeichnet ([zahl] ist i.d.R. negativ)
yt␣[zahl]␣	dto. vom oberen Bildrand entfernt ([zahl] ist i.d.R. positiv)
yv␣[zahl]␣	dto. von der „Bildmitte“

... und der darzustellenden Grafik:

pic␣[stat]␣	das Bild, dessen Index in der Statusvariablen [stat] gespeichert ist
picn␣\"[datei]\"␣	das Bild aus der Datei [datei].pcx
statpic␣\"[datei]\"␣	? steht so im Quelltext als Erklärung zum Statusbarprogramm, soll wohl das selbe wie picn bewirken, zeigte aber in bisherigen Tests keine Funktion!?
num␣[zahl]␣[stat]␣	Der Wert aus der Statusvariablen [stat] wird durch eine Nummer dargestellt, die [zahl] viele Stellen bzw. Ziffern hat, die jeweils 16 Pixel breit und 24 Pixel hoch sind.
stat_string␣[stat]␣	Der Configstring, dessen Nummer in der Statusvariablen [stat] gespeichert ist, wird in weißer Schrift dargestellt, wobei jeder Buchstabe 8 Pixel hoch und breit ist.
string␣\"[text]\"␣	Der Text [text] wird in weißer Schrift dargestellt.
string2␣\"[text]\"␣	Der Text [text] wird in grüner Schrift dargestellt.

cstring␣\"[text]\"␣	Der Text [text] wird im Bild zentriert und erst dann im x-Wert verschoben in weißer Schrift dargestellt.
cstring2␣\"[text]\"␣	Der Text [text] wird im Bild zentriert und erst dann im x-Wert verschoben in grüner Schrift dargestellt.
hnum␣ anum␣ rnum␣	Diese stellen die speziellen Werte der health, ammo und armor Statusvariablen dar.

Wie man sehen kann, scheinen die Erläuterungen im Quelltext zu diesem Thema nicht ganz korrekt, geschweige denn vollständig zu sein. Deshalb ist es gut denkbar, daß (dies gilt im übrigen auch für das Layout) noch weitere verwendbare Befehle existieren; wenn überhaupt wird man diese wohl nur durch Zufall entdecken. Aber mit den oben stehenden (und funktionierenden) Befehlen kann man schon eine Menge erreichen.

client_persistant_t		
char	userinfo[MAX_INFO_STRING]	Aneinanderreihung der Namen (Schlüssel) und Werte aller Client-Cvars mit dem Flag CVAR_USERINFO
char	netname[16]	Spielername im Netz
int	hand	Waffenposition im Sichtfeld des Spielers:
RIGHT_HANDED		(0) rechte Seite
LEFT_HANDED		(1) linke Seite
CENTER_HANDED		(2) nicht sichtbar (in der Mitte)
qboolean	connected	zeigt an, ob diese Client-Struktur mit einem echten Client verbunden ist (möglich, daß z.B. bei einem geladenen Spiel nicht alle vorhandenen benutzt werden)
int int int int int	health max_health savedFlags selected_item inventory[MAX_ITEMS]	Werte, die beim Levelwechsel hier zwischengespeichert werden
int int int int int int	max_bullets max_shells max_rockets max_grenades max_cells max_slugs	Maximale Munitionskapazitäten
gitem_t	*weapon	Zeiger auf die aktivierte Waffe
gitem_t	*lastweapon	Zeiger auf die zuletzt gewählte Waffe

int	power_cubes	für die Power Cubes im Coop benutzt
int	score	verwendet, um die Punkte eines kompletten Levels zu berechnen (z.B. Anzahl erreichter Ziele, Frags, ...)
int	game_helpchanged	Zwischenspeicher für game.helpchanged, um festzustellen, ob sich die Hilfe geändert hat
int	helpchanged	Zähler für die Anzahl der bisherigen Benachrichtigungen, daß sich die Hilfe geändert hat
qboolean	spectator	Wird beim Wechsel in den Zuschauermodus (Spielermodus) auf true (false) gesetzt.

gitem_t

char	*classname	Name bei der Erzeugung des Items
qboolean	(*pickup)(edict_t *ent, gitem_t *item)	Funktion, die beim Aufnehmen des Items aufgerufen wird
void	(*use)(edict_t *ent, gitem_t *item)	Funktion, die beim Benutzen aufgerufen wird
void	(*drop)(edict_t *ent, gitem_t *item)	Funktion, die beim Fallenlassen aufgerufen wird
void	(*weaponthink)(editc_t *ent)	Funktion, die einmal innerhalb der Zeit eines Server-Frames (FRAMETIME) aufgerufen wird, um die Waffenanimation zu steuern (beim Schießen schon in einem Client-Frame, oder, wenn dort nicht geschehen, spätestens im Server-Frame)
char	*pickup_sound	Datei mit dem Geräusch, das beim Aufnehmen des Items abgegeben werden soll (mit Angabe der Dateiendung .wav und des Dateipfads ab dem Verzeichnis sound)
char	*world_model	Datei des Welt-Models des Items; also das Model, welches jeder sehen kann, falls das Item irgendwo in der Welt „herumliegt“ (mit Angabe der Dateiendung .md2 und des Dateipfades ab dem Verzeichnis models)
int	world_model_flags	Die Flags, die in ent→s.effects eingetragen werden
char	*view_model	Datei des Sicht-Models der Waffe; also das Model, welches nur der Spieler sieht, wenn er die Waffe ausgewählt hat (Dateiangaben wie bei world_model)
char	*icon	Datei mit dem Icon der Waffe (ohne Angabe der Dateiendung .pcx und mit Angabe des Pfades ab dem Verzeichnis pics)

char	*pickup_name	Der Name, der beim Aufnehmen als Nachricht in der Statusbar erscheint und zum Auffinden von Items mit Hilfe von FindItem() (S. 133) benutzt wird.
int	count_width	Anzahl an Ziffern, die neben dem Icon dargestellt werden sollen, wenn das Item ausgewählt ist
int	quantity	Anzahl an Munition, die bei einem Schuß mit dieser Waffe benötigt wird bzw. Anzahl an Munition, die dieses Item bereitstellt
char	*ammo	Art der benutzten Munition dieser Waffe, durch deren pickup_name spezifiziert
int	flags	Gruppenzugehörigkeitsflags des Items:
IT_WEAPON		(1) Waffe
IT_AMMO		(2) Munition
IT_ARMOR		(4) Rüstung
IT_STAY_COOP		(8) Items, die im Coop beim Aufnehmen an ihrem Platz für weitere Spieler erhalten bleiben
IT_KEY		(16) Schlüssel
IT_POWERUP		(32) Powerup
int	weapmodel	für das Setzen des Waffen-Model-Indexes:
WEAP_BLASTER		(1) Blaster
WEAP_SHOTGUN		(2) Shotgun
WEAP_SUPERSHOTGUN		(3) Supershotgun
WEAP_MACHINEGUN		(4) Machinegun
WEAP_CHAINGUN		(5) Chaingun
WEAP_GRENADES		(6) Grenades
WEAP_GRENADELAUNCHER		(7) Grenadelauncher
WEAP_ROCKETLAUNCHER		(8) Rocketlauncher
WEAP_HYPERBLASTER		(9) Hyperblaster
WEAP_RAILGUN		(10) Railgun
WEAP_BFG		(11) BFG
void	*info	Zeiger auf weitere Informationen bezüglich der Items; nur bei einigen Rüstungstypen benutzt als Zeiger auf die jeweiligen Strukturen des Typs gitem_armor_t
int	tag	Wert zur schnellen Überprüfung der Itemklasse:
AMMO_BULLETS		(0) Bullets
AMMO_SHELLS		(1) Shells
AMMO_ROCKETS		(2) Rockets
AMMO_GRENADES		(3) Grenades
AMMO_CELLS		(4) Cells
AMMO_SLUGS		(5) Slugs

ARMOR_NONE	(0) keine Rüstung
ARMOR_JACKET	(1) Jacket Rüstung
ARMOR_COMBAT	(2) Combat Rüstung
ARMOR_BODY	(3) Body Rüstung
ARMOR_SHARD	(4) Shard Rüstung

char	*precaches	Aneinanderreihung aller Models, Geräusche und Bilder in einem String (untereinander durch Freizeichen getrennt, Endungen und Pfade wie jeweils oben), die dieses Item benutzen wird, um diese beim Start eines Levels mit den gi.*index()-Funktionen zu laden

Für Items existieren neben den SPAWNFLAG_NOT_∗-Flags (die nur bis kurz vor dem Erzeugen vorhanden sind) noch spezielle ent→spawnflags:

ITEM_TRIGGER_SPAWN	(1) Item ist zu Beginn in der Welt nicht vorhanden (SVF_NOCLIENT); wenn es benutzt wird, erscheint es dann in der Welt
ITEM_NO_TOUCH	(2) Item kann nicht berührt werden, rotiert nicht und strahlt kein Licht ab (wie sonst jedes Item)
...	6 Bit sind für Level-Editor reserviert
...	8 Bit für die Power Cubes Identität im Coop
DROPPED_ITEM	(65536) Item wurde von früherem Besitzer fallen gelassen
DROPPED_PLAYER_ITEM	(131072) Item wurde im DM fallengelassen, weil der Spieler getötet wurde
ITEM_TARGETS_USED	(262144) Beim Berühren eines Items werden die angegebenen Targets benutzt und danach dieses Flag gesetzt, damit dies bei nochmaliger Berührung nicht noch einmal passiert.

gitem_armor_t

int	base_count	Grund-Rüstungspunkte dieser Rüstung
int	max_count	maximale Rüstungspunkte, wenn diese Rüstung getragen wird (neue Rüstungspunkte werden in Abhängigkeit der noch vorhandenen Rüstungspunkte erhöht, so daß mehr als base_count möglich wird)
float	normal_protection	Schutz vor „normalen“ Waffen
float	energy_protection	Schutz vor energie-basierten Waffen
int	armor	Rüstungsart, eine der ARMOR_∗ Konstanten

client_respawn_t

client_persistant_t	coop_respawn	diese Struktur dient im Coop als client→pers, wenn der Spieler wieder erzeugt wird
int	enterframe	level.framenum, an dem der Client das Spiel betreten hat
int	score	momentane Punkte (Frags, erreichte Ziele, ...)
vec3_t	cmd_angles	die ucmd→angles des letzten Client-Frames
qboolean	spectator	wenn true, ist der Spieler im Zuschauermodus

moveinfo_t

Fixe Daten, die sich während einer Bewegung nicht ändern:

vec3_t	start_origin	Start- bzw. Endposition und -orientierung
vec3_t	start_angles	
vec3_t	end_origin	
vec3_t	end_angles	
int	sound_start	Geräusche zu Beginn, während und am Ende der Bewegung
int	sound_middle	
int	sound_end	
float	accel	Beschleunigung
float	speed	Geschwindigkeit
float	decel	Verzögerung
float	distance	Distanz
float	wait	Wartezeit

Zustandsdaten, die sich im Verlauf der Bewegung ändern können:

int	state	Bewegungsstatus (z.B. STATE_UP, STATE_TOP bei Plattformen und Türen)
vec3_t	dir	Bewegungsrichtung (normiert)
float	current_speed	momentane Geschwindigkeit
float	move_speed	Bewegungsgeschwindigkeit
float	next_speed	nächste Geschwindigkeit
float	remaining_distance	verbleibender Weg
float	decel_distance	Verzögerungsweg
void	(*endfunc)(edict_t *ent)	Funktion, die am Ende der Bewegung ausgeführt werden soll

monsterinfo_t		
mmove_t	*currentmove	beschreibt den momentanen Bewegungszyklus
int	aiflags	Flags zur Steuerung der künstlichen Intelligenz des Monsters:

AI_STAND_GROUND	(1) stehen bleiben
AI_TEMP_STAND_GROUND	(2) kurz stehen bleiben
AI_SOUND_TARGET	(4) suche nach einer Geräuschquelle
AI_LOST_SIGHT	(8) Feind aus den Augen verloren
AI_PURSUIT_LAST_SEEN	(16) dem zuletzt gesehenen Feind folgen
AI_PURSUE_NEXT	(32) dem Feind zum nächsten Ziel folgen
AI_PURSUE_TEMP	(64) noch eine kleine Weile weiterverfolgen
AI_HOLD_FRAME	(128) Entity-Frame bleibt gleich (z.B. beim Schießen mit der Maschinenpistole)
AI_GOOD_GUY	(256) Monster greift niemanden an (z.B. Gefangener)
AI_BRUTAL	(512) Angreifen, bis ent→health des Feindes ≤ -80 (sonst bis ≤ 0)
AI_NOSTEP	(1024) von Entity exploding_box benutzt
AI_DUCKED	(2048) ducken
AI_COMBAT_POINT	(4096) bis zum Erreichen des Zielpunktes wird alles andere ignoriert
AI_MEDIC	(8192) andere Monster „heilen“ (nur bei Monster Medic eingesetzt)
AI_RESURRECTING	(16384) Wiederauferstehen des Monsters (nach Heilung durch Medic)

int	nextframe	nächster Entity-Frame
float	scale	Faktor, um Bewegungsgeschwindigkeit der einzelnen Monster anzupassen

Die Funktion wird aufgerufen, …

void	(*stand)(edict_t *self)	
		… wenn das Monster nur herumstehen soll (z.B. Ausgabe spezifischer Geräusche)
void	(*idle)(edict_t *self)	
		… um das Herumstehen nicht zu langweilig werden zu lassen (optional, muß nicht jedes Monster haben)
void	(*search)(edict_t *self)	
		… wenn nach Feinden gesucht wird (optional)
void	(*walk)(edict_t *self)	
		… wenn das Monster zu seinem Zielpunkt gehen soll

void	(*run)(edict_t *self)	
		. . . wenn das Monster einen Feind gesichtet hat und sich auf den Angriff vorbereiten soll
void	(*dodge)(edict_t *self, edict_t *other, float eta)	
		. . . um sich zu ducken, wenn das Monster angegriffen wird (optional)
void	(*attack)(edict_t *self)	
		. . . um einen Fernkampf zu beginnen (optional)
void	(*melee)(edict_t *self)	
		. . . um einen Nahkampf beginnt (optional)
void	(*sight)(ecict_t *self, edict_t *other)	
		. . . wenn das Monster gerade einen Feind gesichtet hat (optional)
qboolean	(*checkattack)(edict_t *self)	
		. . . um zu testen, ob das Monster mit dem Angriff beginnen soll
float	pausetime	Bis zu diesem Zeitpunkt macht das Monster eine „Pause“.
float	attack_finished	An diesem Zeitpunkt wird der momentane Angriff voraussichtlich zu Ende sein.
vec3_t	saved_goal	Zielposition, wenn diese nicht im selben Server-Frame erreichbar ist
float	search_time	Bis zu diesem Zeitpunkt wird nach einem Feind gesucht.
float	trail_time	An diesem Zeitpunkt wurde der Feind (oder sein „Trail“) zuletzt gesehen.
vec3_t	last_sighting	letzte Position, an der der Feind gesehen wurde
int	attack_state	Angriffsmodus des Monsters:
AS_STRAIGHT		(1) geradeaus bis zu einer guten Schußposition laufen
AS_SLIDING		(2) Seitwärts bis zu einer guten Schußposition laufen
AS_MELEE		(3) Nahangrifff
AS_MISSILE		(4) Fernangriff
int	lefty	Entscheidung, ob das Monster beim Angriff im Modus AS_SLIDING nach rechts oder links geht; wechselt immer zwischen 0 und 1
float	idle_time	An diesem Zeitpunkt ruft das Monster, sofern es steht ai_stand() (geht ai_walk()), die idle (search)-Funktion wieder auf
int	linkcount	für die Überprüfung, ob sich das Monster noch auf dem Boden befindet, wenn es sich zuvor bewegt hat

int power_armor_type		Art der Power-Rüstung, die das Monster hat:
POWER_ARMOR_NONE	(0) keine	
POWER_ARMOR_SCREEN	(1) Power Screen Rüstung	
POWER_ARMOR_SHIELD	(2) Power Shield Rüstung	
int power_armor_power		Restenergie für die Power-Rüstung (bei Spielern werden AMMO_CELLS benötigt, um eine Power-Rüstung benutzen zu können)

mmove_t

int	firstframe	erster Frame dieses Bewegungszyklusses
int	lastframe	letzter Frame dieses Bewegungszyklusses
mframe_t	*frame	Zeiger auf den momentanen Bewegungszyklus (jedes Monster hat mehrere Listen dieses Typs)
void	(*endfunc)(edict_t *self)	Funktion, die am Ende dieses Bewegungszyklusses ausgeführt werden soll

mframe_t

void (*aifunc)(edict_t *self, float dist)

Funktion der künstlichen Intelligenz, die in diesem Frame aufgerufen wird (gilt AI_HOLD_FRAME ist Parameter dist = 0) (Parameter der folgenden Funktionen entsprechen denen dieses Funktionszeigers):

ai_stand()	herumstehen und nach Feinden Ausschau halten, dist ist für die Anpassung der Sicht-Position in der Animation
ai_move()	dist weit in die momentane Richtung gehen
ai_walk()	dist weit in Richtung des Ziels gehen, dabei nach Feinden Ausschau halten
ai_turn()	Richtung ent→ideal_yaw drehen, nach Feinden Ausschau halten, dist wie bei ai_stand()
ai_run()	gefundenen Feind verfolgen und versuchen zu töten
ai_charge()	wieder in Richtung Ziel drehen und fortfahren

float dist Parameter, mit der aifunc() aufgerufen wird

void (*thinkfunc)(edict_t *self)

Think-Funktion, die in diesem Frame ausgeführt wird

Strukturtyp spawn_temp_t

Im ganzen Spiel existiert nur eine Struktur dieses Typs (st in `g_main.c`), welche dazu da ist, Entity-Feld-Variablen aus dem Level-Editor kurzzeitig zwischenzuspeichern. Ein Entity kann bei der Ausführung seiner Erzeugungsfunktion (i.d.R. SP_*()) über diese Struktur st auf die im folgenden aufgeführten Variablen zugreifen und bei Bedarf deren Werte in die Entity-Struktur übernehmen (Beachte: die Variablen von st werden bei dem Verlassen der Erzeugungsfunktion für das momentante Entity ungültig, da sie wieder neu mit den Editor-Werten des nächsten Entities belegt werden, das mit der Erzeugung an der Reihe ist; später im Spiel wird diese Struktur dann nicht mehr benutzt!). Die Werte der folgenden Variablen können im Prinzip also für jedes Entity unterschiedlich genutzt werden, so daß hier zum besseren Verständnis beispielhaft ein Verwendungszweck (oft auch der einzige) aufgeführt wird:

char	*sky	Werte für die jeweiligen Configstrings bezüglich der Hintergrundgrafik, die in SP_worldspawn() eingetragen werden
float	skyrotate	
vec3_t	skyaxis	
char	*nextmap	in SP_worldspawn() wird hiermit level.nextmap gesetzt
int	lip	Bei bewegten Objekten (Tür, Knopf) wird diese Größe von der Gesamtbewegungsstrecke abgezogen, so daß diese nach der Bewegung noch lip Einheiten aus der Wand herausstehen.
int	distance	eine Distanz
int	height	eine Höhe
char	*noise	Name einer Geräusch-Datei
float	pausetime	eine Zeitspanne
char	*item	Name eines Items
char	*gravity	in SP_worldspawn() wird hiermit die Cvar gravity, also die globale Gravitation des Levels, gesetzt; ansonsten z.B. der Gravitationsmultiplikator des Entities
float	minyaw	werden für Begrenzung des YAW- und PITCH-Winkels des Geschützturms (turret) benutzt
float	maxyaw	
float	minpitch	
float	maxpitch	

Strukturtyp game_import_t – importierte game engine Funktionen

Im folgenden wird, falls nötig, ein Zeiger auf die Entity-Struktur des Spielers durch spieler dargestellt.

Textausgabe

void (*bprintf) (int printlevel, char *fmt, . . .)

Broadcast – Meldung an alle Spieler, printlevel kennzeichnet die Nachrichtenpriorität:

PRINT_LOW	(0)	Items aufnehmen
PRINT_MEDIUM	(1)	Tod
PRINT_HIGH	(2)	Wichtiges
PRINT_CHAT	(3)	Chat Mitteilungen (am wichtigsten), grüne Schrift

void (*dprintf) (char *fmt, . . .)

Debug Meldungen

void (*cprintf) (edict_t *ent, int printlevel, char *fmt, . . .)

Client Meldung, nur an den Client ent, printlevel s.o.

void (*centerprintf) (edict_t *ent, char *fmt, . . .)

Meldung an Client ent, Ausgabe in der Bildschirmmitte; Text wird nach 40 Zeichen abgeschnitten, falls kein manueller Zeilenwechsel (\n) erfolgt.

Speicherverwaltung

void * (*TagMalloc) (int size, int tag)

Diese Funktion zur Speicherverwaltung entspricht im großen und ganzen der C-Funktion malloc(): es wird ein Speicherbereich der Göße size allokiert und ein Zeiger auf diesen zurückgegeben. Zusätzlich wird dem Speicherbereich auch noch die Nummer tag zugewiesen. Dies erlaubt eine einfache Speicherbereinigung aller Speicherbereiche mit der gleichen Nummer „in einem Rutsch“ (gi.FreeTags()). tag nimmt dabei die folgenden Werte an:

TAG_GAME	(765)	Bereinigung beim Abtrennen der game dll
TAG_LEVEL	(766)	Bereinigung beim Laden eines neuen Levels

void (*FreeTags) (int tag)

Diese Funktion führt die oben angesprochene Speicherbereinigung aller Speicherbereiche durch. Es werden alle Speicherbereiche freigegeben, die durch Aufruf der Funktion gi.TagMalloc() mit der Nummer tag allokiert wurden.

void (*TagFree) (void *block)

Diese Funktion entspricht der C-Funktion free(). Sie gibt den Speicherbereich frei, auf den block zeigt, welcher mit Hilfe der Funktion gi.TagMalloc() allokiert wurde.

Netzwerkkommunikation

Die folgenden Funktionen dienen der Kommunikation, um temporäre Effekte und Bildschirmausgaben bei den Clients zu erzeugen:

void (*multicast)(vec3_t origin, multicast_t to)
void (*unicast) (edict_t *ent, qboolean reliable)
void (*WriteChar) (int c)
void (*WriteByte) (int c)
void (*WriteShort) (int c)
void (*WriteLong) (int c)
void (*WriteFloat) (float f)
void (*WriteString) (char *s)
void (*WritePosition) (vec3_t pos)
void (*WriteDir) (vec3_t pos)
void (*WriteAngle) (float f)

Das Kommunikationsprotokoll sieht dabei den folgenden Ablauf vor: Zuerst wird mit Hilfe der Funktion gi.WriteByte(c) die Klasse der folgenden Nachricht festgelegt, indem der Parameter c einen der folgenden Werte annimmt:

svc_muzzleflash	(1) „Mündungsfeuer" (Grafik und Geräusche) der Spielerwaffen und weitere Spieler-Effekte
svc_muzzleflash2	(2) Monster-Effekte
svc_temp_entity	(3) Temporäre Entities
svc_layout	(4) Bildschirmausgabe eines Layouts
svc_inventory	(5) Bildschirmausgabe der Items, die der Spieler besitzt
svc_stufftext	(11) einfache Textausgabe

Anschließend wird mit Hilfe der gi.Write*()-Funktionen die Nachricht zusammengestellt. Deren Aufbau hängt von der Nachrichtenklasse ab:

`svc_muzzleflash` bzw. `svc_muzzleflash2`

Diese Klassen erlauben das Erzeugen von Mündungsfeuern bei Schüssen sowie weitere Effekte bei Spielern bzw. Monstern. Mit gi.WriteShort(c) wird der Index des Entities im Feld aller Entity-Datenstrukturen g_edicts (S. 83) übermittelt. Der Parameter c hat also die Form Zeiger_auf_das_Entity − g_edicts, z.B. gi.WriteShort(ent − g_edicts). Anschließend wird mit gi.WriteByte(c) der Typ des Effektes ausgewählt. Dessen Parameter c kann die in `q_shared.h` definierten Werte MZ_∗ bzw. MZ2_∗ annehmen, z.B. gi.WriteByte(MZ_BLASTER); hier nur einige Beispiele:

MZ_BLASTER	(0) Mündungsfeuer Spielerwaffe Blaster
MZ_MACHINEGUN	(1) Mündungsfeuer Spielerwaffe Machinegun
MZ_LOGIN	(9) Effekt des Spielers beim Leveleintritt (DM)
MZ_LOGOUT	(10) Effekt des Spielers beim Verlassen des Spiels (DM)

MZ_ITEMRESPAWN	(11) Effekt, wenn Items wiedererscheinen (DM)
MZ_SILENCED	(128) Flag, welches das Schußgeräusch unterdrückt (i.d.R. wenn der Spieler das Item Silencer gerade benutzt); dieses Flag wird bitweise ODER mit dem Mündungsfeuereffekt verbunden, z.B. gi.WriteByte(MZ_BLASTER | MZ_SILENCED)

svc_temp_entity

Diese Nachrichtenklasse erlaubt es, temporäre Entities zu erzeugen. Welches Entity erzeugt werden soll, wird mit gi.WriteByte(c) festgelegt. Der Parameter c entspricht dabei einer der unten aufgeführten Konstanten TE_∗ (Elemente der Enumeration temp_event_t). Je nach temporärem Entity sind unterschiedliche Folgebefehle vonnöten. Deshalb ist die folgende Liste der temporären Entities in Gruppen unterteilt; am Anfang jeder Gruppe werden zuerst die Folgebefehle aufgelistet (dabei stehen die Variablen index und index2 für einen schon oben in svc_muzzleflash erläuterten Ausdruck der Form (ent − g_edicts); dieser Index dient möglicherweise zur Feinjustierung der temporären Entities; weiterhin ist zu berücksichtigen, daß die temporären Entities ab Nummer 30 (TE_BLASTER2) von den Programmierern der Quake 2-Mission-Packs (zusätzliche Quake 2-Missionen mit neuen Levels und teilweise neuen Waffen, Effekten etc.) hinzugefügt wurden und daher manche dieser temporären Entities nur „korrekt“ funktionieren, wenn die Models, Geräusche usw. aus diesen Mission-Packs vorhanden sind; die davon betroffenen temporären Entities sind mit einem * markiert).

I. Gruppe:
- ↪ gi.WriteByte(count)
- ↪ gi.WritePosition(origin)
- ↪ gi.WriteDir(direction)
- ↪ gi.WriteByte(color)

Es werden count viele Partikel am Punkt origin erzeugt, die sich in Richtung direction bewegen:

TE_SPLASH	(10) Partikelansammlung, color bestimmt hier die Partikelart:
SPLASH_SPARKS	(1) orange-rote Partikelwolke mit Geräusch
SPLASH_BLUE_WATER	(2) blaue Partikelwolke
SPLASH_BROWN_WATER	(3) braune Partikelwolke
SPLASH_SLIME	(4) grün-weiße Partikelwolke
SPLASH_LAVA	(5) gelb-rot-orange Partikelwolke
SPLASH_BLOOD	(6) rote Partikelwolke
sonstige Werte	(0,7,8,...) grau-weiß-schwarze Partikelwolke
TE_LASER_SPARKS	(15) absinkende Partikelwolke der Farbe color (Farben-Index in der Quake 2-Palette mit Wertebereich: 0 – 255)

TE_WELDING_SPARKS	(25) dto., zusätzlich noch Lichtblitz im Punkt origin
TE_TUNNEL_SPARKS	(29) aufsteigende Partikelwolke der Farbe color

`II. Gruppe:`

↪ gi.WriteShort(index)
↪ gi.WritePosition(start)
↪ gi.WritePosition(end)

Objekte werden vom Punkt start des Entities mit Index index bis zum Punkt end erzeugt:

TE_PARASITE_ATTACK	(16) „Tentakel“ des Parasite-Monsters
TE_MEDIC_CABLE_ATTACK	(19) „Tentakel“ des Medic-Monsters
TE_HEATBEAM*	(38) sich drehender rot-gelb-orange texturierter Strahl; bezieht sich index auf einen Client, wird eine Spirale um den Strahl gelegt und ein Geräusch ausgegeben
TE_MONSTER_HEATBEAM*	(39) dto. mit oranger Partikelwolke am Punkt start

`III. Gruppe:`

↪ gi.WritePosition(start)
↪ gi.WritePosition(end)

Objekte werden vom Punkt start bis zum Punkt end erzeugt:

TE_RAILTRAIL	(3) blaue Spirale mit weiß-grauen Partikeln gefüllt und einem Schußgeräusch
TE_BUBBLETRAIL	(11) vereinzelte weiße Partikel
TE_BFG_LASER	(23) grüner transparenter Laserstrahl
TE_BLUEHYPERBLASTER	(27) vereinzelte gelb-orange Partikel
TE_DEBUGTRAIL	(34) blaue Partikel-Linie
TE_BUBBLETRAIL2	(41) mehrere weiße Partikel mit blechernem Geräusch

`IV. Gruppe:`

↪ gi.WritePostion(origin)
↪ gi.WriteDir(direction)

Objekte werden im Punkt origin erzeugt und „orientieren“ sich in Richtung direction:

TE_GUNSHOT	(0) kleine graue Partikelwolke mit Rauch
TE_BLOOD	(1) kleine rote Partikelwolke
TE_BLASTER	(2) kleine gelb-rote Partikelwolke mit blechernem Geräusch (gelb-roter Einschlag)
TE_SHOTGUN	(4) kleine graue Partikelwolke mit Rauchwolke
TE_SPARKS	(9) wenige rot-orange Partikel
TE_SCREEN_SPARKS	(12) grün-weißer Einschlag
TE_SHIELD_SPARKS	(13) hellblauer Einschlag

TE_BULLET_SPARKS	(14) wenige rot-orange Partikel mit Rauchwolke
TE_GREENBLOOD	(26) gelb-grüne Partikelwolke
TE_BLASTER2*	(30) weiß-grüner Einschlag mit Objekt in der Mitte
TE_MOREBLOOD	(42) große rote Partikelwolke
TE_HEATBEAM_SPARKS	(43) weiß-grauer Einschlag
TE_HEATBEAM_STEAM	(44) gelb-orange-roter Einschlag
TE_ELECTRIC_SPARKS	(46) dunkelblauer Einschlag
TE_FLECHETTE*	(55) hellblauer Einschlag mit Objekt in der Mitte

`V. Gruppe:`
↪ gi.WritePosition(origin)

Objekte werden im Punkt origin erzeugt:

TE_EXPLOSION1	(5) gelb-oranger Explosionspilz mit gelb-orangen Explosionspartikeln und Explosionsgeräusch (Explosion)
TE_EXPLOSION2	(6) große Explosion
TE_ROCKET_EXPLOSION	(7) Raketen-Explosion
TE_GRENADE_EXPLOSION	(8) Granaten-Explosion
TE_ROCKET_EXPLOSION_WATER	(17) Raketen-Explosion unter Wasser (Explosionsgeräusch gedämpft)
TE_GRENADE_EXPLOSION_WATER	(18) Granaten-Explosion unter Wasser (dto.)
TE_BFG_EXPLOSION	(20) kleiner grün-weißer sternförmiger Lichtblitz
TE_BFG_BIGEXPLOSION	(21) große explodierende grün-weiße Partikelwolke
TE_BOSSTPORT	(22) pilzartige grau-gelb-rote Partikelwolke mit Geräusch
TE_PLASMA_EXPLOSION	(28) Explosion
TE_PLAIN_EXPLOSION	(35) Explosion ohne Partikel
TE_CHAINFIST_SMOKE	(45) kleine Wolke aus grau-schwarzen Partikeln
TE_TRACKER_EXPLOSION*	(47) große schwarze explodierende Partikelwolke mit Geräusch
TE_TELEPORT_EFFECT	(48) große weiß-graue Partikelwolke
TE_DBALL_GOAL	(49) obigem sehr ähnlich
TE_NUKEBLAST	(51) türkisfarbene sich nach einem Schema bewegende Partikelwand
TE_WIDOWSPLASH	(52) leicht explodierende rot-hellgrüne Partikelkugel
TE_EXPLOSION1_BIG*	(53) ähnlich der Plain-Explosion
TE_EXPLOSION1_NP	(54) ähnlich der Plain-Explosion

`VI. Gruppe:`
- ↪ gi.WriteShort(index)
- ↪ gi.WriteShort(index2)
- ↪ gi.WritePosition(end)
- ↪ gi.WritePosition(start)

Objekte werden vom Punkt start des Entities mit Index index2 bis zum Punkt end des Entities mit Index index erzeugt:

TE_LIGHTNING*	(33) weiß-blau texturierte Zickzack Strahlen mit knisterndem Geräusch

`VII. Gruppe:`
- ↪ gi.WritePosition(origin)
- ↪ gi.WriteShort(index)

Objekte werden im Punkt origin des Entities mit Index index erzeugt:

TE_FLASHLIGHT	(36) „weißes“ Licht

`VIII.Gruppe:`
- ↪ gi.WritePosition(start)
- ↪ gi.WritePosition(end)
- ↪ gi.WriteByte(color)

Objekte der Farbe color werden vom Punkt start bis zum Punkt end erzeugt:

TE_FORCEWALL	(37) absinkende Partikel

`IX. Gruppe:`
- ↪ gi.WriteShort(id)
- ↪ gi.WriteByte(count)
- ↪ gi.WritePosition(origin)
- ↪ gi.WriteDir(direction)
- ↪ gi.WriteByte(color)
- ↪ gi.WriteShort(speed)
- ↪ gi.WriteLong(time)

Es werden count Objekte der Farbe color am Punkt origin mit der Geschwindigkeit speed in Richtung direction time lang (in Millisekunden) erzeugt:

TE_STEAM	(40) Partikel; id ist dabei eine ständig inkrementierte Zahl zwischen 0 und 20000, wird id auf -1 gesetzt, so fällt die letzte Teilnachricht (gi.WriteLong()) weg und die Partikel werden nur einmal erzeugt

```
X.   Gruppe:
```
↪ gi.WriteShort(muster)
↪ gi.WritePosition(origin)

Am Punkt origin werden nach dem Muster muster die Objekte erzeugt:

TE_WIDOWBEAMOUT	(50) große bunte Partikelwolke, die sich verschiedenartig bewegt; benutzte muster sind 20000 und 20001

Es gibt auch unbenutzte temporäre Entities:

TE_GRAPPLE_CABLE	(24) „crappling hook cable" bei CTF, Nachrichtenfolge unbekannt (im CTF-Quellcode zu finden)
TE_RAILTRAIL2	(31) in Engine (v 3.20) nicht definiert
TE_FLAME	(32) in Engine (v 3.20) nicht definiert

`svc_layout`

Diese Nachrichtenklasse ermöglicht es, ein komplexes Layout auf dem Bildschirm darzustellen, das aus Bildern und Text zusammengesetzt sein kann. Hierzu wird ein Layout-String erzeugt, der dann mit gi.WriteString(char *s) als Parameter s übergeben wird. Beachte: Der String s darf dabei nicht größer als 1024 Byte sein! Damit das Layout auch auf dem Bildschirm des Clients dargestellt wird, muß die Statusvariable spieler→client→ps.stats[STAT_LAYOUTS] auf dem Wert 1 stehen. Der Layout-String kann dabei aus den untenstehenden Grundelementen aufgebaut werden.
Erläuterungen: [zahl], [text], [datei] sind hierbei nur Platzhalter für konkrete Werte, keine Variablen! Desweiteren sind die Leerzeichen zu beachten, da sie als Trennzeichen der einzelnen Ausdrücke dienen; insbesondere das letzte Leerzeichen sollte man nicht vergessen, da beim Zusammensetzen mehrerer Zeilen sonst das Ende des letzten Befehls nicht mehr erkennbar ist. Der Wert [zahl] hinter xv und yv bezeichnet die x- bzw. y-Koordinaten der linken oberen Ecke der einzusetzenden Pixelgrafik, welche im PCX-Format vorliegen muß (kleiner Exkurs: dies gilt auch für die Texte, da intern jeder Buchstabe eine Grafik von 8x8 Pixeln darstellt; alle Buchstaben sind in der Datei `pics/conchars.pcx` gespeichert; aus dieser werden die 8x8 Pixel großen Stücke für die Buchstaben herausgeschnitten; die Buchstaben sind dort nach dem 7-bit ASCII-Code sortiert und in zwei Farben vorhanden; ein char-Wert von 0 bis 127 (also der „normale" Zeichenwert in C) spricht die obere Hälfte der Zeichen – weiße Buchstaben und Sonderzeichen – an, Werte von 128 bis 255 die untere Hälfte – die grünen Buchstaben und Sonderzeichen (für den grünen Text gibt es besondere Möglichkeiten, so daß auch hier der „normale" C-Wert der chars Verwendung findet); somit können z.B. die deutschen Sonderzeichen ö, ß etc. in den Zeichensatz eingeführt werden, indem man einen (hoffentlich) unbenutzten Platz in der `conchars.pcx` durch die Grafik des neuen Zeichens ersetzt und dieses z.B. über den Oktalwert des Platzes anspricht: *„Wir m\223chten, da\230 …*"; Exkurs-Ende). Die [zahl]-Werte definieren also die Koordinaten

der einzusetzenden Grafik, und zwar bezüglich eines Koordinatensystems, dessen Ursprung bei einer Bildschirmauflösung von 320x240 Bildpunkten in der linken oberen Ecke des Bildschirms liegt; die positive x-Achse zeigt nach rechts und die positive y-Achse nach unten. Bei anderen Bildschirmauflösungen liegt dieser 320x240 große Bildausschnitt genau in der Mitte des Bildschirms (bei 400x300 geht das Koordinatensystem auf dem Bildschirm dann also von -40 bis 360 in x-Richtung und von -30 bis 270 in y-Richtung). Die Werte [datei] bezeichnen den Dateinamen der Bilddatei ohne die Dateiendung .pcx, aber inklusive des Dateipfades relativ zum `pics`-Verzeichnis: Liegt die Datei (`Bild.pcx`) im `pics`-Verzeichnis, so ist kein zusätzlicher Pfad nötig (Angabe also nur: `Bild`), liegt sie im Unterverzeichnis `Bilder`, dann muß dieses mitangegeben werden (dann also: `Bilder/Bild`). Die Werte [text] stellen einen (fast) beliebigen String dar, der in \” eingebettet sein muß, falls Freizeichen darin auftreten, damit in [text] vorhandene Freizeichen nicht als Befehlsende interpretiert werden (⇒ das Zeichen " macht Probleme bei der Ausgabe im Layout, da ein einfaches " vom C-Programm als Stringende-Zeichen verstanden und das \" von Quake 2 innerhalb des Layouts als Textende-Zeichen interpretiert wird; Ausweg: über eine oben angesprochene Quake 2-Zeichensatz-Änderung, oder evtl. auch über die Oktalangabe des grünen " – \42 – im string-Modus (s.u.) für ein grünes Anführungszeichen bzw. im string2-Modus für ein weißes Anführungszeichen).

xv␣[zahl]␣yv␣[zahl]␣picn␣[datei]␣	Bilder
xv␣[zahl]␣yv␣[zahl]␣string␣\”[text]\”␣	weißer Text
xv␣[zahl]␣yv␣[zahl]␣cstring␣\”[text]\”␣	weißer Text zentriert (bzgl. 320x240 Bildausschnitt)
xv␣[zahl]␣yv␣[zahl]␣string2␣\”[text]\”␣	grüner Text
xv␣[zahl]␣yv␣[zahl]␣cstring2␣\”[text]\”␣	grüner Text zentriert

Weiterhin gibt es speziell für die Client-Wertung in der Intermission im DM die Möglichkeit (siehe Quellcode: p_hud.c/DeathmatchScoreboardMessage()):

client␣[x]␣[y]␣[playernum]␣[score]␣[ping]␣[time]␣

Um beim Erzeugen des Layout-Strings auch Variableninhalte einzubringen, kann man die Hilfsfunktion Com_sprintf() benutzen (Hilfsfunktionen S. 133).

`svc_inventory`

Diese Klasse ist speziell dafür gedacht, das Inventar des Spielers aufzulisten. Hierzu wird für jedes Element des Inventars gi.WriteShort(int c) aufgerufen. Der Parameter c entspricht dabei den jeweiligen Werten des Arrays spieler→client→pers.inventory. Damit dies auf dem Client-Bildschirm auch dargestellt wird, muß die Statusvariable spieler→client→ps.stats[STAT_LAYOUTS] auf dem Wert 2 stehen.

svc_stufftext

Diese Klasse erlaubt es, Chat-Text auszugeben. Dieser wird mit gi.WriteString(char *s) übermittelt (1024 Byte Grenze beachten!). Nachdem diese Nachricht verschickt wurde (s.u.) wird diese anscheinend mit Hilfe des Standard-Chat-Mechanismus (g_cmds.c/Cmd_Say_f()) übertragen (dort dann sogar 150 Byte Grenze).

Nachdem nun mit Hilfe der gi.Write*()-Funktionen die Nachricht hinreichend spezifiziert und alle notwendigen Inhalte in der richtigen Reihenfolge übermittelt wurde, also die Nachricht im Server zusammengestellt ist, wird diese mit einer der beiden folgenden Funktionen verschickt:

void (*unicast) (edict_t *ent, qboolean reliable)

Die Nachricht wird nur an einen Client ent verschickt. Wird dabei reliable auf den Wert true gesetzt, wird zur Kommunikation eine sichere Verbindung (TCP) benutzt (wichtige Daten), bei false ein Datagramm (UDP).

void (*multicast) (vec3_t origin, multicast_t to)

Die Nachricht wird an mehrere Clients geschickt. origin stellt den Ausgangspunkt des Ereignisses dar, to spezifiziert die Adressaten genauer und kann folgende Werte annehmen (Enumeration multicast_t):

MULTICAST_ALL	(0) alle Clients
MULTICAST_PHS	(1) alle Clients im PHS (Potentially Hearable Set: S. 130) des Punktes origin
MULTICAST_PVS	(2) alle Clients im PVS (Potentially Visible Set: S. 130) des Punktes origin
MULTICAST_ALL_R MULTICAST_PHS_R MULTICAST_PVS_R	wie oben, nur reliable, also über eine gesicherte Verbindung (normal nur Datagramm)

Konsole

int (*argc) (void)

gibt die Argumentenanzahl des letzten Konsolenbefehls zurück (der Befehl selbst zählt hierbei auch als Argument)

char* (*argv) (int n)

gibt das Argument des letzten Konsolenbefehls mit Indexnummer n zurück (der Konsolenbefehl selbst hat die Indexnummer 0)

char* (*args) (void)

gibt alle Argumente mit Index $\geq$ 1 des letzten Konsolenbefehls als aneinanderhängenden String zurück

void (*AddCommandString) (char *text)
führt den Konsolenbefehl text auf der Server-Konsole aus, als ob er dort eingegeben worden wäre (z.B. um den Level zu wechseln)

Verschiedenes

void (*DebugGraph) (float value, int color)
Dies ist eine Funktion, um Debug-Informationen während des Spiels grafisch darzustellen. Wird am jeweiligen Client die Konsolenvariable debuggraph auf 1 gesetzt, so wird der Debug-Wert value in der Farbe color (0 – 255) bei Aufruf dieser Funktion als wenige Pixel breiter Balken im linken unteren Bereich des Client-Bildschirms dargestellt. Die so entstehenden Balken bewegen sich im Laufe der Zeit von links nach rechts, so daß eine kontinuierliche Darstellung des Wertes entsteht, sofern diese Funktion, gi.DebugGraph(), im Laufe der Zeit öfter (z.B. in jedem Frame) aufgerufen wird. Das Umschalten der Konsolenvariablen debuggraph kann auch direkt vom Quellcode aus geschehen (s.o.), dann aber nur für den Client, der auf dem gleichen Rechner wie der Server läuft.

void (*Pmove) (pmove_t *pmove)
Diese Funktion bewegt den Spieler entsprechend den Werten in pmove (Client-Frame S. 147). „Eingabeparameter“ (E) müssen vor Aufruf dieser Funktion gesetzt werden, „Ausgabeparameter“ (A) dienen sozusagen als Rückgabewert. pmove_t ist dabei wie folgt definiert (eine pmove_t-Struktur wird durch pm dargestellt):

pmove_state_t	s	(E/A) Bewegungs-Status, dient u.a. auch der Client-Bewegungsvorhersage
usercmd_t	cmd	(E) Struktur mit der Benutzereingabe
qboolean	snapinitial	(E) true, wenn s außerhalb eines Funktionsaufrufs von gi.Pmove() geändert wurde
int	numtouch	(A) Anzahl der berührten Entities
edict_t	*touchents[32]	(A) Liste der berührten Entities; es werden aber nur MAXTOUCH (32) berücksichtigt.
vec3_t	viewangles	(A) Sichtrichtung des Spielers (⇒ spieler→client→v_angle und spieler→client→ps.viewangle)
float	viewheight	(A) Sichthöhe des Spielers (⇒ spieler→viewheight)
vec3_t	mins, maxs	(A) Ausdehnung der Bounding Box des Spielers (⇒ spieler→mins, spieler→maxs)
edict_t	*groundentity	(A) Entity, auf dem der Spieler gerade steht (⇒ spieler→groundentity und falls ≠NULL spieler→groundentity_linkcount=pm.groundentity→linkcount)
int	waterlevel	(A) Wassertiefe; Wertebereich: 0 (nicht im Wasser) – 3 (ganz unter Wasser) (⇒ spieler→waterlevel)

int	watertype	(A) Art des „Wassers“; Wertebereich: CONTENTS_LAVA, CONTENTS_WATER, CONTENTS_SLIME, 0 (⇒ spieler→watertype)
trace_t	(*trace)()	(E) Zeiger auf die gi.trace() Funktion, (S. 128)
int	(*pointcontents)()	(E) Zeiger auf die gi.pointcontents() Funktion (S. 127)

Der hierbei verwendete Strukturtyp pmove_state_t dient u.a. der Client-Bewegungsvorhersage: wird diese irgendwo im game-Code verändert, so führt das in gewissem Maße zu Vorhersage-Fehlern! Um die Synchronität der Client Vorhersage zu gewährleisten, werden keine float-Typen benutzt:

pmtype_t	pm_type	Art der Bewegung
short	origin[3]	Position bezüglich Level-Ursprung (⇔ spieler→origin)
short	velocity[3]	Bewegungsvektor (⇔ spieler→velocity)
byte	pm_flags	Bewegungsflags (s.u.)
byte	pm_time	Zeit, jede Einheit ≙ 8 ms (s.u. pm_flags)
short	gravity	Gravitation (⇐ Wert der Cvar sv_gravity)
short	delta_angles[3]	Sichtrichtungsdelta (z.B. durch Teleporter oder drehende Objekte geändert); ergibt zusammen mit pm.cmd.angles[3] (s.u.) die resultierende Sichtrichtung (pm.viewangles)

Der verwendete Typ pmtype_t ist eine Enumeration mit folgenden Werten:

PM_NORMAL	(0) nomale Bewegung mit Beschleunigen und Drehen
PM_SPECTATOR	(1) Zuschauermodus, wie normal, nur keine Kollisionserkennung (noclip)
PM_DEAD	(2) Tod, keine Bewegung möglich
PM_GIB	(3) Spieler ist in Teile „zerfetzt“, hat also eine andere Bounding Box und kann sich nicht mehr bewegen
PM_FREEZE	(4) eingefroren, keine Bewegung möglich (während Intermission)

Die Bewegungsflags pm_flags können sein:

PMF_DUCKED	(1) geduckt
PMF_JUMP_HELD	(2) springend
PMF_ON_GROUND	(4) normal auf dem Boden
PMF_TIME_WATERJUMP	(8) ? pm_time-langer Sprung im Wasser
PMF_TIME_LAND	(16) pm_time ist die Zeitspanne bis zum nächst möglichen Sprung
PMF_TIME_TELEPORT	(32) pm_time ist die Zeitspanne, während der man sich beim Teleportieren nicht bewegen kann
PMF_NO_PREDICTION	(64) Bewegungsvorhersage wird temporär abgeschaltet

Den in pmove_t verwendeten Strukturtyp usercmd_t schickt der Client jeden Client-Frame an den Server (dies sind die Tastatur-, Maus- und sonstigen Eingabewerte). Dieser ruft dann globals.ClientThink() u.a. mit einer Struktur diesen Typs als Parameter auf. Dort wird dann die Bewegung mit Hilfe gi.Pmove() ausgeführt (Client-Frame S. 147). usercmd_t sieht nun folgendermaßen aus:

byte	msec	? die seit dem letzen Paket vergangene Zeit auf dem Client-Rechner
byte	buttons	die gedrückten Knöpfe auf Tastatur, Maus, ...
short	angles[3]	die vom Spieler verlangte Sichtänderung
short	forwardmove	dto. bzgl. Vorwärtsbewegung
short	sidemove	dto. bzgl. Seitwärtsbewegung
short	upmove	dto. bzgl. Hoch-, Runterbewegung
byte	impulse	für sonstige Befehle reservierter Platz (mögliche Erweiterungen)
byte	lightlevel	Lichtstärke am Ort des Spielers (⇒ spieler→light_level), für Monster-AI beim Sehen des Spieler

Nun noch die Werte für buttons:

BUTTON_ATTACK	(1) Bit beim Schießen gesetzt
BUTTON_USE	(2) unbenutzt
BUTTON_ANY	(128) bei allen Knopfdrücken gesetzt

void (*error) (char *fmt, ...)

Die Funktion bricht das Spiel ab (Server wird beendet) und zeigt die Fehlermeldung fmt incl. eventueller Argumente (...) an der Client-Konsole.

void (*configstring) (int num, char *string)

Configstrings beinhalten alle Index-Strings, die Lichtstile (flackern, blinken, ...) sowie verschiedene levelbezogene Daten wie Himmel-Definitionen oder CD-Tracks. Alle momentan existierenden Configstrings werden an die Clients beim Verbinden geschickt und Veränderungen werden an alle verbundenen Clients geschickt. Nicht zuletzt deshalb sollte ein Configstring maximal MAX_QPATH (64) Zeichen lang sein (mit Ausnahme von CS_STATUSBAR). Gesetzt oder verändert wird ein Configstring, indem sein Name (s.u.) im num- und sein „Wert" – der String also – im string-Parameter dieser Funktion übergeben werden. Die vorhandenen Configstrings bedeuten (hierbei bedeutet ein Wert von +x, daß x zu der zuletzt erwähnten Zahl hinzuaddiert werden muß):

CS_NAME	(0) Level Name
CS_CDTRACK	(1) CD-Musik-Titel für den Level
CS_SKY	(2) Bild des Himmels
CS_SKYAXIS	(3) Drehachse des Himmels (Format: %f %f %f)
CS_SKYROTATE	(4) Drehgeschwindigkeit des Himmels

CS_STATUSBAR	(5) Hier wird das „Statusbarprogramm“ übergeben (Statusbar S. 102); die folgenden Werte sind anscheinend aufgrund der Länge dieses Configstrings frei
CS_AIRACCEL	(29) für Luftbeschleunigungskontrolle
CS_MAXCLIENTS	(30) Wert der Cvar maxclients
CS_MAPCHECKSUM	(31) um „cheater“-Maps im DM aufzuspüren
CS_MODELS	(32 – +MAX_MODELS−1) die Model-Configstrings (siehe gi.modelindex() S. 126)
CS_SOUNDS	(+1 – +MAX_SOUNDS−1) die Geräusch-Configstrings (siehe gi.soundindex() S. 126)
CS_IMAGES	(+1 – +MAX_IMAGES−1) die Bilder-Configstrings (siehe gi.imageindex() S. 126)
CS_LIGHTS	(+1 – +MAX_LIGHTSTYLES−1) Lichteranimationstabellen: String aus Kleinbuchstaben, der das Lichtverhalten definiert (max. 64 Bytes!), a steht für ganz dunkel und z für doppelte Helligkeit (werden i.d.R. in SP_worldspawn() definiert, z.B. Kerze: „mmmmmaaaaammmmmaaaaaabcdefgabcdefg“)
CS_ITEMS	(+1 – +MAX_ITEMS−1) pickup-Namen aller Items in itemlist (Items S. 86)
CS_PLAYERSKINS	(+1 – +MAX_CLIENTS−1) Name und Skin aller Spieler
CS_GENERAL	(+1 – +MAX_GENERAL−1) Platz für Erweiterungen
MAX_CONFIGSTRINGS	(+1=2080) maximale Anzahl an Configstrings

void (*linkentity) (edict_t *ent)

Das Entity ent wird durch den Aufruf dieser Funktion „verbunden“. Erst nach dem Aufruf dieser Funktion wird das Entity zu den Clients geschickt und für die Kollisionserkennung benutzt, also existiert ein Entity vor dem ersten Aufruf dieser Funktion faktisch noch nicht in der 3D-Welt. Ändern sich im Verlauf der Zeit die Parameter bezüglich der Größe, der Position oder der „Solidity“ (ent→solid) des Entities, so muß erneut diese Funktion aufgerufen werden, um das Entity wieder zu verbinden.

void (*unlinkentity) (edict_t *ent)

Bevor ein mit der Welt interagierendes Entity ent entfernt und/oder gelöscht wird, muß es mit Hilfe dieser Funktion von der Welt „getrennt“ werden. Nach dem Aufruf dieser Funktion interagiert das Entity ent nicht mehr mit der restlichen Welt. Somit stellt dies gewissermaßen die Umkehrfunktion von gi.linkentity() dar.

Bilder

int (*imageindex) (char *name)

Beim ersten Aufruf wird das Bild aus der Datei name geladen, ein Configstring (S. 124) und einige serverinterne Vermerke erzeugt und anschließend ein Index zurückgegeben; bei weiteren Aufrufen wird dann nur noch der Index zurückgegeben. Es empfiehlt sich daher, beim Laden eines Levels alle benötigten Bilder hiermit zu cachen. (name ist der Dateiname incl. Pfad der Bilder relativ zum Verzeichnis `pics` ohne die Dateiendung `.pcx`)

Models

int (*modelindex) (char *name)

wie gi.imageindex(); betrifft Models anstelle von Bildern (`name` ist der Dateiname incl. Pfad der Models relativ zum Verzeichnis des momentanen Spiels – i.d.R. also incl. `models/` – mit der Dateiendung `.md2`)

void (*setmodel) (edict_t *ent, char *name)

Das Model des Entities ent wird auf das in der Datei name definierte gesetzt (entspricht wohl dem Aufruf: *ent→s.modelindex=gi.modelindex(name);*).

Geräusche

int (*soundindex) (char *name)

siehe gi.imageindex(); betrifft Geräusche anstelle von Bildern (name ist der Dateiname incl. Pfad der Geräusche relativ zum Verzeichnis `sounds` mit der Dateiendung `.wav`)

void (*sound) (edict_t *ent, int channel, int soundindex, float volume, float attenuation, float timeofs)

Das Entity ent strahlt das Geräusch mit Index soundindex aus. channel spezifiziert den Kanal des Geräusches; ein schon auf dem gleichen Kanal spielendes Geräusch wird überschrieben, mit Ausnahme von CHAN_AUTO:

CHAN_AUTO	(0) neues Geräusch kommt hinzu, kein anderes wird überschrieben
CHAN_WEAPON	(1) Waffen-Geräusch
CHAN_VOICE	(2) Geräusch von Entities
CHAN_ITEM	(3) Powerup-Geräusch z.B. bei Quad Damage oder Invulnerability
CHAN_BODY	(4) Spielergeräusche z.B. Schritte, Magenknurren, ...
CHAN_∗	(5-7) noch freie Kanäle

Zusätzlich können noch folgende Flags gesetzt werden:

CHAN_NO_PHS_ADD	(8) Jeder hört das Geräusch (entspricht ATTN_NONE).

CHAN_RELIABLE	(16) wichtiges Geräusch; es wird eine gesicherte Verbindung (TCP) benutzt, kein Datagramm (UDP).

volume bezeichnet die Lauststärke (zwischen 0 und 1); attenuation bezeichnet die Reichweite:

ATTN_NONE	(0) volle Lautstärke auf dem ganzen Level
ATTN_NORM	(1) sehr große Reichweite
ATTN_IDLE	(2) mittlere Reichweite
ATTN_STATIC	(3) sehr begrenzte Reichweite

timeofs beschreibt die Zeitverzögerung, bevor das Geräusch gespielt wird.

void (*positioned_sound) (vec3_t origin, edict_t *ent, int channel, int soundindex, float volume, float attenuation, float timeofs)

ähnlich zu gi.sound(); origin bezeichnet den Ausgangspunkt des Geräusches relativ zum Level-Ursprung, ent bezeichnet den Verursacher.

Kollisionserkennung

int (*pointcontents) (vec3_t point)

Gibt den Inhalt des Punktes point zurück; dieser Wert entspricht einem (oder mehreren) der folgenden Flags:

CONTENTS_SOLID	(1) feste Objekte
CONTENTS_WINDOW	(2) durchsichtige feste Objekte
CONTENTS_AUX	(4) unbenutzt
CONTENTS_LAVA	(8) Lava
CONTENTS_SLIME	(16) Säure
CONTENTS_WATER	(32) Wasser
CONTENTS_MIST	(64) Nebel, unbenutzt
LAST_VISIBLE_CONTENTS	(64) Alle untenstehenden contents sind nicht sichtbar.
CONTENTS_AREAPORTAL	(0x00008000) Portale, die im Level-Design benutzt werden, um Speicher zu sparen; im geschlossenen Zustand (die Regel) kann weder Licht (Grafik) noch Ton (Geräusche) passieren.
CONTENTS_PLAYERCLIP	(0x00010000) Brush, den ein Spieler nicht passieren kann
CONTENTS_MONSTERCLIP	(0x00020000) Brush, den ein Monster nicht passieren kann
CONTENTS_CURRENT_0	(0x00040000) im Winkel 0° „fließende“ Textur
CONTENTS_CURRENT_90	(0x00080000) dto. mit Winkel 90°
CONTENTS_CURRENT_180	(0x00100000) dto. mit Winkel 180°
CONTENTS_CURRENT_270	(0x00200000) dto. mit Winkel 270°

CONTENTS_CURRENT_UP	(0x00400000) dto. nach oben
CONTENTS_CURRENT_DOWN	(0x00800000) dto. nach unten
CONTENTS_ORIGIN	(0x01000000) Ursprungs-Brush für Rotationen
CONTENTS_MONSTER	(0x02000000) lebende Monster/Spieler
CONTENTS_DEADMONSTER	(0x04000000) tote Monster/Spieler
CONTENTS_DETAIL	(0x08000000) Brushes, die nicht an QVIS übergeben werden und nach den QVIS-Blättern eingefügt werden
CONTENTS_TRANSLUCENT	(0x10000000) transparenter Brush
CONTENTS_LADDER	(0x20000000) Leiter

trace_t (*trace) (vec3_t start, vec3_t mins, vec3_t maxs, vec3_t end, edict_t *passent, int contentmask)

Spannt eine Box von Punkt start bis maximal zum Punkt end auf. Sollte unterwegs ein Objekt des Typs contentmask getroffen werden, stoppt die Box, es sei denn, es handelt sich um das Entity passent. mins definiert dabei die untere linke Ecke, maxs die obere rechte Ecke der Box bezüglich des Punktes start (sind beide 0, so degeneriert die Box zu einer Linie). contentmask ist dabei eine Kombination der CONTENTS_$*$-Flags (s.o.); einige Kombinationen sind schon vordefiniert (durch bitweise ODER-Verknüpfung der Flags; CONTENT_ ist hier zugunsten der Übersichtlichkeit weggelassen):

MASK_ALL	(-1) alles
MASK_SOLID	(SOLID | WINDOW) alle festen Objekte
MASK_PLAYERSOLID	(SOLID | PLAYERCLIP | WINDOW | MONSTER) alle für den Spieler als fest erscheinenden Objekte (incl. lebenden Monstern/Spielern)
MASK_DEADSOLID	(SOLID | PLAYERCLIP | WINDOW) dto. nur ohne lebende Monster/Spieler
MASK_MONSTERSOLID	(SOLID | MONSTERCLIP | WINDOW | MONSTER) alle für ein Monster als fest erscheinenden Objekte (incl. lebenden Monstern/Spielern)
MASK_WATER	(WATER | LAVA | SLIME) alles Flüssige
MASK_OPAQUE	(SOLID | SLIME | LAVA) alles Undurchsichtige
MASK_SHOT	(SOLID | MONSTER | WINDOW | DEADMONSTER) alles, was beschossen werden kann; alle festen Objekte (incl. lebenden und toten Monstern/Spielern)
MASK_CURRENT	(CURRENT_$*$) alles, was eine „fließende“ Textur besitzt

Diese Funktion hat eine Struktur des Typs trace_t als Rückgabewert:

qboolean	allsolid	true, wenn komplette Box in Solids; plane ist dann ungültig!
qboolean	startsolid	true, wenn Startpunkt in Solid war.

float	fraction	Bruchteil der kompletten Box (1.0, wenn bis end gelangt)
vec3_t	endpos	tatsächlicher Endpunkt
cplane_t	plane	Auftreffebene (Normale)
csurface_t	*surface	getroffene Oberfläche (Textur)
int	contents	Inhalt hinter der Oberfläche (siehe gi.pointcontents())
edict_t	*ent	getroffenes Entity

Dabei ist der Strukturtyp cplane_t:

vec3_t	normal	Normale der Ebene
float	dist	?
byte	type	? für schnelle Seiten-Tests
byte	signbits	Vorzeichenbits der Normalen (signx + (signy$\ll$1) + (signz$\ll$2))
byte	pad[2]	?

und der Strukturtyp csurface_t:

char	name[16]	Texturname
int	flags	Texturflags
int	value	Wertparameter

mit den Texturflags flags:

SURF_LIGHT	In value steht die Lichtintensität.
SURF_SLICK	beeinflußt Physik im Spiel (glatt)
SURF_SKY	wird nicht gemalt, kennzeichnet Oberfläche als Himmel
SURF_WARP	turbulente Wasserbewegung
SURF_TRANS33	33% durchsichtig
SURF_TRANS66	66% durchsichtig
SURF_FLOWING	die Textur „fließt“ in eine Richtung (siehe CONTENT_CURRENT_$*$)
SURF_NODRAW	nicht malen

int (*BoxEdicts) (vec3_t mins, vec3_t maxs, edict_t **list, int maxcount, int areatype)

mins (untere linke vordere Ecke) und maxs (obere rechte hintere Ecke) spannen eine Box relativ zum Level-Ursprung auf; die sich in dieser Box befindlichen Entities werden in die Liste list eigetragen, jedoch höchstens maxcount viele (also muß gelten: maxcount $\leq$ Länge der list!); areatype bezeichnet den Entitytyp, der berücksichtigt werden soll:

AREA_SOLID	(1) feste Entities
AREA_TRIGGERS	(2) Trigger-Entities

void (*SetAreaPortalState) (int portalnum, qboolean open)

Mit dieser Funktion werden Areaportals (CONTENTS_AEREAPORTAL) geschaltet. Ein Areaportal ist ein nicht sichtbares Objekt, das die Welt in voneinander getrennte Bereiche aufspaltet, wenn es geschlossen ist (und das Areaportal zwischen zwei sonst nur durch Brushes getrennten Räumen steht, die kein Entity sein dürfen). I.d.R. wird der Areaportalbrush in eine Tür eingeschlossen und tür$\rightarrow$target auf areaportal$\rightarrow$targetname gesetzt, so daß das

Areaportal durch die Tür geschaltet wird – also nur offen ist, wenn auch die Tür geöffnet ist. portalnum ist dabei der Wert in areaportal→style und open je nach gewünschtem Zustand des Areaportals true oder false.

qboolean (*AreasConnected) (int area1, int area2)

Prüfung, ob zwei Gebiete miteinander verbunden sind. area1 und area2 sind dabei die Nummern der beiden Gebiete (z.B. ent→areanum). Wird für die Verfolgung von Clients bei Monstern benötigt, um dem Effekt der Areaportals entgegenzuwirken, kein Geräusch passieren zu lassen.

qboolean (*inPVS) (vec3_t p1, vec3_t p2)

Überprüft, ob Punkt p1 im Potentially Visible Set (PVS) des Punktes p2 liegt. PVS ist eine von Quake 2 benutzte Datenstruktur, um Sichtbarkeit und Sichtlinien festzustellen (liegt p1 im PVS von p2, so liegt natürlich p2 auch im PVS von p1).

qboolean (*inPHS) (vec3_t p1, vec3_t p2)

Überprüft, ob Punkt p1 im Potentially Hearable Set (PHS) des Punktes p2 liegt. Wie oben, nur geht es hier um Geräusche, nicht um die Sichtbarkeit.

Strukturtyp game_export_t – exportierte game dll Funktionen

Die Struktur dieses Typs steht in engem Zusammenhang zum Programmablauf des ganzen Spiels, da sie diejenigen Variablen und Funktionen beinhaltet, über welche der Server auf die Funktionalität der game dll zugreifen und somit den Spielfluß beeinflussen kann. Deshalb sei zur nähren Erläuterung der meisten der hier aufgeführten Funktionen auf das nächste Kapitel B.3.3 verwiesen. Der hier behandelte Strukturtyp wird ausschließlich in der Funktion game_export_t *GetGameAPI(game_import_t *import) benutzt, um dem Server beim Laden der game dll die Server-game dll Schittstelle bereitzustellen. Deshalb werden zusätzlich zu den Funktionszeigern auch die in GetGameAPI() zugeordneten Funktionen erwähnt, falls die Bezeichnungen nicht identisch sind; Syntax: (=Funktion).

int apiversion

(=GAME_API_VERSION (3)) Die Versionsnummer der Schnittstellen zwischen game dll und der game engine

void (*SpawnEntities) (char *mapname, char *entstring, char *spawnpoint)

Entity Strukturen werden initialisiert; wird bei jedem Eintritt in ein neues Level aufgerufen

void (*Init) (void)
(=InitGame()) Initialisierungsfunktion; diese wird sofort nach dem Laden der game dll aufgerufen, um das neue Spiel zu initialisieren

void (*Shutdown) (void)
(=ShutdownGame()) „Aufräumaktionen“, die vor dem Abtrennen der game dll ausgeführt werden, also bei der Beendigung eines Spiels

void (*WriteGame) (char *filename, qboolean autosaved)
Speicherung des Spielstandes – Teil 1

void (*ReadGame) (char *filename)
Laden eines Spielstandes – Teil 1

void (*WriteLevel) (char *filename)
Speicherung des Spielstandes – Teil 2

void (*ReadLevel) (char *filename)
Laden eines Spielstandes – Teil 2

qboolean (*ClientConnect) (edict_t *ent, char *userinfo)
Client wird verbunden.

void (*ClientBegin) (edict_t *ent)
Client wird gestartet.

void (*ClientUserinfoChanged) (edict_t *ent, char *userinfo)
Eintragen der Userinfo-Daten in die Client-Struktur; wird immer aufgerufen, wenn ein Client ent eine seiner Cvars ändert, die mit dem Flag CVAR_USERINFO markiert ist.

void (*ClientDisconnect) (edict_t *ent)
Client trennt Verbindung.

void (*ClientCommand) (edict_t *ent)
Diese Funktion wird immer dann aufgerufen, wenn an einer Client-Konsole ein Konsolenbefehl eingegeben wird, der mit cmd␣ eingeleitet wurde. Das einleitende cmd␣ tritt hier dann nicht mehr als Parameter auf, so daß der Aufruf von gi.argv(0) (S. 121) gleich den String des Konsolenbefehls selbst zurückgibt (bei der Eingabe eines Konsolenbefehls an einer Client-Konsole, den die Engine nicht kennt, wird i.d.R. auch diese Funktion aufgerufen, so daß die Einleitung cmd␣ in den meisten Fällen auch weggelassen werden kann). In dieser Funktion (zu finden in der Datei `g_cmds.c`) wird dann die auf den Konsolenbefehl folgende Reaktion initiiert. Möchte man also eigene Konsolenbefehle hinzufügen, so muß man diese Funktion nur dahingehend erweitern.

void (*ClientThink) (edict_t *ent, usercmd_t *cmd)
Diese Funktion führt einen Client-Frame des Clients ent aus.

void (*RunFrame) (void)
Diese Funktion führt einen Server-Frame aus und dreht somit die Zeit in der 3D-Welt um 0.1 Sekunden weiter.

void (*ServerCommand) (void)
Diese Funktion entspricht gi.ClientCommand() von oben, nur daß es sich hier um Konsolenbefehle an der Server-Konsole handelt, die mit sv␣ eingeleitet wurden (das sv␣ ist hier unbedingt erforderlich); hier wird u.a. auch das schon öfter angesprochene IP-Filtering behandelt, auf das im folgenden noch kurz eingegangen wird. Beim IP-Filtering führt der Server eine Liste von IP-Adressen [ip]; diese werden im „Punktformat“ angegeben, wobei unspezifizierte Ziffern alle Werte darstellen, so daß man z.B. ganze C-Klasse Netzwerke angeben kann mit „192.246.40“. Die Server-Konsolenbefehle für die Listenverwaltung lauten:

sv addip [ip]	Hinzufügen von IP-Adressen zu der Liste
sv removeip [ip]	Entfernen von IP-Adressen aus der Liste
sv listip [ip]	Liste der IP-Adressen ausgeben
sv writeip [ip]	„sv addip [ip]“-Befehle für den Aufbau der momentanen Liste werden in die Datei `listip.cfg` geschrieben, damit diese zu einem späteren Zeitpunkt ausgeführt werden kann (mit exec listip.cfg), um die momentane Liste wieder herzustellen

Je nachdem welchen Wert die Cvar filterban aufweist, werden die IP-Adressen aus der Liste vom Spiel ausgeschlossen (Wert 1 – default), oder nur diese IP-Adressen zum Spiel zugelassen (Wert 0).

edict_t *edicts
Zeiger auf alle Entities (wie g_edicts)

int edict_size
Größe einer Struktur des Typs edict_t

int num_edicts
momentane Anzahl an Entities

int max_edicts
maximale Anzahl an Entities

Nützliche Funktionen

Operationen auf Basistypen

long Q_ftol (float f)
gibt den Long-Wert der Float f zurück

float Q_fabs (float f)
gibt den Betrag der Float f zurück

int Q_log2 (int val)
gibt den Zweierlogarithmus des Intergers val zurück

char *G_CopyString (char *in)
Diese Funktion kopiert den String in in einen neu allokierten Speicherbereich (mit Hilfe von gi.TagMalloc(. . . , TAG_LEVEL)) und gibt diesen kopierten String zurück.

int Q_stricmp (char *s1, char *s2)
Kontextinsensitiver Vergleich der beiden Strings s1 und s2; das Ergebnis ist kleiner, gleich oder größer 0, falls der String s1 kleiner, gleich oder größer als s2 ist (Groß- und Kleinschreibung findet keine Beachtung)

int Q_strncasecmp (char *s1, char *s2, int n)
wie Q_stricmp(), außer daß maximal die ersten n-Zeichen berücksichtigt werden und das Ergebnis bei Gleichheit 0 sowie bei Ungleichheit -1 ist

int Q_strcasecmp (char *s1, char *s2)
wie Q_strncasecmp(), außer daß die ersten 99999 Zeichen berücksichtigt werden

void Com_sprintf (char *dest, int size, char *fmt, . . .)
Diese Funktion entspricht der C-Funktion sprintf(), nur daß maximal size Zeichen in den Zielstring dest kopiert werden; mit dieser Funktion wird eine formatierte Ausgabe des Strings fmt mit den anschließenden Argumenten im String dest erzeugt, wobei die maximale Zeichenanzahl size in dest nicht überschritten wird; diese Funktion ist sehr nützlich, um ein Layout zusammenzustellen, in dem auch Variableninhalte mit ausgegeben werden sollen.

Vektoren und Winkel

float DEG2RAD (a)
gibt die Zahl a im Bogenmaß zurück, wobei a im Gradmaß sein sollte

float LerpAngle (float a2, float a1, float frac)
gibt einen Winkel zurück, der zwischen den Winkeln a2 und a1 liegt, wenn frac Werte zwischen 0 und 1 annimmt; der Rückgabewert ist $\mathsf{frac} * \mathsf{a1} + (1 - \mathsf{frac}) * \mathsf{a2}$

float anglemod (float a)
gibt den zu a äquivalenten Winkel zwischen 0° und 360° zurück

void AngleVectors (vec3_t angles, vec3_t forward, vec3_t right, vec3_t up)
Aufgrund des Orientierungsvektors angles (PITCH, YAW, ROLL) werden die Richtungsvektoren gesetzt: forward, der entlang der Orientierung zeigt, right, der bezüglich forward um 90° nach rechts gedreht ist, und up, der bezüglich forward um 90° nach oben gedreht ist; die drei Ergebnisvektoren werden nur gesetzt, falls sie auch vorhanden sind, d.h. forward, right und up können beim Aufruf auch NULL sein.

void vectoangles (vec3_t value1, vec3_t angles)
Aufgrund des Richtungsvektors value1 wird der Orientierungsvektor angles so gesetzt, daß angles in die Richtung von value1 weist; angles[2] bleibt dabei 0.

float vectoyaw (vec3_t v)
entspricht der obigen Funktion vectoangles(), außer daß hier nur die YAW-Komponente des Orientierungsvektors berechnet und zurückgegeben wird

void VectorClear (vec3_t v)
setzt alle Vektorkomponenten des Vektors v auf 0

void VectorSet (vec3_t v, float x, float y, float z)
setzt die Vektorkomponenten 0, 1 und 2 des Vektors v auf die Werte x, y und z

void VectorCopy (vec3_t in, vec3_t out)
kopiert den Vektor in in den Vektor out

int VectorCompare (vec3_t v1, vec3_t v2)
vergleicht die beiden Vektoren v1 und v2 miteinander; bei Gleichheit wird 1 zurückgegeben, ansonsten 0

vec_t VectorLength (vec3_t v)
gibt die Länge des Vektors v zurück

vec_t VectorNormalize (vec3_t v)
normalisiert den Vektor v und gibt dessen ursprüngliche Länge zurück

vec_t VectorNormalize2 (vec3_t v, vec3_t out)
setzt den Vektor out auf den normalisierten Vektor von v und gibt die Länge von v zurück

void VectorInverse (vec3_t v)
invertiert den Vektor v ($v = -v$)

void VectorNegate (vec3_t v, vec3_t out)
setzt den Vektor out auf den Wert des invertierten Vektors v

void VectorScale (vec3_t in, vec_t scale, vec3_t out)
setzt den Vektor out auf $\mathsf{scale} * \mathsf{in}$

void VectorMA (vec3_t veca, float scale, vec3_t vecb, vec3_t vecc)
setzt den Vektor vecc auf $\mathsf{veca} + \mathsf{scale} * \mathsf{vecb}$

void VectorAdd (vec3_t veca, vec3_t vecb, vec3_t out)
setzt den Vektor out auf $\mathsf{veca} + \mathsf{vecb}$

void VectorSubtract (vec3_t veca, vec3_t vecb, vec3_t out)
setzt den Vektor out auf $\mathsf{veca} - \mathsf{vecb}$

vec_t DotProduct (vec3_t v1, vec3_t v2)
gibt das Skalarprodukt (innere Produkt) der beiden Vektoren v1 und v2 zurück ($= \|\mathsf{v1}\| * \|\mathsf{v2}\| * \cos(\mathsf{v1}, \mathsf{v2})$)

void CrossProduct (vec3_t v1, vec3_t v2, vec3_t cross)
setzt den Vektor cross auf das Vektorprodukt (äußere Produkt) der beiden Vektoren v1 und v2 (cross ist senkrecht zu v1 und v2 mit $\|\mathsf{cross}\| = \|\mathsf{v1}\| * \|\mathsf{v2}\| * |\sin(\mathit{Winkel}(\mathsf{v1}, \mathsf{v2}))|$)

float *tv (float x, float y, float z)
Diese Funktion erzeugt einen temporären Vektor mit den Komponenten x, y und z und gibt diesen zurück; dies ist eine einfache Möglichkeit, temporäre Vektoren z.B. für Funktionsaufrufe zu erzeugen; hier werden acht verschiedene Plätze für solche Vektoren angeboten, d.h. ein Vektor wird nach achtmaligem Aufruf dieser Funktion durch einen anderen Vektor ersetzt.

char *vtos (vec3_t v)
Diese Funktion erzeugt einen temporären String, in den die (in Integer umgewandelten) Werte des Vektors v eingetragen werden, welcher dann auch zurückgegeben wird; dies ist eine einfache Möglichkeit, um Vektoren auszudrucken; für die temporäre Eigenart gilt das Gleiche wie bei tv().

Bounding Box

Die Bounding Box eines Entities ist der Raum, der von Parallelen zu den Welt-Koordinatenachsen durch die beiden Punkte ent→mins und ent→maxs eingeschlossen wird. Bei Solid Entities (i.d.R. SOLID_BSP oder SOLID_TRIGGER) sind diese Punkte schon durch die Größe und Lage des Brushes im Level-Editor definiert; bei Point-Entities, die mit der Welt interagieren sollen (i.d.R. SOLID_TRIGGER oder SOLID_BBOX), wie z.B. den Models, müssen Werte für diese vor dem Erzeugen der Entities eingetragen werden. Ändert sich irgendwann die Orientierung des Entity, also ent→s.angles $\neq$ vec3_origin, so dreht sich zwar das Objekt im Spiel, jedoch wird die Bounding Box nicht automatisch mitgedreht, so daß diese manuell angepaßt werden muß, damit die Kollisionserkennung noch richtig funktioniert. Man kann aber auch von Beginn an eine symmetrische Ausdehnungen der Bounding Box bezüglich der möglichen Drehachsen wählen (wie z.B. beim Spieler-Entity: ent→mins= $(-16,-16,-24)$ und ent→maxs= $(16,16,32)$), um diesem Dilemma auf einfache Weise zu entgehen.

int BoxOnPlaneSide (vec3_t emins, vec3_t emaxs, cplane_t *p)

Diese Funktion bedeutet wahrscheinlich: Wenn die durch emins und emaxs definierte Bounding Box über der Oberseite der durch p definierten Ebene liegt, ist das erste Bit des Rückgabewertes gesetzt, wenn sie unter der Unterseite liegt, das zweite (dabei ist die Oberseite diejenige, von der die Ebenennormale wegzeigt).

void ClearBounds (vec3_t mins, vec3_t maxs)

initialisiert den Minimumvektor mins und den Maximumvektor maxs, indem die Komponenten von mins auf +99999 und die von maxs auf -99999 gesetzt werden

void AddPointToBounds (vec3_t v, vec3_t mins, vec3_t maxs)

Wenn der Punkt v außerhalb der durch mins und maxs definierten Bounding Box liegt, wird die Bounding Box, also mins und maxs, derart angepaßt, daß der Punkt v danach innerhalb dieser liegt.

Geometrische Transformationen

void RotatePointAroundVector (vec3_t dst, const vec3_t dir, const vec3_t point, float degrees)

Der Punkt point wird um degrees° um den Vector dir (sollte normalisierter Vektor sein) gedreht und das Ergebnis im Punkt dst gespeichert.

void ProjectPointOnPlane (vec3_t dst, const vec3_t p, const vec3_t normal)

projiziert den Punkt p auf die durch die Normale normal (sollte normalisierter Vektor sein) definierte Ebene durch den Nullpunkt und speichert das Ergebnis im Punkt dst

void PerpendicularVector (vec3_t dst, const vec3_t src)
Im Ergebnisvektor dst wird ein zum Vektor src (sollte normalisierter Vektor sein) senkrechter, normalisierter Vektor gespeichert.

void R_ConcatRotations (float in1[3][3], float in2[3][3], float out[3][3])
Die beiden durch die Drehmatrizen in1 und in2 definierten Drehungen werden miteineinander verkettet und in der Ergebnisdrehmatrix out gespeichert.

void R_ConcatTransforms (float in1[3][4], float in2[3][4], float out[3][4])
Die beiden durch die Transformationsmatrizen in1 und in2 definierten Transformationen werden miteinander verkettet und in der Ergebnistransformationsmatrix out gespeichert.

Info-Strings

Info-Strings sind Strings, die eine Folge von Schlüsseln und Werten von Cvars beinhalten (Name-Wert-Paare). In der game dll taucht an verschiedenen Stellen ein Info-String auf, der die Cvars mit dem Flag CVAR_USERINFO enthält. Dieser hat den Namen userinfo. Mit den folgenden Funktionen können Info-Strings auf einfache Weise benutzt werden:

char *Info_ValueForKey (char *s, char *key)
durchsucht den Info-String s nach dem Schlüssel key und gibt dessen Wert als String zurück; sollte kein Eintrag existieren, wird ein leerer String zurückgegeben

void Info_RemoveKey (char *s, char *key)
durchsucht den Info-String s nach dem Schlüssel key und entfernt das Schlüssel-Wert-Paar aus dem Info-String

qboolean Info_Validate (char *s)
Einige Zeichen sind in einem Info-String nicht erlaubt, da sie das Parsen des Servers durcheinander bringen würden (" und ;). Diese Funktion liefert true, wenn diese Character nicht im Info-String auftauchen, ansonsten false.

void Info_SetValueForKey (char *s, char *key, char *value)
Diese Funktion fügt das Schlüssel-Wert-Paar key und value in den Info-String s ein; es sei denn sie enthalten nicht erlaubte Character wie ", ; oder \ (wird als Trennzeichen der Schlüssel und Werte benutzt), oder sie sind größer als die maximal erlaubte Länge für Werte bzw. Schlüssel (MAX_INFO_KEY bzw. MAX_INFO_VALUE), oder der Info-String wird insgesamt länger als maximal erlaubt (MAX_INFO_STRING), oder der Wert value existiert gar nicht.

Items

gitem_t *FindItem (char *pickup_name)
gibt das Item mit dem übergebenen pickup_name aus der Itemliste zurück

gitem_t *FindItemByClassname (char *classname)
wie FindItem(), nur daß hier der classname des Items ausschlaggebend ist

int ITEM_INDEX (gitem_t *item)
gibt den Index des Items item in der Itemliste zurück

gitem_t *GetItemByIndex (int index)
gibt das Item mit dem Index index aus der Itemliste zurück

Sonstiges

float random ()
gibt eine Zufallszahl zwischen 0 und 1 zurück

float crandom ()
gibt eine Zufallszahl zwischen -1 und 1 zurück

qboolean KillBox (edict_t *ent)
Es wird versucht, alle Entities, die innerhalb der (zukünftigen) Bounding Box des (noch nicht mit der Welt verbundenen!) Entities ent liegen, mit Hilfe von T_Damage() zu zerstören. Gelingt dies, wird true zurückgegeben, ansonsten false.

void G_ProjectSource (vec3_t point, vec3_t distance, vec3_t forward, vec3_t right, vec3_t result)
Das Ergebnis result errechnet sich aus $\mathsf{point} + \mathsf{distance}[0] * \mathsf{forward} + \mathsf{distance}[1] * \mathsf{right} + \mathsf{distance}[2] * (0,0,1)$. Der Punkt point wird also um den Faktor distance[0] nach forward, um den Faktor distance[1] nach right und um den Faktor distance[2] senkrecht nach oben verschoben und das Ergebnis in result gespeichert.

edict_t *G_Find (edict_t *from, int fieldofs, char *match)
Diese Funktion sucht, beginnend beim Entity from (oder, wenn dieses als NULL übergeben wird, bei g_edicts), das nächste aktive Entity, das den String match in dem Entity-Feld mit dem Offset fieldofs (mit Hilfe von FOFS() S. 160 beim Funktionsaufruf übergeben) stehen hat. Dieses Entity (bzw. NULL) wird dann zurückgegeben.

edict_t *findradius (edict_t *from, vec3_t org, float rad)

Diese Funktion sucht, ebenfalls beim Entity from beginnend wie G_Find(), das nächste aktive Entity mit ent→solid≠SOLID_NOT, dessen Mittelpunkt (Mitte der Bounding Box) innerhalb der Kugel um den Punkt org mit dem Radius rad liegt. Dieses Entity (bzw. NULL) wird dann zurückgegeben.

edict_t *G_PickTarget (char *targetname)

Es wird zufällig eines der ersten acht Entities ausgewählt, die targetname als ent→targetname besitzen. Dieses (bzw. NULL) wird dann zurückgegeben.

void G_UseTargets (edict_t *ent, edict_t *activator)

Sollte ent→delay gesetzt sein, werden die folgenden Aktionen um diese Zeitspanne verzögert ausgeführt. Ist ent→message vorhanden, so wird auf dem Bildschirm des activator diese Nachricht ausgegeben. Existiert ent→killtarget, werden alle Entities mit diesem als other→targetname aus dem Level entfernt; existiert ent→target, so werden zuletzt noch alle Entities mit diesem als other→targetname benutzt über deren Funktion other→use (falls vorhanden) unter Angabe von ent als 2. und activator als 3. Argument.

void G_SetMovedir (vec3_t angles, vec3_t movedir)

Mit dieser Funktion werden die im Orientierungsvektor angles stehenden Werte, die i.d.R. durch die Übergabe des angle-Wertes aus dem Level-Editor entstanden sind, in einen Richtungsvektor movedir umgewandelt und dort eingetragen. Danach wird angles auf (0,0,0) gesetzt.

edict_t *G_Spawn (void)

Es wird ein freies oder noch nicht benutztes Entity zurückgegeben (falls keine Entities mehr erlaubt sind, bricht das Spiel ab); dabei wird dieses folgendermaßen initialisiert: ent→inuse auf true, ent→classname auf „noclass“, ent→gravity auf 1.0 und ent→s.number auf den Index des Entities in g_edicts.

void G_FreeEdict (edict_t *ent)

Das Entity ent wird aus der Welt entfernt (gi.unlinkentity()) und alle Inhalte der Entity-Struktur auf 0 gesetzt; danach wird noch ent→classname auf „freed“, ent→freetime auf level.time und ent→inuse auf false gesetzt.

void G_TouchTriggers (edict_t *ent)

Diese Funktion läßt das Entity ent alle Trigger berühren, die sich innerhalb seiner Bounding Box befinden.

void G_TouchSolids (edict_t *ent)

Diese Funktion zwingt alle Entities dazu den (i.d.R. soeben neu eingesetzten) Trigger ent zu berühren, falls sie sich innerhalb dieses Triggers befinden.

qboolean OnSameTeam (edict_t *ent1, edict_t *ent2)

gibt true zurück, wenn die beiden Spieler ent1 und ent2 im selben Team spielen (im DM, z.B. bei CTF), ansonsten false

void T_Damage (edict_t *targ, editct_t *inflictor, edict_t *attacker, vec3_t dir, vec3_t point, vec3_t normal, int damage, int knockback, int dflags, int mod)

Mit dieser Funktion sollte jeglicher Schaden an Entities erzeugt werden; nirgendwo sonst sollte ent→health eines Entities erniedrigt werden! targ ist das Entity, das den Schaden abbekommen soll, inflictor ist das Entity, das den Schaden auslöst (z.B. die Rakete), attacker ist das Entity, das für den Schaden verantwortlich ist (z.B. der Spieler), dir ist die Richtung, in die der Rückstoß wirkt, point ist der genaue Punkt, in dem der Schaden auftritt, normal ist die Normale der Oberfläche an diesem Punkt, damage ist die Größe des Schadens, knockback ist ein Maß für die Stärke des Rückstoßes, mod bezeichnet „Means of Death", also die Todesart, falls das Entity den Schaden nicht überlebt (die entsprechenden Werte hierzu sind in `g_local.h` als Konstanten MOD_* definiert), dflags sind spezielle Schadens-Flags, die die Wirkung des Schadens beeinflussen können:

DAMAGE_RADIUS	(1) der Schaden ist indirekt entstanden
DAMAGE_NO_ARMOR	(2) Rüstung schützt nicht vor diesem Schaden
DAMAGE_ENERGY	(4) Schaden ist aufgrund einer energiebasierten Waffe entstanden
DAMAGE_NO_KNOCKBACK	(8) der Schaden löst keinen Rückstoß aus, sondern nur eine Sichtwinkeländerung
DAMAGE_BULLET	(16) Schaden ist von einer Kugel (Munition: Bullets)
DAMAGE_NO_PROTECTION	(32) es gibt keinen Schutz vor diesem Schaden (auch nicht godmode o.ä.)

void T_RadiusDamage (edict_t *inflictor, edict_t *attacker, float damage, edict_t *ignore, float radius, int mod)

Diese Funktion fügt allen Entities außer ignore innerhalb des Radius radius einen mit der Entfernung skalierten Schaden damage zu. inflictor, attacker und mod werden dabei mit an T_Damage() übergeben, so daß deren Bedeutung dort zu finden ist (hier wird z.B. dflags beim Aufruf von T_Damage() auf DAMAGE_RADIUS gesetzt).

void BecomeExplosion1 (edict_t self)

Das Entity self explodiert; das temporäre Entity TE_EXPLOSION1 wird erzeugt und das Entity self mit G_FreeEdict() entfernt (es gibt auch noch die Funktion BecomeExplosion2(), die genauso wirkt, nur TE_EXPLOSION2 benutzt, diese ist aber nur in `g_misc.c` definiert).

B.3.3 Programmablauf

Wird auf einem Rechner quake2.exe ausgeführt, so wird als erstes der Client geladen (z.B. landet man durch Aufruf quake2.exe + sofort an der Client-Konsole, ohne daß der Server gestartet wird). Der Client ist sozusagen nur eine „dumme leere grafische Oberfläche“, der diejenigen Entities und Brushes auf dem Bildschirm darstellt, die der Server ihm übermittelt. Um an einem Spiel teilnehmen zu können, muß der Client mit einem Server verbunden werden, auf dem ein Spiel gestartet werden kann oder schon gestartet ist. Der einfachste Weg hierfür ist lokal ein Spiel zu starten, z.B. mit dem Konsolenbefehl map x. Es wird dann automatisch auf dem lokalen Rechner ein Server gestartet und der Client mit diesem verbunden. Eine andere Möglichkeit besteht darin, z.B. über den Befehl connect IP_ADRESSE, den Client mit einem Server auf einem entfernten Rechner mit der IP-Adresse IP_ADRESSE zu verbinden. Ist der Client mit einem Server verbunden, kann die Client-Spielfigur am Client-Rechner gesteuert werden.
Für uns nun interessanter ist der Startvorgang des Servers. Dieser wird entweder automatisch von einem Client aus gestartet (wie oben beschrieben), oder als alleinstehender Server ohne Client mit einer Kommandozeile der Art quake2 +set dedicated 1. Im folgenden werden die Bereiche Startvorgang eines Spiels, Ablauf eines Client-Frames, Ablauf eines Server-Frames, Speicherung eines Spiels und Spielende näher beschrieben. Die Funktionserläuterungen erfolgen schrittweise, wobei die Befehlsfolgen eingerückt und mit $\hookrightarrow$ eingeleitet werden; bei weiteren Funktionsaufrufen wird nochmals eingerückt, wobei vor einer Einrückung die im folgenden erklärte `Funktion` derart hervorgehoben wird. Desweiteren werden viele Variablennamen gebraucht, dabei gilt: ent, self oder other bezeichnen Entity-Strukturen vom Typ edict_t; g_edicts, game, level, gi, globals und st bezeichnen die so in `g_main.c` definierten Variablen (S. 83); client ist eine Abkürzung für ent→client, pers ist eine Abkürzung von ent→client→pers und resp kürzt ent→client→resp ab; alle anderen Bezeichner der untersten Ebene sind Konsolenvariablen (Cvars) wie z.B. timelimit, dmflags, skill, deathmatch, ... Funktionen werden durch Anhängen von „()“ als solche kenntlich gemacht. Sollte eine erklärte Funktion Parameter besitzen, so beziehen sich folgende Bezeichner vornehmlich auf diese Parameter. Ausnahmen hiervon werden gekennzeichnet.

Spielbeginn

Zu Beginn eines Spieles wird zuerst der Server gestartet. Dieser lädt dann alle von ihm benötigten Resourcen wie z.B. (unter Win32) die `winsock.dll` und natürlich die `gamex86.dll`. Der Server lädt diejenige game dll, die sich in dem Verzeichnis befindet, welches in der Konsolenvariable (S. 121, 83) game angegeben ist (wenn dort keine game dll gefunden wird, die aus dem Verzeichnis `baseq2`). Dabei ruft der Server dann die Funktion game_export_t *GetGameAPI(game_import_t *import) der game dll auf. Hierbei übergibt der Server der game dll einen Zeiger auf eine Struktur des Typs game_import_t (S. 113), deren Informationen in gi gespeichert werden; die Struktur globals vom Typ game_export_t (S. 130) wird zusammengestellt und ein Zeiger darauf an den Server zurückgegeben. Hiermit wird also die

Server-game dll Schnittstelle initialisiert. Hieran ist zu erkennen, daß der Server nicht alle Informationen der game dll kennt. Sein Wissen über die game dll bezieht er ausschließlich über die bei der game engine-Kompilierung beinhalteten Dateien `game.h` und `q_shared.h`, sowie aus der Struktur globals.
Direkt nach der Schnittstelleninitialisierung wird dann die Funktion globals.Init() (=InitGame()) aufgerufen (Bildschirmausgabe „==== InitGame ====“). Dort werden alle in der game dll definierten Konsolenvariablen (Konsole S. 83), die restlichen Teile der Struktur globals sowie große Teile der Struktur game (vom Typ game_locals_t (S. 87)) initialisiert. Insbesondere wird hier auch der Speicherbereich für die Entity- und Client-Strukturen (vom Typ edict_t bzw. gclient_t) allokiert. Sollte es sich hier um ein gespeichertes Spiel handeln, so wird noch globals.ReadGame() (Speicherung S. 160) aufgerufen. Nun ist das Spiel initialisiert und der Level kann geladen werden.

Laden des Levels

Der hier beschriebene Ladevorgang eines Levels findet bei jedem Levelwechsel im Spiel statt; es erscheint die Bildschirmausgabe „------ Server Initialisation ------“. Wenn ein neuer Level geladen werden soll, wird zuerst die Funktion `void SpawnEntities(char *mapname, char *entities, char *spawnpoint)` aufgerufen (dies ist der nullte Server-Frame):

- ↪ Einige Daten der Client-Entities, die bei einem Levelwechsel konstant bleiben sollen, werden mit Hilfe der Funktion void SaveClientData(void) in den jeweiligen Client-Strukturen zwischengespeichert (ent→health, ent→max_health sowie ent→flags (ohne FL_GODMODE, FL_NOTARGET und FL_POWER_ARMOR) werden in pers.health, pers.max_health und pers.savedFlags gespeichert).
- ↪ Alle Level-bezogenen Speicherbereiche (mit Tag TAG_LEVEL) werden freigegeben.
- ↪ Sämtliche Daten aller Entity-Strukturen g_edicts und der Struktur level werden gelöscht.
- ↪ Die übergebenen Strings mapname und spawnpoint werden soweit wie möglich in die Struktur level übernommen.
- ↪ Die Zeiger ent→client der Client-Entities werden auf die Client-Strukturen gesetzt.
- ↪ Die Informationen über die restlichen Entities des Levels werden „Entity für Entity“ aus dem übergebenen String entities gelesen und in freie Entity-Strukturen gespeichert. Dabei müssen alle Felder des Strings in der Liste field_t fields[] aufgeführt sein, damit sie auch verwendet werden können, d.h. sollen bei eigenen Erweiterungen neue Felder aus dem Level-Editor übernommen werden, so müssen diese in der Liste fields[] aufgeführt werden (siehe auch Speicherung S. 160). Entities, die aufgrund ihrer ent→spawnflags und dem gewählten „Schwierigkeitsgrad“ nicht erscheinen sollen, werden dabei ignoriert. Dann wird die jeweilige individuelle Entity-Erzeugungsfunktion aufgerufen (in der Liste spawns[] in g_spawn.c sind Tupel des Typs spawn_t aufgeführt,

bestehend aus dem Entity-Klassennamen ent→classname und einem Funktionszeiger auf die zugehörige Erzeugungsfunktion void (*spawn)(edict_t *ent)). ⇒ Beachte: während der Ausführung der Entity-Erzeugungsfunktionen sind noch nicht alle Entities des Levels vorhanden! Also müssen Initialisierungsaufgaben, die noch von anderen Entities abhängen, auf den ersten Frame verlegt werden; jedoch sollten diese am Ende des zweiten Frames beendet sein.

↪ Die Anzahl der ignorierten Entities wird ausgegeben (Bildschirmausgabe „x entities inhibited").

↪ Die Entity-Teams werden gebildet: Alle Entities mit dem gleichen Team-String ent→team werden „aneinandergekettet"; das erste Entity eines Teams ist der Teammaster, alle anderen haben das Flag FL_TEAMSLAVE gesetzt (ent→flags); alle Entities des Teams haben einen Zeiger ent→teammaster auf den Teammaster; jedes Entity hat einen Zeiger ent→teamchain auf das folgende Entity im Team (der Teamletzte hat hier NULL). Welches Entity an welcher Stelle der Teamchain steht, ist mehr oder weniger zufällig und vom Level-Editor aus nicht beeinflußbar.

↪ Die Anzahl der Teams und der daran beteiligten Entities wird ausgegeben (Bildschirmausgabe „x teams with y entities").

↪ Die Player-Trail-Entities werden initialisiert (nicht im DM).

Nachdem nun alle Entities initialisiert sind, laufen erst einmal zwei Server-Frames ab (S. 148), damit auch alle Entities vollständig im Spiel sind (z.B. wenn individuelle Erzeugungsfunktionen Initialisierungsarbeit auf den ersten oder sogar zweiten (Server-)Frame verlegen, weil z.B. Teams im nullten Frame noch nicht vorhanden sind). Sollte es sich um ein manuell gespeichertes Spiel handeln, wird nun globals.ReadLevel() ausgeführt (Speicherung S. 160). Abschließend erscheint noch die Bildschirmausgabe „------------------". Jetzt wird globals.WriteGame() aufgerufen und das Spiel automatisch gespeichert, jedoch sinnvollerweise nur, falls dies kein geladener Spielstand ist. Nun hat der Server den Level gestartet, so daß im folgenden alle 0.1 s (=FRAMETIME) ein Server-Frame (S. 148) ausgeführt wird (bei den nun folgenden Ladevorgängen werden diese mehr oder weniger verzögert).

Clients verbinden

Nun, da der Server gestartet und der Level initialisiert ist, fehlen nur noch die Clients. Wird ein neues Spiel begonnen, so werden nun die Clients mit dem Server verbunden. Dazu ruft der Server globals.ClientConnect() auf. Anderenfalls, wenn „nur" ein Levelwechsel stattfindet, wird die schon bestehende Verbindung zu den Clients wieder „aktiviert", so daß das folgende übersprungen und gleich zum Abschnitt „Clients starten" (S. 145) übergegangen werden kann. Die Funktion `qboolean ClientConnect(edict_t *ent, char *userinfo)` hat einen Rückgabewert des Typs qboolean, durch den die game dll dem Server mitteilt, ob einem Client der Zutritt zu diesem Spiel gestattet wird. Der Paramter ent zeigt auf das Client-Entity, das für diesen Client im positiven Fall verwendet werden soll. Der Parameter userinfo ist ein String, der alle mit dem Flag CVAR_USERINFO markierten Cvars des Clients (z.B. name, spectator, ip) als Name/Wert Paar beinhaltet; diese Teilstrings werden der besseren Handhabbarkeit wegen im folgenden über userinfo→name etc. angesprochen, obwohl dies syntaktisch nicht korrekt ist und hierzu eigens Funktionen existieren (Info-Strings S. 137). ClientConnect() verläuft folgendermaßen:

- ↪ Ist die IP-Adresse des Clients (userinfo→ip) auf dem Server nicht zugelassen (IP-Filtering S. 132), wird false zurückgegeben und dem Client der Zutritt zum Spiel verwehrt.
- ↪ Ist dies ein DM-Spiel und der Client möchte als Zuschauer beitreten (userinfo→spectator $\neq$ 0), so wird der Wert userinfo→spectator mit der Server-Cvar spectator_password verglichen sowie die maximale Zuschaueranzahl überprüft (Server-Cvar maxspectators); ist dabei etwas nicht in Ordnung, wird false zurückgegeben.

  Ansonsten (normales Einzel- oder DM-Spiel) wird geprüft, ob ein Paßwort benötigt wird (Server-Cvar password), und wenn ja, ob der Client (userinfo→password) das korrekte eingegeben hat; bei falschem Paßwort wird false zurückgegeben.
- ↪ Hat alles geklappt, so kann die Verbindung aufgenommen und der Entity-Client-Zeiger ent→client auf eine Client-Struktur gesetzt werden.
- ↪ Ist das Entity ent nicht in Gebrauch (z.B. bei einem neuen Spiel), so wird die Struktur ent→client→resp initialisiert mit Hilfe von `void InitClientResp(gclient_t *client)`:
  - ↪ resp Struktur wird gelöscht.
  - ↪ resp.enterframe auf level.framenum
  - ↪ resp.coop_respawn auf pers
- ↪ War dies ein manuell gespeichertes Spiel oder hat der Spieler noch keine Waffe in der Hand (z.B. bei einem neuen Spiel), wird auch noch die Struktur ent→client→pers initialisiert mit Hilfe von `void InitClientPersistant(gclient_t *client)`:
  - ↪ pers Struktur wird gelöscht.
  - ↪ Waffe Blaster wird dem Inventar hinzugefügt und in die Hand gelegt.
  - ↪ Initialer Gesundheitszustand wird festgelegt.
  - ↪ pers.max_∗-Werte werden festgelegt.
  - ↪ pers.connected auf true

  Ansonsten handelt es sich um einen geladenen Spielstand und ein „alter Körper“ wird einfach ohne weitere Initialisierung übernommen.
- ↪ Anschließend werden die Userinfo-Daten in die Client-Struktur mit Hilfe von `void ClientUserinfoChanged(edict_t *ent, char *userinfo)` eingetragen:

- ↪ String userinfo wird auf Korrektheit überprüft.
- ↪ pers.netname wird auf userinfo→name gesetzt.
- ↪ Je nachdem, ob der Spieler ein Zuschauer ist oder nicht, wird pers.spectator zu true oder false.
- ↪ Spielername und -skin werden in den Configstring CS_PLAYERSKINS + Spielernummer (S. 124) geschrieben.
- ↪ Spielersichtfeld wird angepaßt, indem client→ps.fov auf userinfo→fov gesetzt wird, sofern wir nicht im DM sind und dies durch DM-Flags verboten ist.
- ↪ Die Hand, in der die Waffe liegt, wird angepaßt (userinfo→hand).
- ↪ userinfo wird in pers.userinfo gespeichert.

Diese Funktion (globals.ClientUserinfoChanged()) wird vom Server ansonsten immer dann aufgerufen, wenn ein Spieler eine seiner Userinfo-Cvars ändert.

- ↪ Auf der Serverkonsole wird entsprechend mitgeteilt: „pers.netname connected“.
- ↪ ent→svflags auf default 0
- ↪ pers.connected auf true (evtl. schon in InitClientPersistant() geschehen).
- ↪ true wird zurückgegeben.

Hat der Server nun true zurückbekommen, so kann er den Client wie im folgenden beschrieben starten.

Clients starten

Sind alle Clients verbunden (bzw. wieder „aktiviert“) (Bildschirmausgabe: Querbalken, Levelname in grüner Schrift), werden die levelspezifischen Texturen, Models, ... an die Clients geschickt, sofern diese dort noch nicht vorhanden sind (und deren Konsolenvariable(n) allow_download(_∗) nicht auf 0 steht (stehen)). Ist das erledigt, wird jeder Client mit `void ClientBegin(edict_t *ent)` ins Spiel gesetzt:

- ↪ client wird auf die zugehörige Client-Struktur gesetzt.
- ↪ Ist ent→inuse auf true, so ist dies ein geladener Spielstand und ein „alter Körper“ steht schon bereit, welcher dann übernommen wird (im DM nicht).

  Anderenfalls (im DM immer) wird das Entity mit `G_InitEdict(edict_t *e)` initialisiert:

  - ↪ e→inuse auf true
  - ↪ e→classname auf „noclass“
  - ↪ e→gravity auf 1.0
  - ↪ e→s.nummer auf Entityindex

Dann wird ent→classname auf „player“ gesetzt und InitClientResp() (s.o.) als auch `PutClientInServer(edict_t *ent)` ausgeführt:

- ↪ Erzeugungspunkt (Punkt, an dem der Spieler das Level betritt) wird gesucht.
- ↪ Sind wir im DM, werden alle Client-Daten bis auf resp und die Userinfo gelöscht. Mit InitClientPersistant() und ClientUserinfoChanged() wird pers wieder initialisiert.

  Sind wir im Coop, werden ebenfalls alle Client-Daten bis auf resp und die Userinfo gelöscht, wobei aber pers.game_helpchanged und pers.helpchanged noch in resp.coop_respawn gespeichert werden. Anschließend wird pers auf den Zustand in resp.coop_respawn gesetzt, mit ClientUserinfoChanged() die Userinfo eingetragen und pers.score wenn nötig angepasst.

  Ansonsten werden alle Client-Daten außer pers gelöscht.
- ↪ Sollte der Spieler tot sein, wird InitClientPersistant() ausgeführt.
- ↪ Mit FetchClientEntData() werden einige Daten von pers in die Entity-Struktur kopiert (Umkehrung von SaveClientData(): health, max_health, savedFlags); bei Coop wird noch resp.score an pers.score angeglichen.
- ↪ Anschließend werden große Teile der Entity- und der Client-Struktur initialisiert …(siehe Quelltext).
- ↪ Alle Entities im Erzeugungsbereich der Client-Spielfigur werden entfernt bzw. getötet.
- ↪ Das Client-Entity wird in die Welt gesetzt (gi.linkentity() S. 125)
- ↪ Die aktivierte Waffe wird in die Hand genommen.
- ↪ Sollte die Intermission gerade laufen, wird der Client mit MoveClientToIntermission() ebenfalls dorthin gebracht.

  Ansonsten wird (sofern nicht im Einspielermodus) noch der Effekt MZ_LOGIN (Effekte S. 114) gesendet und allen Mitspielern mitgeteilt: „pers.netname entered the game“.
- ↪ Dann wird noch ClientEndServerFrame() aufgerufen, damit die Client-Sicht angepaßt wird; dazu aber mehr im Abschnitt Server-Frame (S. 148).

Nun sind die Clients im Spiel und jeder Client hat seine eigene Sicht auf das Geschehen. Von nun an wird je nach Leistung des jeweiligen Client-Rechners, alle x Sekunden ein Client-Frame ausgeführt. Die Clients schicken also $\frac{1}{x}$-mal in der Sekunde den Eingabegerätezustand an ihren Rechnern (sprich: das, was der Spieler ausgelöst hat) an den Server, welcher dann für diesen Client globals.ClientThink() aufruft und dem Client den neuen Stand der Dinge zurücksendet (Die Client-Server Kommunikation wird von der game engine ausgeführt, so daß man aus der game dll heraus weder Einsicht noch Einfluß auf diesen Bereich des Spiels hat). I.d.R. sind die Client-Frames häufiger als die Server-Frames. Die Client-Frames sollten im Schnitt nicht allzu weit unter 15 Stück pro Sekunde fallen, da einem sonst der Spaß am Spiel schnell vergeht; nach oben sind hier keine Grenzen gesetzt, je mehr desto besser. Die Server-Frames sind wie schon erwähnt mit 10 Stück pro Sekunde konstant.

Ablauf eines Client-Frames

Der Client-Frame behandelt vor allem die Bewegung und direkte Aktionen des Clients wie Schießen. Wie schon erwähnt, schickt jeder Client ungefähr 10-100 mal in der Sekunde dem Server ein Paket mit den Eingaben des Benutzers (viele in diesem Abschnitt auftauchenden Datenstrukturen sind bei der Funktion gi.Pmove() S. 122 erklärt). Daraufhin ruft der Server die game dll-Funktion `void ClientThink(edict_t *ent, usercmd_t *ucmd)` mit dem Entity des Clients als Parameter ent und dessen Benutzereingaben als Parameter ucmd auf:

- ↪ level.current_entity auf ent
- ↪ Wenn wir gerade in der Intermission sind, wird client→ps.pmove.pm_type auf PM_FREEZE gesetzt. Falls wir schon mehr als 5 s in der Intermission sind und der Spieler einen Knopf oder eine Taste gedrückt hat (Bit BUTTON_ANY in ucmd→buttons gesetzt), kann die Intermission demnächst verlassen werden (level.exitintermission auf true). Client Think ist hiermit beendet.
- ↪ Ist der Spieler ein Zuschauer, wird sein Sichtänderungswunsch (ucmd→angles) in resp→.cmd_angles kopiert.

  Ansonsten wird die Spielerbewegung vorbereitet, ausgeführt und ausgewertet: Dazu werden die Eingabedaten einer pmove_t-Sruktur pm gesetzt und die Engine-Funktion gi.Pmove() damit als Parameter aufgerufen. Dann werden die Ausgabedaten von pm in die Entity- und die Client-Struktur übernommen. Wie auch im Fall eines Zuschauers wird der Sichtänderungswunsch gespeichert. Ein Sprunggeräusch wird ausgegeben, falls bisher ein ent→groundentity vohanden war, jetzt aber nicht mehr und sich der Spieler sich (nicht im Wasser) nach oben bewegt. Falls der Spieler tot ist, wird sein Blick (client→ps.viewangles) „verdreht“, ansonsten der Blick von pm übernommen. Dann wird das Client-Entity wieder mit der Welt verbunden (gi.linkentity()). Sofern ent→movetype nicht MOVETYPE_NOCLIP ist, werden mit Hilfe von G_TouchTriggers() alle Trigger im Bereich des Entities berührt, d.h. falls diese eine touch()-Funktion besitzen, werden sie (self) mit (self, ent, NULL, NULL) als Parameter aufgerufen. Dann werden noch alle in der Liste pm.touchents aufgeführten Entities auf die gleiche Weise berührt.
- ↪ Die Knopfdrücke (ucmd→buttons) und deren Veränderungen bezüglich des letzten Client-Frames, sowie die Helligkeit am Standort des Clients (ucmd→light_level) werden gespeichert.
- ↪ Wurde der Angriffs-Button (Buttons-Flag BUTTON_ATTACK) gedrückt: Im Zuschauermodus wird mit GetChaseTarget() ein Spieler in der Umgebung gesucht, dem der Blick dieses Clients dann folgt bzw., wenn im Moment gerade jemandem gefolgt wird,

die Verfolgung abgebrochen. Im normalen Spielmodus wird die Waffe abgefeuert, falls client→weapon_thunk auf false steht (ansonsten ist dies schon in einem früheren Client-Frame geschehen und zwischen diesem und jetzt fand noch kein Server-Frame statt). Dies geschieht durch den Aufruf von `void Think_Weapon(edict_t *ent)`, wo auch die Waffen-Frames geregelt werden:

- ↪ Ist der Spieler gerade gestorben, so wird seine Waffe aus dem Sichtfeld entfernt (client→newweapon auf NULL und ChangeWeapon() aufrufen).
- ↪ Es wird festgestellt, ob Quad Damage oder Silencer aktiv sind und dann die Waffen-Think-Routine der aktuellen Waffe pers.weapon→weaponthink() aufgerufen. Dort werden normalerweise zwei statische Integer-Listen pause_frames und fire_frames definiert, welche dann als Parameter für den Funktionsaufruf void WeaponGeneric(edict_t *ent, int FRAME_ACTIVATE_LAST, int FRAME_FIRE_LAST, int FRAME_IDLE_LAST, int FRAME_DEACTIVATE_LAST, int *pause_frames, int *fire_frames, void (*fire)(edict_t *ent)) dienen. In WeaponGeneric() wird dann abhängig vom aktuellen Waffenstatus client→weaponstate und vom momentanen Waffenframe client→ps.gunframe die Waffenanimation des Clients durchgeführt (also client→weaponstate und client→ps.gunframe angepaßt) und im Falle eines gedrückten Angriffsknopfes die übergegebene Schußfunktion ausgeführt (das kann nur hier geschehen, beim Aufruf von ThinkWeapon() im Server-Frame (s.u.) jedoch nicht).

↪ Im Zuschauermodus wird dann noch überprüft, ob der Spieler gerade springen will (ucmd→upmove $\geq$ 10) und dabei client→ps.pmove.pm_flags angepaßt. Wenn ja, wird mit ChaseNext() ein anderer Spieler zum Folgen gesucht bzw., falls noch keinem gefolgt wird, mit GetChaseTarget() einer zum Folgen gesucht.

↪ Zum Schluß wird noch der Bildausschnitt aller diesem Spieler folgenden mit UpdateChaseCam() angepaßt.

Damit ist der Client-Frame beendet und der Server übermittelt dem Client den neuen Zustand, so daß dieser sein Bild darstellen kann.

Ablauf eines Server-Frames

Der Server-Frame ist im Vergleich zum Client-Frame recht umfangreich. Hier wird das Verhalten aller nicht Client-Entities behandelt. Wie schon erwähnt, führt der Server genau 10-mal pro Sekunde einen solchen Server-Frame aus. Zumindest dreht er pro Aufruf der Funktion `void G_RunFrame(void)` die Uhr der 3D-Welt um 0.1 s weiter, egal wie schnell die Zeit in der realen Welt verrinnt. In diesem Abschnitt bezeichnet Frame wohlgemerkt immer den Server-Frame.

↪ level.framenum plus eins

↪ level.time plus FRAMETIME

- Mit der Funktion AI_SetSightClient() wird ein Client ausgewählt, den die Monster in diesem Frame sehen können; dieser wird dann für diesen Frame zum level.sight_client. Sollten alle Clients tot sein oder sich im „Notarget“-Modus befinden, wird level.sight_client zu NULL (In einem Frame ist aus Perfomanz-Gründen wirklich immer nur *ein* Client für die Monster sichtbar).
- Wurde in einem Client-Frame level.exitintermission auf true gesetzt, so wird die Intermission mit Hilfe von `void ExitLevel(void)` verlassen:
  - gi.AddCommandString("gamemap level.changemap") (S. 122) wird ausgeführt.
  - level.changemap auf NULL
  - level.exitintermission und level.intermissiontime auf 0
  - ClientEndServerFrames() wird aufgerufen, was für jeden Client die Funktion ClientEndServerFrame() ausführt (s.u.).
  - Die Gesundheit der Clients wird auf pers.max_health reduziert, falls sie diesen Wert überschreiten sollte (z.B. durch item_health_mega).

  ... und dann dieser Frame beendet.
- Nacheinander wird für jedes benutzte Entity (ent→inuse ist true) folgendes ausgeführt (und zwar entsprechend dem Index des Entities: also zuerst die Welt, dann die Clients und anschließend die Restlichen):
  - level.current_entity auf ent
  - ent→s.old_origin auf ent→s.origin
  - Wenn das Entity auf einem anderen steht (ent→groundentity ≠ NULL) und sich dieses seit dem letzten Frame bewegt hat (ent→groundentity→linkcount ≠ ent→groundentity_linkcount), wird ent→groundentity auf NULL gesetzt. Handelt es sich hier um ein Monster (bei ent→svflags ist SVF_MONSTER gesetzt), welches weder schwimmt noch fliegt, wird zusätzlich M_CheckGround() aufgerufen, um zu prüfen, ob das Monster immer noch auf einem Entity steht (dort wird dann auch das neue ent→groundentity sowie dessen ent→groundentity_linkcount eingetragen).
  - Für jeden Client wird ClientBeginServerFrame() aufgerufen, für alle anderen Entities G_RunEntity() (s.u.).
  - Mit CheckDMRules() wird geprüft, ob im DM der Level aufgrund des timelimit oder des fraglimit beendet werden muß; ist dies der Fall, wird die Funktion EndDMLevel() ausgeführt, welche dann den Folgelevel aussucht und die Spieler mit Hilfe von BeginIntermission() in die Intermission schickt.
  - Mit CheckNeedPass() wird geprüft, ob sich das password oder das spectator_password geändert hat; wenn ja wird needpass entsprechend gesetzt.

    - Zum Schluß wird noch ClientEndServerFrames() aufgerufen, was für jeden Client die Funktion ClientEndServerFrame() ausführt (s.u.).

Und jetzt die oben ausgelassenen Funktionen.
Die Funktion `void ClientBeginServerFrame(edict_t *ent)`:

- Sind wir in Intermission: fertig.
- Sind wir im DM und der Spieler hat seinen Modus (von Zuschauer zu Spieler oder umgekehrt) geändert, wird spectator_respawn() aufgerufen.
- Wurde seit dem letzten Server-Frame in keinem Client-Frame geschossen (client→weapon_thunk ist false), wird jetzt mit Hilfe von Think_Weapon() (siehe bei Client-Frame S. 147) die Waffenanimation durchgeführt.

  Wurde geschossen (client→weapon_thunk ist true), so wird client→weapon_thunk wieder auf false zurückgesetzt.
- Ist der Spieler tot: Ist client→respawn_time (Zeitpunkt an dem der Spieler frühestens wieder eigesetzt werden kann) schon überschritten und wurde ein Knopf gedrückt (im DM nur beim Angriffsknopf) oder ist das dmflags-Flag DF_FORCE_RESPAWN gesetzt, wird der Spieler durch den Aufruf von respawn() wieder eingesetzt. Dann sind wir aber auf jeden Fall hier fertig.
- Sind wir nicht im DM und ist der letzte Punkt des Spieler-Trails nicht mehr im sichtbaren Bereich, wird der jetzige Standpunkt dem Trail übergeben und dabei der älteste Punkt gelöscht.

Die Funktion `void G_RunEdict(edict_t *ent)`:

- Falls vorhanden, wird die ent→prethink()-Funktion des Entities ausgeführt.
- Nun wird, je nach Bewegungsart ent→movetype, die Bewegungsfunktion aufgerufen.
- Bei MOVETYPE_PUSH und MOVETYPE_STOP (i.d.R. Türen, Plattformen, etc.) ist das die Bewegungsfunktion `void SV_Physics_Pusher(edict_t *ent)`:
    - Ist dies ein untergeordetes Teammitglied (Flag FL_TEAMSLAVE in ent→flags gesetzt), dann sind wir auch schon fertig; deren Bewegung wird nämlich vom Teammaster veranlaßt und somit das Team als eine Einheit behandelt (geht natürlich nur gut, wenn alle Entities im Team MOVETYPE_PUSH oder MOVETYPE_STOP haben).
    - Der Zeiger pushed_p wird auf Liste pushed[MAX_EDICTS] gesetzt; beide sind in der Datei `g_phys.c` lokal definiert und vom Typ pushed_t:

edict_t	*ent	Zeiger auf das Entity
vec3_t	origin	Position und Orientierung des Entities vor der Bewegung
vec3_t	angles	
float	deltayaw	

↪ Für jedes Teammitglied wird nun, falls ent→velocity oder ent→avelocity $\neq 0$ ist, `qboolean SV_Push(edict_t *pusher, vec3_t move, vec3_t amove)` aufgerufen; hierbei sind move und amove jeweils $\frac{1}{10}$ von ent→velocity bzw. ent→avelocity:

  ↪ Die Bewegung wird auf achtel-Einheiten gerundet, damit die Position des pusher-Entities (im folgenden nur noch der „Pusher" genannt) für die Client-Vorhersage besser zu verarbeiten ist (Bemerkung: Hier wird also ein Fehler eingebracht, der sich im Spiel sichtbar auswirken kann, z.B. wenn eine Tür ständig auf- und zugeht. Um dies zu verhindern, sollte man dieses Verhalten beim Festlegen von ent→(a)velocity berücksichtigen (wohlgemerkt nur für Entities mit MOVETYPE_STOP oder MOVETYPE_PUSH); dazu muß $\frac{8*\text{ent}\rightarrow\text{(a)velocity}}{10}$ eine ganze Zahl sein!)).

  ↪ Die Ziel-Bounding-Box des Pushers wird errechnet.

  ↪ Die Ausgangsposition wird über pushed_p in der Liste pushed gespeichert und pushed_p auf das nächste Listenelement gerichtet.

  ↪ Der Pusher wird an seine Zielposition gesetzt und mit der Welt verbunden (gi.linkentity()).

  ↪ Nun werden alle Entities (other) der Reihe nach geprüft: Kann es vom Pusher überhaupt beeinflußt werden (*nicht* beeinflußt werden die Entities mit MOVETYPE_PUSH, MOVETYPE_STOP, MOVETYPE_NONE als auch MOVETYPE_NOCLIP, d.h. durch diese Art Entities wird einfach „durchgefahren")? Wenn ja, befindet es sich überhaupt innerhalb der Ziel-Bounding-Box oder besitzt es den Pusher als other→groundentity (im folgenden nur noch „Groundentity" genannt)? Ist dies auch mit ja zu beantworten, so folgen die weiteren Schritte (das betreffende Entity other wird im folgenden auch als der „Gepushte" bezeichnet):

  Hat der Pusher den MOVETYPE_PUSH oder ist der Pusher das Groundentity des Gepushten, so wird es ebenfalls in der Liste pushed gespeichert und bewegt (die Bewegung des Gepushten besteht aus zwei Teilen: erstens die Vowärtsbewegung move des Pushers und zweitens eine Vorwärtsbewegung aufgrund der Drehbewegung des Pushers amove). Für den Fall, daß das Groundentity des Gepushten nicht der Pusher ist, wird noch other→groundentity auf NULL gesetzt, da der Gepushte eventuell von seinem Groundentity heruntergeschoben wurde (somit prüft dieser im nächsten Frame wieder, ob er auf etwas draufsteht). Nun wird ermittelt, ob der Gepushte seine Bewegung ungehindert durchführen kann, oder ob er durch ein festes Objekt aufgehalten wird. Wird er nicht behindert, so wird er mit der Welt verbunden (gi.linkentity()) und das nächste Entity ist an der Reihe, geprüft zu werden. Wird er jedoch behindert, so wird seine Bewegung zurückgenommen und dann ermittelt, ob er nicht dort stehenbleiben kann (funktioniert natürlich nur bei Gepushten, die auf dem Pusher mitfahren); kann er stehenbleiben, so wird er aus der pushed-Liste gestrichen und das nächste Entity ist an der Reihe, geprüft zu werden (Pusher fährt dann also unter ihm weg; Bemerkung: Hier muß das Groundentity nicht auf NULL gesetzt werden, da der Gepushte im nächsten Frame ja überprüft, ob sich sein Groundentity bewegt hat). Funktioniert dies auch nicht, so ist die Bewegung endgültig blockiert.

Ist die Bewegung wie im letzten Abschnitt beschrieben blockiert oder hat der Pusher den MOVETYPE_STOP, so ist die Bewegung des Pushers fehlgeschlagen. Der Gepushte wird über den in dieser Datei lokal definierten Zeiger obstacle (Typ edict_t) gespeichert, damit später (außerhalb dieser Funktion) die Block-Funktion mit den Beteiligten der Blockierung als Parameter aufgerufen werden kann. Alle bisher bewegten Entities (auch die schon durch einen früheren Aufruf dieser Funktion bewegten Entities durch andere Teammitglieder bzw. die Teammitglieder selbst), deren frühere Positionen alle in der Liste pushed gespeichert wurden, werden nun auf ihre originalen Positionen zurückgesetzt und mit der Welt verbunden (gi.linkentity()) (dabei wird die pushed-Liste von hinten her abgearbeitet, damit alles stimmt, falls einige Entities mehrfach bewegt wurden, was ja im Bereich des Möglichen liegt). Wenn dies erledigt ist, wird false zurückgegeben.

↪ Wurde die Bewegung von keinem Entity blockiert, so war diese erfolgreich. Für alle bewegten Entities aus der Liste pushed wird nun die Funktion G_TouchTriggers() aufgerufen, um, wenn nötig, Trigger zu berühren (Bemerkung: meiner Meinung nach wäre dies in der aufrufenden Funktion SV_Physics_Pusher() besser aufgehoben, und zwar erst nachdem *alle* Teammitglieder sich bewegt haben; hier werden nämlich möglicherweise einige Trigger mehrfach berührt (diese Funktion wird ja von allen Teammitgliedern nacheinander aufgerufen) oder – was eigentlich noch gravierender sein kann – die Trigger werden von den ersten Teammitgliedern berührt, aber ein folgendes Teammitglied wird blockiert und damit alle Bewegungen annulliert, also wurde der Trigger fälschlicherweise berührt). Anschließend wird true zurückgegeben.

↪ Sollte SV_Push() nun für ein Teammitglied negativ beendet werden, so wird die Funktion für die folgenden Teammitglieder erst gar nicht mehr aufgerufen, da die Bewegung des Teams blockiert wurde. In jedem Fall, bei einem Blockieren oder auch bei geglückter Bewegung, wird nun überprüft, ob eventuell die Liste pushed bei dieser Aktion „übergelaufen“ ist. Dies ist – rein theoretisch – möglich, da zum einen (wahrscheinlich aus Performanz-Gründen) keine Listenlängekontrolle durchgeführt wurde, und zum anderen jedes Entity auch mehrfach bewegt werden kann. Ist dies also wider aller Erwartung aufgetreten, so verabschiedet sich das Spiel mit der Fehlermeldung „pushed_p > &pushed[MAX_EDICTS], memory corrupted“.

- ↪ Wurde die Bewegung blockiert, werden, falls überhaupt vorhanden, die ent→nextthink-Zeitpunkte aller Teammitglieder um einen Frame nach hinten verschoben. Anschließend wird die ent→blocked()-Funktion des Teammitglieds aufgerufen, welches die Blockierung ausgelöst hat (und zwar mit obstacle (s.o.) als zweiten Parameter, die Parameter drei und vier sind NULL).

  Hat die Bewegung aller Teammitglieder jedoch geklappt, also SV_Push() immer true zurückgeliefert, werden die ent→think()-Funktionen aller Teammitglieder bei Bedarf mit Hilfe von `qboolean SV_RunThink(edict_t *ent)` aufgerufen:

  - ↪ Sollte ent→nextthink ≤ 0 oder $>$ (level.time + 0.001) sein, wird true zurückgegeben und nicht „gedacht“.
  - ↪ ent→nextthink wird auf 0 gesetzt und die ent→think()-Funktion aufgerufen (*Achtung:* sollte hier keine Think-Funktion vorhanden sein, wird das Spiel mit der Fehlermeldung „NULL ent→think“ abgebrochen); dann wird die Funktion mit der Rückgabe von false beendet.

  Damit ist dann auch SV_Physics_Pusher() am Ende angelangt.

- ↪ Bei MOVETYPE_NONE ist das die Bewegungsfunktion `void SV_Physics_None (edict_t *ent)`:

  - ↪ Es wird nur SV_RunThink() aufgerufen; fertig.

- ↪ Bei MOVETYPE_NOCLIP ist das die Bewegungsfunktion `void SV_Physics_Noclip(edict_t *ent)`:

  - ↪ Es wird SV_RunThink() aufgerufen. Wurde „gedacht“, dann sind wir fertig.

    Ansonsten wird $\frac{1}{10}$ der Bewegung aus ent→(a)velocity durchgeführt und das Entity wieder mit der Welt verbunden (gi.linkentity()) (d.h. dieses Entity kann entweder nur „denken“ oder sich nur bewegen!).

- ↪ Bei MOVETYPE_STEP (i.d.R. Monster) ist das die Bewegungsfunktion `void SV_Physics_Step(edict_t *ent)` (ent→velocity wird auch als „die Velocity“, ent→avelocity auch als „die Avelocity“ bezeichnet). Hierbei wird die Bewegung durch etliche Maßnahmen „gebremst“, damit die Bewegungen nicht gleitend aussehen, sondern wie echte Schritte:

  - ↪ Ist kein Groundentity vorhanden, wird mit M_CheckGround() nach einem gesucht.
  - ↪ Mit SV_CheckVelocity() werden die einzelnen Richtungen der Velocity im Betrag auf sv_maxvelocity→value begrenzt, d.h. die Schnelligkeit der Monster wird

durch die Cvar sv_maxvelocity eingeschränkt.

↪ Mit SV_AddRotationalFriction() wird die Avelocity etwas verringert, falls sie $\neq$ vec3_origin ist (Reibung)

↪ Ist immer noch kein Groundentity vorhanden und ist das Monster weder fliegend noch ganz im Wasser schwimmend (es fällt bzw. springt oder sinkt gerade ins Wasser ein):

Ist die vertikale Velocity nach unten größer als die Gravitationsbeschleunigung eines Frames, so wird später ein Landegeräusch ausgegeben, wenn das Monster dann auf dem Boden angekommen ist. Ist das Monster ganz außerhalb des Wassers (ent→waterlevel ist 0), wird mit `void SV_AddGravity(edict_t *ent)` die Gravitationsbeschleunigung hinzugefügt:

  ↪ Von der vertikalen Velocity wird ent→gravity∗sv_gravity→value∗FRAMETIME abgezogen (also jeden Frame fliegt das Monster etwas schneller nach unten bzw. etwas langsamer nach oben).

↪ Bei fliegenden Monstern mit vertikaler Velocity wird diese ein wenig verringert (Reibung der Luft).

↪ Bei schwimmenden Monstern mit vertikaler Velocity wird dasselbe mit höherem Wert durchgeführt (Reibung des Wassers).

↪ Hat das Monster nun eine Velocity:

  ↪ Hat es ein Groundentity oder schwimmt bzw. fliegt es und ist es darüberhinaus noch am Leben oder tot, aber noch nicht ganz am Boden, wird die horizontale Velocity reduziert (tote Monster, die in der Luft oder im Wasser nach unten „driften“, machen dies gleitend, also mit konstanter Geschwindigkeit)

  ↪ Dann wird `int SV_FlyMove(edict_t *ent, float time, int mask)` aufgerufen; dabei ist time die FRAMETIME und mask MASK_MONSTERSOLID falls das Entity wirklich ein Monster ist (Flag SVF_MONSTER ist bei ent→svflags gesetzt) und MASK_SOLID sonst; der Rückgabewert wird nicht berücksichtigt:

  Hier wird eine Schleife gestartet, die bis zu viermal durchlaufen werden kann; dabei wird (mit Hilfe von gi.trace()) die Komplettbewegung eventuell in Teilbewegungen aufgespalten, wobei sich zwischen den Teilbewegungen auch die Velocity ändern kann (insbesondere kann die Velocity auch nach Verlassen dieser Funktion andere Werte aufweisen als zu Beginn): Kann das Monster seine restliche Bewegung ohne Kollision ausführen, wird diese ausgeführt; würde die ganze restliche Bewegung in festen Objekten stattfinden, wird sie nicht ausgefüht und wir sind fertig; bei einer Kollision wird die Teilstrecke bis zu deren Auftreten zurückgelegt und dann mit Hilfe der Funktion `void SV_Impact(edict_t *ent, trace_t *trace)` die Touch-Funktionen beider kollidierter Entities aufgerufen:

SV_Impact() ruft die Touch-Funktion des Entities ent, das sich i.d.R. auf das andere zubewegt hat, mit den Parametern (ent, trace→ent, &trace→plane, trace→surface) auf; jedoch die Touch-Funktion des berührten Entities trace→ent mit (trace→ent, ent, NULL, NULL).

Sollte das Monster jetzt nicht mehr in der Welt sein (von einer Touch-Funktion entfernt), sind wir fertig. Anderenfalls wird versucht, die Bewegung fortzusetzen. Ist nun die vertikale Normalenkoordinate der getroffenen Oberfläche > 0.7, so stehen wir auf dem Boden (wird auch als Groundentity eingetragen, falls SOLID_BSP), ist diese Koordinate = 0, sind wir gegen eine senkrechte Wand geprallt. Die restliche Bewegungszeit wird berechnet und die Velocity dahingehend verändert, daß das Monster auf der Oberfläche entlanggleitet, sich also parallel zur Oberfläche bewegt (in späteren Schleifendurchläufen, falls das Monster noch einmal kollidiert, sind es mehrere Oberflächen; dann wird diejenige Velocity genommen, die parallel zu einer der Oberflächen verläuft, aber in einem spitzen Winkel zu allen anderen Oberflächennormalen steht; wird keine solche gefunden, so wird bei genau zwei Oberflächen noch ein Mittelweg genommen, bei mehreren dann aber die Velocity auf vec3_origin gesetzt und die Bewegung beendet). Jetzt wird eine neue Schleife ausgeführt, aber wiederum nur, falls die neue Velocity mit der ursprünglichen einen spitzen Winkel aufspannt; anderenfalls wird die Velocity auf vec3_origin gesetzt und die Bewegung beendet.

- ↪ Anschließend wird das Monster mit der Welt verbunden (gi.linkentity()) und mit G_TouchTriggers(), wenn nötig, Trigger berührt.
- ↪ Sollte das Monster nun nicht mehr in der Welt sein (z.B. durch Killtarget oder durch eine tödliche Kollision bei der Bewegung): fertig.
- ↪ Sollte das Monster auf dem Boden gelandet sein, und ist oben festgestellt worden, daß die Landung „hart“ genug war, wird noch ein Landegeräusch ausgegeben.

↪ Abschließend wird noch mit SV_RunThink() gedacht.

↪ Bei MOVETYPE_TOSS (der Gravitation unterworfen), MOVETYPE_BOUNCE (abprallend, i.d.R. Granaten etc.), MOVETYPE_FLY (fliegend, nicht benutzt) als auch MOVETYPE_FLYMISSILE (i.d.R. Geschosse wie Rakete, BFG- oder Blaster-Schuß) ist das die Bewegungsfunktion `void SV_Physics_Toss(edict_t *ent)`:

- ↪ Hier wird zuerst gedacht, indem SV_RunThink() aufgerufen wird.
- ↪ Ist das Entity nicht der Teammaster: fertig.
- ↪ Ist die vertikale Velocity > 0 oder das bisherige Groundentity nicht mehr in der

Welt (other→inuse ist false), wird Groundentity zu NULL.

- ↪ Stehen wir jetzt noch auf einem Groundentity, bewegen wir uns nicht und sind fertig.
- ↪ Die Velocity wird mit SV_CheckVelocity() (s.o.) begrenzt.
- ↪ Falls ent→movetype≠(MOVETYPE_FLY oder MOVETYPE_FLYMISSILE), wird mit SV_AddGravity() (s.o.) die Gravitationsbeschleunigung hinzuaddiert.
- ↪ Die Drehbewegung wird ausgeführt.
- ↪ Die mit der FRAMETIME skalierte Velocity (das entspricht der Bewegung in diesem Frame) wird der Funktion `trace_t SV_PushEntity(edict_t *ent, vec3_t push)` als Parameter push übergeben:
  - ↪ Die Bewegung push wird soweit als möglich ausgeführt, bis das Entity entweder auf ein Objekt der Art ent→clipmask (falls nicht vorhanden MASK_SOLID) trifft, oder die ganze Bewegung ausgeführt hat (hier wird gi.trace() verwendet).
  - ↪ Dort wird es in jedem Fall mit der Welt verbunden (gi.linkentity()) und im Fall einer Kollision SV_Impact() aufgerufen (s.o.).
  - ↪ Sollte anschließend das getroffene Objekt nicht mehr in der Welt sein und das Entity selbst hingegen immer noch da sein, wird es wieder in den Startzustand gesetzt, mit der Welt verbunden und das Ganze noch einmal versucht.
  - ↪ Zuletzt wird, wenn das Entity selbst noch in der Welt ist, G_TouchTriggers() ausgeführt, um, wenn nötig, Trigger zu berühren.
  - ↪ Danach wird die trace_t-Struktur zurückgegeben.
- ↪ Sollte das Entity nun nicht mehr in der Welt sein (z.B. wenn von SV_Impact() zerstört), sind wir fertig.
- ↪ Sind wir bei der Bewegung kollidiert, steht das Entity jetzt am Punkt der Kollision:
  - ↪ Jetzt wird eine neue Velocity berechnet: gilt MOVETYPE_BOUNCE, so zeigt sie von der getroffenen Oberfläche weg, sonst zeigt sie entlang der Oberfläche.
  - ↪ Ist die getroffene Oberfläche annähernd waagerecht (vertikale Normalenkoordinate > 0.7), sind wir auf dem Boden und das Entity bleibt liegen (trace.ent wird Groundentity und (A)Velocity werden zu vec3_origin), es sei denn es gilt MOVETYPE_BOUNCE und die neue vertikale Velocity ist ≥ 60. In jedem Fall findet eine weitere Bewegung erst im nächsten Frame statt.
- ↪ Nun wird noch getestet, ob das Entity im Wasser war und ob es jetzt im Wasser

ist (ent→waterlevel und ent→watertype werden angepaßt). Ist es ins Wasser gefallen oder aus dem Wasser herausgekommen, wird noch ein Geräusch erzeugt.

↪ Zuletzt werden noch die anderen Teammitglieder bewegt; jedoch nur in einfacher Weise, indem der Ort dieser Entities (ent→s.origin) bei den anderen auch eingetragen wird (other→s.origin) und diese mit der Welt verbunden werden.

Die Funktion `void ClientEndServerFrame(edict_t *ent)`:

↪ Die Werte client→ps.pmove.origin und client→ps.pmove.velocity werden erneuert, damit die Client-Sicht nicht der Position des Körpers hinterherhinkt, falls der Spieler im Server-Frame durch andere Entities geschoben oder gestoßen wurde.

↪ Befinden wir uns in der Intermission:

  ↪ client→ps.blend[3] auf 0

  ↪ client→ps.fov auf 90

  ↪ Aufruf von G_SetStats() (s.u.)

  . . . und schon fertig!

↪ Aufruf von `void P_WorldEffects(void)` für die Effekte in der Welt, z.B. Ertrinken in Wasser, Schaden in Lava, . . . :

  ↪ Sind wir MOVETYPE_NOCLIP, haben Flüssigkeiten keine Wirkung auf uns (insbesondere brauchen wir nicht zu atmen): ent→air_finished auf level.time + 12, fertig.

  ↪ Sind wir gerade in Flüssigkeit (Wasser, Lava oder Säure) eingetaucht oder aus einer solchen herausgestiegen, wird ein entsprechendes Geräusch ausgegeben und ein Sound-Entity mit Hilfe PlayerNoise() erzeugt (damit Monster, die keinen Client sehen, wenigstens in Richtung eines Geräusches marschieren können, um dort möglicherweise fündig zu werden).

  ↪ Geht gerade der Kopf unter Wasser oder kommt er gerade raus, wird ebenfalls ein entsprechendes Geräusch abgegeben, schnappt der Spieler dabei nach Luft, weil ihm diese unter Wasser ausgegangen ist, wird ebenfalls ein Sound-Entity erzeugt.

  ↪ Ist der Spieler ganz unter Wasser, wird das „Ersticken“ untersucht: Wird der Breather oder der Environmentsuit benutzt, wird ent→air_finished erhöht und ein Geräusch incl. Sound-Entity erzeugt. Hat der Spieler hingegen keine Luft mehr (ent→air_finished-Zeitpunkt abgelaufen) und ist die nächste „Schadenszeit“ erreicht (client→next_drown_time vorbei) und lebt der Spieler noch, wird ein Gurgelgeräusch ausgegeben und dem Spieler Schaden zugefügt (je länger man am Ertrinken ist, desto mehr Schaden bekommt man). Ansonsten, wenn der Spieler nicht ganz unter Wasser ist, wird einfach der Luftvorrat erhöht.

  - ↪ Ist der Spieler dazu noch in Lava, wird ein Schreigeräusch ausgegeben und ihm Schaden zugefügt (mit „Environmentsuit“ nur $\frac{1}{3}$); in Säure gibt es nur Schaden, wenn der „Environmentsuit“ nicht aktiv ist.

- ↪ Die Orientierung des Models ent→s.angles wird mit Hilfe von client→v_angle und SV_CalcRoll() gesetzt, damit andere Entities sehen können, in welche Richtung der Spieler schaut.
- ↪ Die Geschwindigkeit und der Zyklus für alle zyklischen Geheffekte werden berechnet.
- ↪ Mit `void P_FallingDamage(edict_t *ent)` wird untersucht, ob der Spieler auf den Boden gefallen ist und bei Bedarf dem Spieler entsprechender Schaden zugefügt:
  - ↪ Wasser dämpft dabei den Aufprall (ganz im Wasser ⇒ kein Aufprall, knapp im Wasser ⇒ halber Aufprall).
  - ↪ mini-Aufprall ⇒ war nur ein Schritt (EV_FOOTSTEP)
  - ↪ mittlerer Aufprall ⇒ Fall von niedriger Höhe (EV_FALLSHORT)
  - ↪ starker Aufprall ⇒ Fall (EV_FALL)
  - ↪ enormer Aufprall ⇒ Fall von hoch oben (EV_FALLFAR)
- ↪ Mit `void P_DamageFeedback(edict_t *player)` wird die Reaktion des Spielers auf den Schaden dieses Frames (mit Hilfe von T_Damage() zugefügt) durchgeführt:
  - ↪ Farbschleier über das Bild legen (Statusbar oder Hilfeanzeige sind davon nicht betroffen, nur die 3D-Welt) – client→damage_alpha: Durchsichtigkeit, client→damage_blend: Farbe.
  - ↪ Farbschleier unter Rüstungs- bzw. Erste-Hilfe-Anzeige legen (client→ps.stats[STAT_FLASHES])
  - ↪ Spieler Schmerz-Animation
  - ↪ Spieler Schmerz-Geräusch (falls nicht schon geschehen)
  - ↪ Berechnung der Sichtwinkel-Verschiebung (v_dmg_∗)
- ↪ Mit `void SV_CalcViewOffset(edict_t *ent)` wird die kumulierte Verschiebung (client→ps.kick_angles) und der kumulierte Offset (client→ps.viewoffset) der Spieler-Sicht errechnet:
  - ↪ Bei Toten wird der Sichtwinkel (client→ps.viewangles) festgelegt (es findet keine Verschiebung statt).
  - ↪ Die Verschiebung setzt sich zusammen aus dem Abfeuern einer Waffe, dem erlittenen Schaden, dem freien Fall, der allgemeinen Bewegung sowie der oben berechneten Gehbewegung.
  - ↪ Der Offset setzt sich hingegen aus der Sichthöhe des Spielers, der Fallhöhe, der

Gehbewegung und dem Rückstoß zusammen (Viewoffset kann nie außerhalb der Spieler Bounding Box liegen).

↪ Mit SV_CalcGunOffset() wird dementsprechend der Winkel der Waffe client→ps.gunangles gesetzt und (eigentlich nur für die Waffenentwicklung gedacht) deren Offset client→ps.gunoffset berechnet (abhängig von gun_x, gun_y, gun_z).

↪ Mit SV_CalcBlend() werden alle Farbschleiereffekte kumuliert (client→ps.blend); diese sind abhängig von der Zusammensetzung der Materie vor den Augen (Wasser, Lava, Inneres einer Wand) und treten zeitweise auf bei der Benutzung von Powerups, bei erlittenem Schaden und beim Aufheben von Items.

↪ Nun werden die STAT_∗-Variablen gesetzt. Im Zuschauermodus wird dazu `void G_SetSpectatorStats(edict_t *ent)` benutzt:

  ↪ Wird keinem Spieler gefolgt, so wird zuerst G_SetStats ausgeführt.

  ↪ client→ps.stats[STAT_LAYOUT] wird neu berechnet.

  ↪ Wird nun ein anderer verfolgt, so wird STAT_CHASE gesetzt.

  Im Spielermodus werden mit `void G_SetStats(edict_t *ent)` die Zustände aktualisiert bezüglich der Gesundheit, der Munition, der Rüstung, einer eventuellen Aufnehme-Nachricht, diverser Timer, dem ausgewählten Item, dem Layout, der Frags (Zähler im DM) sowie dem Hilfe-Symbol bzw. der aktivierten Waffe.

↪ Mit G_CheckChaseStats() wird allen, diesem Spieler als Zuschauer Folgenden dessen Status übermittelt und „deren“ G_SetSpectatorStats() ausgeführt.

↪ Mit G_SetClientEvent() wird, wenn sonst noch kein Event gesetzt wurde und es von der Bewegung bzw. Zeit her möglich ist, ein EV_FOOTSTEP-Event ausgelöst (ent→s.event).

↪ Mit `void G_SetClientEffects(edict_t *ent)` werden die Client-Effekte gesetzt (ent→s.effects bzw. ent→s.renderfx), wenn der Spieler Powerup-Items benutzt (oder im Godmode spielt).

↪ Mit G_SetClientSound() wird das Hilfe-Piep-Geräusch wenn nötig ausgegeben und das Geräusch festgelegt, das der Spieler sonst von sich gibt (ent→s.sound) (z.B. Waffengeräusche, Geräusch beim Stehen in schädlichen Flüssigkeiten – nicht das Schreien)

↪ Mit G_SetClientFrame() wird die laufende Spielfiguranimation ausgeführt, also der Spieler-Frame ent→s.frame gesetzt.

↪ ent→velocity und client→ps.viewangles werden in der Client-Struktur (old_∗) gespeichert.

↪ Der Waffenrückstoß wird gelöscht.

↪ Ist das „Scoreboard“ (die DM-Anzeigetafel) auf dem Schirm, wird es mit Hilfe von

DeathmatchScoreboardMessage() aktualisiert.

Damit ist der Server-Frame abgeschlossen … für die nächsten 0.1 Sekunden …

Speicherung

In Quake 2 sind zwei Arten von Speicherungen vorgesehen. Zum einen die explizite Speicherung des Spielstandes durch den Spieler, zum anderen die automatische Speicherung zu Beginn eines jeden Levels, um nach dem Tod des Spielers wieder beim letzten Leveleinstieg beginnen zu können. Da zu Beginn des Levels jedoch alle Entities noch in ihrem Ausgangszustand aus der Leveldefinitionsdatei sind, kann bei der automatischen Speicherung, die ja schnell gehen soll, der umfangreiche Entity-Zustand weggelassen werden. Aus diesem Grund wurde die Speicherung wohl auf zwei Dateien aufgeteilt (game.ssv und server.ssv). Bei der Speicherung tritt auch noch ein Problem mit den Zeigervariablen auf: da sie ja Speicheradressen im Arbeitsspeicher darstellen, werden diese Adressen beim Laden eines Spielstandes sicherlich nicht mehr mit den ursprünglichen Speicheradressen übereinstimmen. Also erfahren die Zeigervariablen bei der Speicherung eine Sonderbehandlung. Da die Variablendefinitionen der game dll von Version zu Version unterschiedlich sein können, ist es nur erlaubt, Spiele zu laden, die mit der gleichen game dll-Version abgespeichert wurden. Desweiteren müssen auch die Variablen der Entities aus der Leveldefinitionsdatei richtig zugeordnet werden; die Leveldefinitionen bezüglich der Entities entsprechen ja im gewissen Sinn auch einer Speicherung, nur daß sich diese Leveldefinitionen nicht nur auf eine game dll Version beschränken darf, sondern eigentlich mit allen game dll-Versionen zusammenarbeiten sollte. Um dieses Problem und das der Zeigervariablen zu lösen, existiert der Strukturtyp field_t:

char	*name	Feldname bei Speicherung, i.d.R. der Variablenname
int	ofs	Offsetadresse in der Struktur
fieldtype_t	type	Feldtyp
int	flags	zusätzliche Flags

Die Strukturoffsetadresse wird mit Funktionsmakros bei der Kompilierung der game dll berechnet, Argument x entspricht dann dem jeweiligen Variablennamen in der Struktur:

FOFS(x)	Offset in edict_t
STOFS(x)	Offset in spawn_temp_t
LLOFS(x)	Offset in level_locals_t
CLOFS(x)	Offset in gclient_t

Die Feldtypen sind in der Enumeration fieldtype_t definiert (Typen ohne Spezialbehandlung, die also so gespeichert werden wie sie im Speicher stehen, sind mit (o) markiert; Zeigertypen, die als Index oder String gespeichert werden, mit (m)):

F_INT	(0) Integertyp int (o)
F_FLOAT	(1) Floattyp float (o)
F_LSTRING	(2) Zeiger, TAG_LEVEL (m)
F_GSTRING	(3) Zeiger, TAG_GAME (m)
F_VECTOR	(4) Vektortyp vec3_t (o)
F_ANGLEHACK	(5) speziell für Variable angle (o)
F_EDICT	(6) Zeiger auf Entity (m)
F_ITEM	(7) Zeiger auf Item (m)
F_CLIENT	(8) Zeiger auf Client (m)
F_FUNKTION	(9) Funktionszeiger (m)
F_MMOVE	(10) Zeiger auf Monsterinfo monsterinfo_t (m)
F_IGNORE	(11) wird ignoriert

als Flags sind noch möglich:

FFL_SPAWNTEMP	(1) Feld ist eine temporäre Variable bei der Entity-Erzeugung (S. 112); wird nicht gespeichert, sondern kann nur beim Lesen eines Levels vorhanden sein
FFL_NOSPAWN	(2) Felder, die nicht durch den Level-Editor setzbar sind

Es gibt nun drei Listen des Typs field_t (in `g_save.c`), die beim Laden/Speichern benutzt werden. fields: zum einen die Felder, die beim Lesen eines Levels berücksichtigt werden, zum anderen alle Zeigerfelder im Speicherbereich einer edict_t-Struktur (also auch in deren „Unterstrukturen“ wie z.B. ent→s, ent→solid und ent→monsterinfo); levelfields: alle Zeigerfelder im Speicherbereich der level-Struktur; clientfields: alle Zeigerfelder im Speicherbereich der Client-Strukturen (Typ: gclient_t). Möchte man nun, daß eigene Zeiger auch richtig gespeichert werden, so sollten diese in die obigen Listen mitaufgenommen werden. Sollen auch eigene Strukturen, die nach dem obigen Muster nicht mit gespeichert werden (z.B. wenn in edict_t nur ein Zeiger auf die eigene Struktur definiert ist), so muß die Speicherungsroutine dementsprechend erweitert werden. Oder will man, daß man eigene Variablen auch vom Level-Editor her übernehmen kann, so muß die Liste fields dahingehend erweitert werden.

Bei der expliziten Speicherung des Spielstandes durch den Spieler werden nacheinander die Funktionen globals.WriteLevel() (schreibt in Datei server.ssv) und globals.WriteGame() (schreibt in Datei game.ssv) aufgerufen. In void WriteLevel(char *filename) wird die level-Struktur sowie alle Entity-Strukturen hintereinander weg gespeichert. In void WriteGame(char *filename, qboolean autosaved) wird das game dll-Erstellungsdatum, die game-Struktur sowie alle Client-Strukturen (gclient_t) gespeichert; hierbei ist autosaved auf false, so daß einige Daten der Client-Entities (z.B. ent→health) noch mit Hilfe der Funktion SaveClient-

Data() (siehe Levelbeginn S. 142) in den Client-Strukturen gespeichert werden. autosaved selbst wird in game.autosaved festgehalten. Bei der automatischen Speicherung ist WriteLevel() überflüssig, so daß hier nur WriteGame() aufgerufen wird, jedoch mit autosaved auf true; somit wird SaveClientData() nicht ausgeführt, so daß der Spieler z.B. immer 100 Gesundheitspunkte besitzt, wenn er nach seinem Tod im Einzelspieler-Modus beim letzen Level-Eintritt weiterspielt.
Die Lesevorgänge verlaufen entsprechend: ReadGame() wird immer direkt vor SpawnEntities() aufgerufen (also nach InitGame()), wenn ein Spielstand geladen wird, ReadLevel() dann nach SpawnEntities(), falls es sich um einen manuell gespeicherten Spielstand handelt.

Client trennt Verbindung

Wird die Verbindung eines Clients zum Server getrennt, sei es auf Wunsch des Clients, oder aufgrund eines Netzwerkfehlers, so ruft der Server die Funktion `void ClientDisconnect (edict_t *ent)` auf. Diese Funktion nimmt den Client aus dem Spiel:

- ↪ Auf der Serverkonsole wird „Clientdisconnect“ ausgegeben.
- ↪ Den Mitspielern wird „pers.netname disconnected“ mitgeteilt.
- ↪ Der Effekt MZ_LOGOUT wird gesendet.
- ↪ Das Client-Entity wird aus der Welt genommen (gi.unlinkentity()).
- ↪ ent→s.modelindex auf 0
- ↪ ent→solid auf NOT_SOLID
- ↪ ent→inuse auf false
- ↪ ent→classname auf disconnected
- ↪ pers.connected auf false
- ↪ Der zum Client gehörige Configstring wird „gelöscht“.

Spielende

Wird ein Spiel und damit auch der Server beendet, ruft der Server die Funktion ShutdownGame() auf, damit die game dll noch Aufräumarbeiten erledigen kann. In ShutdownGame() werden alle in der game dll über gi.TagMalloc() allokierten Speicherbereiche freigegeben. Danch wird die game dll „entladen“ und der Server beendet.

B.3.4 Alphabetisch geordnete Kurzübersicht

Datentypen und Konstanten

Funktionen

B.3.5 Eigene Erweiterungen

In diesem Abschnitt wird nun genauer auf die im Rahmen dieser Arbeit entstandenen Quellcodeerweiterungen der game dll eingegangen. Neben den neuen Datenstrukturen und Entities wird auch der Großteil der Funktionen erläutert.
Folgende Dateien sind entstanden:

b_UniCats.h Alle für die Erweiterungen nötigen Deklarationen; wird zu Beginn von `g_local.h` eingelesen

b_Bewegung.c Teilweise leicht abgeänderte allgemeine Funktionen zur Bewegungsberechnung aus `g_func.c` (u.a. B_Move_Calc(), mit welcher eine Bewegung gestartet wird, an deren Ende eine Funktion ausgeführt werden kann)

b_LaserPointer.c Hier wird die leicht abgeänderte Version der Waffe plasmabeam aus dem Rogue-Mission-Pack implementiert; dieser LaserPointer wird benutzt, um Knöpfe und Bücher zu aktivieren bzw. auszuwählen

b_Funktionen.c Definition der Fahrstuhl- und Rotationsfunktionalität

b_Regal.c Definition der Regalfunktionalität

b_Buch.c Alles rund um Bücher und deren Auswahl

b_UniCats.c Alle sonstigen zum UniCats-System gehörenden Entities

b_EinAusgabe.c Die Funktionen, die die Ein- und Ausgabe der Clients realisieren

b_Kommunikation.c Die zur Kommunikation mit der Benutzeragenten-Schnittstelle nötigen Funktionen

Erweiterungen der Entity-Struktur edict_t

Fahrstuhl_t	*Fahrstuhl	Fahrstuhl-Daten
UniCats_t	*UniCats	UniCats-System-Daten
Buch_t	*Buchdaten	Buch-Daten
Regal_t	*Regal	Regal-Daten

Der Fahrstuhl

Um einen Fahrstuhl zu realisieren, benötigt man eine Gruppe von Entities, die sich wie folgt beschreibt:

B_Fahrstuhl: Dieses Entity stellt das fahrende Fahrstuhl-Gehäuse dar, welches die ganze Fahrstuhllogik beinhaltet. Es handelt sich um ein Solid-Entity inklusive eines origin-Brushes, damit die B_Fahrstuhl_Stock-Entities (s.u.) richtig plaziert werden können (sie liegen in einer Linie mit dem origin-Brush). Alles, was sich mit dem Fahrstuhl mitbewegen soll und sonst kein Entity ist, gehört zu diesem Entity. Zu Beginn steht der Fahrstuhl immer auf Stockwerk 0.

In der Entity-Gruppe, die den Fahrstuhl realisiert, werden folgende Entities erwartet, die alle denselben team-Namen aufweisen (die in Klammer stehende Zahl beschreibt die Anzahl der jeweiligen Entities, wobei a für die Anzahl der Fahrstuhl- Stockwerke steht und in der momentanen Einstellung bis zu STOCKWERKMAXIMUM=16 Stockwerke erlaubt sind):

B_Fahrstuhl (1) dieses Entity

B_Fahrstuhl_Tuer (2) Fahrstuhl-Innentüren, also diejenigen Türen, die mit dem Fahrstuhl-Gehäuse mitfahren

B_Fahrstuhl_Tuer (2a) Fahrstuhl-Außentüren, auf jedem Stockwerk zwei, um dadurch den Fahrstuhlschacht vom Stockwerk abzutrennen

B_Fahrstuhl_Knopf (a) Stockwerksauswahlknöpfe im Inneren des Fahrstuhl-Gehäuses

sonstige Entities (1–10) beliebige, sich normalerweise nicht bewegende Entities, die mit dem Fahrstuhl ständig mitfahren sollen (z.B. Lichter)

B_Fahrstuhl_Knopf (a) ein Fahrstuhlanforderungsknopf auf jedem Stockwerk

B_Fahrtstuhl_Stock (a) Haltepunkt des Fahrstuhls auf jedem Stockwerk

Variablen dieses Entities, die aus dem Level-Editor übernommen werden (in Klammern stehen die Standardwerte, falls nichts übergeben wird):

team () Name des Fahrstuhl-Teams

speed (100) wie üblich (Geschwindigkeit des Fahrstuhls)

accel, decel () wie üblich (Beschleunigung bzw. Verzögerung des Fahrstuhls, falls der Fahrstuhl eine beschleunigte Bewegung ausführen soll; ansonsten gleichmäßige Bewegung)

dmg (2) wie üblich (Schaden, den der Fahrstuhl beim Blockieren dem Blockierenden in jedem Frame zufügt)

sounds (0) wie üblich (bei 0 gibt der Fahrstuhl Geräusche von sich, bei 1 nicht)

B_Fahrstuhl_Tuer: Dieses Entity stellt die zwei Außentüren jedes Stockwerks sowie die zwei Innentüren des Fahrstuhl-Gehäuse dar. Es handelt sich um Solid-Entities. Dabei öffnen und schließen die Innentüren bei Fahrtende am jeweiligen Stockwerk zusammen mit den dortigen Außentüren.

Aus dem Level-Editor werden folgende Variablen übernommen:

team () Name des Fahrstuhl-Teams

height (0) Nummer des Stockwerks, auf dem die Tür plaziert ist; bei 0–$(a-1)$ entspricht dies einer Fahrstuhl-Außentür, bei 16 handelt es sich um eine Fahrstuhl-Innentür

target () wie bei Türen üblich (Areaportalname, falls vorhanden)

speed (40) wie üblich (Geschwindigkeit, mit der die Türen auf- und zugehen)

accel, decel () wie üblich

lip (8) wie bei Türen üblich (die Türen bewegen sich um Breite der eigenen Bounding-Box$-$lip in Richtung angle)

angle (0) wie üblich (Winkel zwischen 0 und 360 Grad für die Richtung; -2 bedeutet nach unten, -1 nach oben)

wait (3) wie bei Türen üblich (die Zeit, die die Türen im offenen Zustand warten, bevor sie sich wieder schließen)

sounds (0) wie üblich

dmg (2) wie üblich

B_Fahrstuhl_Knopf: Diese Entities stellen die Stockwerksauswahlknöpfe im inneren des Fahrstuhl-Gehäuses sowie die Anforderungsknöpfe auf jedem Stockwerk dar. Es handelt sich um Solid-Entities. Durch deren Betätigung nimmt der Fahrstuhl einen Fahrauftrag entgegen, den er zu geeigneter Zeit ausführt (ist er bereits unterwegs und fährt gerade an diesem Stockwerk vorbei, so hält er an; steht er gerade, so beginnt er, sich in Richtung dieses Stockwerks zu bewegen). Beim Betätigen wechseln die Texturframes von 0, 1 auf 2, 3; d.h. es sollte eine Textur mit mindestens vier Folgetexturen gewählt werden.

Aus dem Level-Editor werden folgende Variablen übernommen:

team () Name des Fahrstuhl-Teams

height (0) Nummer des Stockwerks, für welches dieser Knopf eine Anforderung abgibt, also 0–$(a-1)$

distance (0) Ort des Knopfes: bei 0 handelt es sich um einen sich mit dem Fahrstuhl bewegenden Innenknopf, bei 1 um einen auf jedem Stockwerk festen Außenknopf

sounds (0) wie üblich

B_Fahrstuhl_Stock: Diese Entities stellen die Haltepunkte des Fahrstuhls auf jedem Stockwerk dar. Es handelt sich um Point-Entities.

Aus dem Level-Editor werden folgende Variablen übernommen:

team () Name des Fahrstuhl-Teams

height (0) Nummer des Stockwerks, für welches dieses Entity den Haltepunkt darstellt, also 0–$(a-1)$

Datenstrukturtyp Fahrstuhl_t

Bei der Initialisierung eines B_Fahrstuhl-Entities wird Speicher für diese Datenstruktur alloktiert. Der Zeiger ent→Fahrstuhl aller Entities, die zu diesem Fahrstuhl gehören, wird dann auf diese Struktur gerichtet. Somit können alle Entities des Fahrstuhls auf diese gemeinsame Datenstruktur zugreifen.

edict_t	*Fahrstuhl_Entity	das B_Fahrstuhl-Entity
edict_t	*Innentuer[2]	die Fahrstuhl-Innentüren
edict_t	*Aussentuer[16][2]	die Fahrstuhl-Außentüren
edict_t	*Innenknopf[16]	die Fahrstuhl-Innenknöpfe
edict_t	*Aussenknopf[16]	die Fahrstuhl-Außenknöpfe
edict_t	*SonstigeEntities[10]	die sonstigen mitfahrenden Fahrstuhl-Entities
int	sonstigeentities	Anzahl der sonstigen Entities
vec3_t	Stockwerk[16]	Orte der Stockwerke (origin-Werte der B_Fahrstuhl_Stock-Entities)
int	stockwerkanzahl	Anzahl der Stockwerke
qboolean	komplett	true, wenn der Fahrstuhl korrekt initialisiert wurde, alle notwendigen Teile vorhanden sind und der Fahrstuhl einsatzbereit ist
qboolean	aktiv	true, wenn der Fahrstuhl gerade unterwegs ist
edict_t	*Warteschlange[32]	hier wird ein Fahrstuhl-Knopf eingetragen, wenn er betätigt wurde; also der Fahrauftrag gespeichert
int	wartende	Anzahl der Aufträge in der Warteschlange
int	mitfahrende	Anzahl der Entities des Fahrstuhls, die sich gerade zusammen bewegen, damit man erkennen kann, wann alle Entities ihre Bewegung beendet haben
int	stockwerk	Stockwerk, auf dem sich der Fahrstuhl momentan befindet

Folgende Funktionen wurden im Zusammenhang mit dem Fahrstuhl in `b_Funktionen.c` definiert:

Spawn-Funktionen der Entities: B_SP_Fahrstuhl_Stock(), B_SP_Fahrstuhl_Tuer(), B_SP_Fahrstuhl_Knopf(), B_SP_Fahrstuhl()

B_Fahrstuhl_initialisieren() Hier werden die Teile des Fahrstuhls zusammengesucht und die Fahrstuhldaten initialisiert. Diese Funktion wird einen Frame nach dem Aufruf der B_SP_Fahrstuhl()-Spawn-Funktion ausgeführt, damit alle anderen Fahrstuhl-Entities schon erzeugt wurden.

B_Fahrstuhl_Tueren_synchronisieren() Diese Funktion synchronisiert alle Fahrstuhl-Türen (also Innen- wie Außentüren), damit alle gleichzeitig auf- bzw. zugehen.

B_Fahrstuhl_blockiert() Dies ist die ent→blocked-Funktion des Fahrstuhls.

B_Fahrstuhl_Knopf_benutzt() Wird ein Fahrstuhl-Knopf von einem Spieler mit dem Laser-Pointer angeschossen und ist der zugehörige Fahrstuhl einsatzbereit, dann wird die Knopf-Texturanimation umgeschaltet, die weitere Benutzung dieses Knopfes gesperrt, der Knopf in die Auftragswarteschlange eingetragen und, sofern der Fahrstuhl nicht schon unterwegs ist, der Fahrstuhl mit B_Fahrstuhl_weiterfahren() in Bewegung gesetzt.

B_Fahrstuhl_weiterfahren() Ist der Fahrstuhl an einem Stockwerk aus der Auftragswarteschlange angekommen, wird B_Fahrstuhl_angekommen() aufgerufen und fertig. Ansonsten wird die Fahrstuhl-Teamchain aus allen mitfahrenden Entities gebildet (falls sie nicht schon existiert) und zum nächsten Stockwerk in Richtung Auftragsziel geschickt (am Ende der Bewegung ruft jedes Entity B_Fahrstuhl_angekommen() auf).

B_Fahrstuhl_angekommen() Sind alle sich bewegenden Entities an einem Stockwerk angekommen, so wird getestet, ob dadurch ein Auftrag aus der Warteschlange erfüllt wurde. Wenn ja, werden die betreffenden Knöpfe mit Hilfe von B_Fahrstuhl_Knopf_zuruecksetzen() wieder aktiviert, der Auftrag aus der Warteschlange entfernt und die Türen geöffnet. Dazu wird, falls nötig, die bestehende Fahrstuhl-Teamchain aufgelöst und eine Tür-Teamchain gebildet sowie anschließend die Türen in Bewegung gesetzt (am Ende der Bewegung ruft jede Tür wieder B_Fahrstuhl_angekommen() auf). Wurde kein Auftrag aus der Warteschlange erfüllt, so wird einfach B_Fahrstuhl_weiterfahren aufgerufen.

B_Fahrstuhl_Knopf_zuruecksetzen() Hier wird die Knopf-Texturanimation wieder zurückgeschaltet und die Sperrung der Knopfbenutzung aufgehoben.

Erweiterte Rotation

Für die erweiterte Rotation genügt ein einziges vielseitiges Entity, B_Rotation:
Dieses Entity ist ein Solid-Entity inklusive eines origin-Brushes. Variablen dieses Entities, die aus dem Level-Editor übernommen werden:

move_angles (0 1 0) übergeben werden drei Fließkommawerte, die die Drehrichtung des Entities festlegen; dabei entspricht die Reihenfolge der Werte der Angabe von Winkeln: PITCH (y), YAW (z) und ROLL (x) (die Angabe (3 2 1) bedeutet dann also für die Drehrichtung: das Objekt dreht sich 3-mal um die y-Achse während es sich 2-mal um die z-Achse und einmal um die x-Achse dreht)

targetname () Wenn gesetzt, muß das Objekt getriggert werden, um die Drehung bzw. Drehbarkeit im Wechsel ein- und auszuschalten

spawnflags Je nachdem, welche Flags gesetzt sind, hat das folgende Auswirkungen:

- 1 Entity ist drehbar, d.h. das Entity muß angestoßen werden, damit es sich bewegt (wie z.B. die Drehtür; beachte, daß diese Funktionalität nur mit der Standarddrehrichtung (um z-Achse) zusammen korrekt arbeitet); ansonsten dreht sich das Objekt von alleine (selbstdrehend)
- 2 Drehrichtung wird umgekehrt
- 4 Entity dreht sich nur in die Bewegungsrichtung, in die der Spieler drückt (wie bei Drehtür, macht nur Sinn in Verbindung mit Spawnflag 1
- 8 wie func_rotating, die Berührung mit dem Entity fügt dem Spieler Schaden zu (nur bei selbstdrehend)
- 16 dto., beim Blockieren bleibt das Entity stehen, anstatt zu versuchen, den Blockierenden beiseite zu schieben (nur bei selbstdrehend)
- 32 dto., Textur ist animiert (durch alle Frames gehend)
- 64 dto., Textur ist schnell animiert (schnell durch alle Frames gehend)

dmg (0) wie üblich

speed (100) wie üblich

sounds (0) wie üblich

noise () wie üblich (Angabe einer Geräusch-Datei, die das Geräusch enthält, welches während der Bewegung dieses Entities abgegeben werden soll)

Folgende Funktionen wurden im Zusammenhang mit diesem Entity in `b_Funktionen.c` definiert:

B_SP_Rotation() Dies ist die Spawn-Funktion dieses Entities.

B_Rotation_drehbar() Hiermit wird die Drehbarkeit des Entities an- und ausgeschaltet, also die Touch-Funktion dieses Entities auf B_Rotation_drehen() bzw. NULL gestellt (nur bei drehbar).

B_Rotation_drehen() Berührt ein Spieler das Entity, wird (falls drehbar gewählt und Drehbarkeit eingeschaltet) diese Funktion aufgerufen, die das Entity für zwei Sekunden in Bewegung setzt (nach zwei Sekunden wird B_Rotation_drehen_beendet() aufgerufen). Ist die Standarddrehrichtung eingestellt, so wird errechnet, in welche Richtung der Spieler das Entity angestoßen hat und dementsprechend das Entity in Bewegung gesetzt, es sei denn die Drehung in dieser Richtung ist nicht erlaubt (spawnflags: 4 bzw. 2). Ansonsten wird die eingestellte Drehrichtung ohne Richtungsüberprüfung benutzt.

B_Rotation_drehen_beendet() Diese Funktion stoppt die Bewegung der Drehtür.

B_Rotation_selbstdrehend() Hiermit wird die Drehung des Entities ein- und ausgeschaltet (nur bei selbstdrehend). Beim Einschalten wird die Touch-Funktion auf B_Rotation_beruehrt(), beim Ausschalten auf NULL gesetzt, falls das entsprechende spawnflags (16) gesetzt ist.

B_Rotation_beruehrt() Dem berührenden Entity wird Schaden zugefügt.

B_Rotation_blockiert() Beim Blockieren wird dem blockierenden Entity dmg-Schaden zugefügt.

Das UniCats-System

Die restlichen Quelltexterweiterungen dienen einzig und allein dem Recherchesystem – dem sogenannten UniCats-System. Hierfür entworfene Entities beschreiben sich wie folgt:

B_UniCats: Dieses Point-Entity übernimmt die zentrale Rolle des ganzen UniCats-Systems. Es verwaltet alle darin enthaltenen Entities, Daten und Zustände. Für jeden UniCats-Rechercheraum muß ein solches existieren.

In der Entity-Gruppe, die das UniCats-System realisiert, müssen folgende Entities vorhanden sein, die alle den gleichen target-Eintrag aufweisen müssen:

B_UniCats (1) dieses Entity

B_Regal (3) die Regale, in denen dann die Bücher stehen

B_UniCats_Computer (1) der Computerarbeitsplatz, der die Systemeingabemaske aktiviert, wenn er berührt wird

B_UniCats_Auswahlknopf (12) die Auswahlknöpfe, mit denen man die Zuordnung der Darstellungsformen {Buchfarbe, Buchlabel, Regal} zu den Kriterien {Ausleihe, Katalog, Sprache, Jahr} einstellen kann

B_UniCats_Legende (2) die Anzeigen, deren Textur die Zuordnung der Ausprägungen der Darstellungsformen Buchfarbe und Buchlabel zu Ausprägungen der ausgewählten Kriterien beschreibt (z.B. Buchfarbe *rot* – nicht ausleihbar, Buchlabel *englische Flagge* – englisch)

B_UniCats_Schleusentuer (2-4) eine innere und eine äußere Schleusentür (i.d.R. jeweils 2 Türhälften, also 4 Entities)

B_UniCats_Schleusentrigger (4) die Trigger, um die Schleusentüren zu aktivieren, also auf jeder Seite einer Schleusentür einer; zusätzlich wird mit Hilfe der Schleuseninnentürtrigger die Benutzeran- und -abmeldung geregelt

Variablen dieses Entities, die aus dem Level-Editor übernommen werden:

target () Name des UniCats-Systems

sounds (0) wie üblich, Geräusch bei Beginn und Ende der Suchanfrage

B_Regal: Zum einen ist dies das zentrale Entity eines Regals, das z.B. auch die Bücherliste des Regals verwaltet, zum anderen stellt es das Legendenmodul des Regals dar, dessen Textur die Ausprägung des momentan für die Regaldarstellung gewählten Kriteriums für dieses Regal beschreibt (z.B. sofort ausleihbar). Es handelt sich um ein Solid Entity, dessen Textur vier Frames aufweist:

	Regal 1	Regal 2	Regal 3
Frame 0	Sofort ausleihbar	Später ausleihbar	Nicht ausleihbar
Frame 1	Universitätsbibliothek KA	Badische Landesbibliothek	Springer Verlag
Frame 2	Deutsch	Englisch	Sonstige
Frame 3	Ab 1996	1990–1995	Bis 1989

In der Entity-Gruppe, die ein Regal realisiert, sind folgende Entities vorhanden:

B_Regal (1) dieses Entity

B_Regal_Boden (3) die „Kästen“, in denen die Bücher auf den Regalen stehen und transportiert werden

B_Regal_Begrenzung (4) diese definieren den Bereich zur Erstellung der Bücher

B_Regal_Buecherzahl_Anzeige (3) die Ziffern (Hunderter, Zehner und Einer), die die Bücheranzahl auf dem Regal darstellen

B_Regal_Knopf (2) die Schalter, mit denen man die Regalböden nach unten bzw. oben weiterschalten kann

Aus dem Level-Editor werden folgende Variablen übernommen:

target () Name des UniCats-Systems

team () Name des Regalteams, wobei dieser bei Regalen unterschiedlicher UniCats-Systeme durchaus gleich sein kann

height (0) Regalnummer, also 0–2

dmg (2) wie üblich

B_Regal_Boden: Die als „Kasten“ (mit Decke, Boden und Seiten) realisierten Zwischenböden des Regals (also Solid Entities), auf denen ein Teil der Bücher aus der diesem Regal zugeordneten Bücherliste stehen. Dabei bietet das Regal Platz für vier senkrecht übereinander stehende Böden: die zwei mittleren Plätze sind für den Benutzer sichtbar, die zwei äußeren durch das Regalgehäuse (das einfach aus Brushes besteht) verdeckt. Dort werden die Bücher auf den Regalböden erzeugt bzw. entfernt und dann der Regalboden senkrecht nach unten bzw. oben gefahren. Da für den Transport der Bücher nur 3 Böden benötigt werden, sind im Level-Editor nur die untersten drei zu modellieren; die Höhe des Kastenzwischenraums sollte dabei mindestens 36 u betragen (Bücher sind 32 u hoch).

Aus dem Level-Editor werden folgende Variablen übernommen:

target () Name des UniCats-Systems

team () Name des Regalteams

height (0) Regalbodennummer:

0 unterer

1 mittlerer

2 oberer

speed (50) wie üblich

sounds (0) wie üblich

B_Regal_Begrenzung: Diese Point Entities definieren die Grenzen des Bereichs, in dem die Bücher erstellt werden. Sie sollten sich am obersten und untersten Platz für die Regalböden (in dem für den Benutzer nicht sichtbaren Bereich) jeweils am rechten und linken Rand genau auf Höhe der Oberkante des Bodens des Regalboden-Kastens liegen (die vorderen unteren Kanten der Bücher werden dann genau auf der Linie zwischen einer rechten und linken Begrenzung liegen). Tip: im Level-Editor sollte man für die Plazierung der beiden oberen Begrenzungen kurzzeitig einen Regalboden an den oberen Regalbodenplatz verschieben, da dieser Platz normalerweise nicht von einem Regalboden belegt ist.

Aus dem Level-Editor werden folgende Variablen übernommen:

target () Name des UniCats-Systems

team () Name des Regalteams

height (0) Ort der Begrenzung:

0 oben links

1 oben rechts

2 unten links

3 unten rechts

B_Regal_Buecherzahl_Anzeige: Dieses Solid Entity stellt eine Ziffer der Gesamtzahl der auf dem Regal befindlichen Bücher dar. Die Textur umfaßt also zehn Frames mit den Ziffern 0 bis 9.

Aus dem Level-Editor werden folgende Variablen übernommen:

target () Name des UniCats-Systems

team () Name des Regalteams

height (0) Art der Ziffer:

0 Einer

1 Zehner

2 Hunderter

B_Regal_Knopf: Diese Solid Entities stellen Knöpfe dar, die das Regal nach oben bzw. unten weiterschalten. Beim Betätigen des Knopfes nach oben (für den Knopf nach unten gilt entspechendes) wird der Knopf um 4 u nach hinten bewegt und anschließend die drei Zwischenböden nacheinander um eine „Stufe“ nach oben. Ist dabei der obere Zwischenboden im oberen nicht sichtbaren Bereich des Regalgehäuses verschwunden, werden dessen Bücher aus dem Level entfernt und anschließend die in der Bücherliste nächsten Bücher auf dem unteren Zwischenboden erzeugt (dieser befindet sich in diesem Moment noch im unteren nicht sichtbaren Bereich des Regalgehäuses). Mit Hilfe dieses Mechanismus wird es möglich, mehrere hundert Bücher in einem Regal unterzubringen, obwohl nur maximal 26 Stück gleichzeitig auf den zwei sichtbaren Zwischenböden zu sehen sind (hierbei begrenzt nicht der Platz die Anzahl der sichtbaren Bücher, sondern die Tatsache, daß von Quake 2 anscheinend maximal 128 sichtbare Entities gleichzeitig dargestellt werden können). Die Texturen der Knöpfe stellen Pfeile nach oben bzw. unten dar, je nach Richtung der Regalbewegung, die sie auslösen; sind in einer Richtung noch weitere Bücher in der Bücherliste vorhanden, so leuchtet der jeweilige Pfeil. Die Knöpfe haben also entsprechend den Fahrstuhl-Knöpfen vier Frames aufzuweisen: Frame 0, 1 mit nicht leuchtenden Pfeilen; Frame 2, 3 mit leuchtenden Pfeilen.

Aus dem Level-Editor werden folgende Variablen übernommen:

target () Name des UniCats-Systems

team () Name des Regalteams

height (0) Richtung des Knopfes:

0 nach unten

1 nach oben

speed (40) wie üblich

accel, decel () wie üblich

angle (0) wie üblich

lip (4) wie üblich

sounds (0) wie üblich

B_UniCats_Computer: Dieses Point Entity wird mit einem Model eines Computers besetzt (Ausmaße: mins(−44, 0, 0) und maxs(0, 38, 30)). Bei Berührung dieses Computers wird die Eingabemaske aktiviert, so daß der Benutzer eine Suchanfrage formulieren und versenden kann.

Aus dem Level-Editor werden folgende Variablen übernommen:

target () Name des UniCats-Systems, fehlt dieser Eintrag, so hat der Computer keinerlei System-Funktion (kann ohne Probleme auch nur als Computer-Model verwendet werden)

B_UniCats_Auswahlknopf: Mit Hilfe dieser Knöpfe (Solid Entities) auf einer Auswahltafel (Brush mit entsprechender Textur, damit man weiß, welcher Knopf welche Wahl trifft) wird die Zuordnung der Kriterien zu den Darstellungsformen festgelegt (s.o.). Die Textur dieser Knöpfe weist – wie jeder Knopf – vier Frames auf: z.B. Frames 0, 1 rot (nicht ausgewählt) und Frames 2, 3 grün (ausgewählt).

Aus dem Level-Editor werden folgende Variablen übernommen:

target () Name des UniCats-Systems

height (0) Kriterium des Knopfes:

- 0 Ausleihe
- 1 Katalog
- 2 Sprache
- 3 Jahr

distance (0) Darstellungsform des Knopfes:

- 0 Buchfarbe
- 1 Regal
- 2 Buchlabel

B_UniCats_Legende: Diese zwei Solid Entities stellen die Legenden dar, die dem Benutzer erläutern, durch welche Farbe bzw. welches Label der Bücher die Ausprägungen der jeweils gewählten Kriterien symbolisiert werden. Dabei stellt eine Legende die Buchfarben dar, die andere die Buchlabel. Dazu hat jede Legende vier Frames (für die Ausdrücke *Label* bzw. *blau*, *grün* und *rot* werden die jeweils benutzten Grafiken/ Farben eingesetzt):

	Legende 0	Legende 1
Frame 0	*blau* – Sofort ausleihbar *grün* – Später ausleihbar *rot* – Nicht ausleihbar	*Label* – Sofort ausleihbar *Label* – Später ausleihbar *Label* – Nicht ausleihbar
Frame 1	*blau* – Universitätsbibliothek KA *grün* – Badische Landesbibliothek *rot* – Springer Verlag	*Label* – Universitätsbibliothek KA *Label* – Badische Landesbibliothek *Label* – Springer Verlag
Frame 2	*blau* – Deutsch *grün* – Englisch *rot* – Sonstige	*Label* – Deutsch *Label* – Englisch *Label* – Sonstige
Frame 3	*blau* – Bücher ab 1996 *grün* – Bücher von 1990 bis 1995 *rot* – Bücher bis 1989	*Label* – Bücher ab 1996 *Label* – Bücher von 1990 bis 1995 *Label* – Bücher bis 1989

Aus dem Level-Editor werden folgende Variablen übernommen:

target () Name des UniCats-Systems

height (0) Darstellungsform:

- 0 Buchfarbe
- 1 Buchlabel

B_UniCats_Schleusentuer: Diese Solid Entities sind spezielle Türen, um eine Art Schleuse zu realisieren; das ganze UniCats-System wird in einem durch eine Schleuse (eine Innentür und eine Außentür, jeweils inklusive eines areaportal-Entities) abgetrenntenRaum realisiert, da die vielen hier auftretenden Entities nahe an die Darstellungsgrenzen von Quake 2 heranreichen und nur mit Hilfe einer Schleuse gewährleistet werden kann, daß diese Grenzen im Betrieb nicht überschritten werden (was sonst u.U. zum Programmabsturz führen würde). Eine Tür kann dabei aus einem oder zwei Entities dieser Art bestehen. Bei den Schleusentüren ist vor allem darauf zu achten, daß diese in unterschiedlichen UniCats-Systemen auch unterschiedliche Team-, Tür- und Areaportalnamen besitzen.

Aus dem Level-Editor werden folgende Variablen übernommen:

target () Name des UniCats-Systems

killtarget () Name des zugehörigen Areaportals

height (0) Zuordnung:

- 0 Schleuseninntür
- 1 Schleusenaußentür

speed (100) wie üblich

accel, decel () wie üblich

lip (8) wie üblich

angle (0) wie üblich

wait (3) wie üblich

sounds (0) wie üblich

dmg (2) wie üblich

team () wie üblich, falls zwei Türhälften für eine Tür

B_UniCats_Schleusentrigger: Diese nicht sichtbaren Solid Entities stellen die Auslöser für die Schleusentüren dar; läuft der Benutzer in deren „virtuellen“ Bereich, werden die Schleusentüren betätigt; zusätzlich wird hiermit die An- und Abmeldung eines Benutzers an das UniCats-System geregelt (nur der am System angemeldete Benutzer darf das System benutzen und steuern).

Aus dem Level-Editor werden folgende Variablen übernommen:

target () Name des UniCats-Systems

height (0) Zuordnung:

0 abmelden, Schleuse Innentür außen

1 anmelden, Schleuse Innentür innen

2 Schleuse Außentür außen/innen

sounds (0) wie üblich (An- und Abmeldegeräusch noch nicht implementiert)

B_Buch: Hier handelt es sich um ein internes Entity, das im Level-Editor nicht eingesetzt werden kann. Es wird bei Bedarf erzeugt und stellt ein Model eines Buches dar, das in drei verschiedenen Dicken und zu jeder Dicke mit 36 unterschiedlichen Skins verfügbar ist; diese sind nach folgendem Schema definiert:

		Farbe 1	Farbe 2	Farbe 3
	sofort	1	4	7
Ausleihe	später	2	5	8
	nie	3	6	9
	UB	10	13	16
Katalog	BLB	11	14	17
	Springer	12	15	18
	deutsch	19	22	25
Sprache	englisch	20	23	26
	sonstige	21	24	27
	jung	28	31	34
Jahr	mittel	29	32	35
	alt	30	33	36

Da nicht mehr als 32 Skins pro Model erlaubt sind, wurden je Dicke zwei identische Models mit jeweils 18 Skins verwendet. Die Models sind 32 u hoch, 24 u tief sowie 8 u, 12 u bzw. 16 u dick.

B_UniCats_Model: Obwohl dieses Point Entity nicht speziell zum UniCats-System gehört, wird es hier mit aufgeführt, weil hier wohl der beste Platz dafür ist. Mit Hilfe dieses Entities kann man jedes beliebige Model in den Level einfügen. Das Model erscheint dann an dem Platz, hat aber sonst keinerlei Funktionalität (z.B. werden keine Frames oder Skins verändert).

Aus dem Level-Editor werden hierfür folgende Variablen benötigt:

model (models/bib/garderobe/tris.md2) Name der entsprechenden Model-Datei

mins (5 5 5) Angabe des minimalen Punktes der Bounding Box

maxs (5 5 5) Angabe des maximalen Punktes der Bounding Box

health (0) Gesundheitspunkte des Entities; wenn $\neq 0$, kann das Entity zerstört werden

UniCats_t-Strukturtyp

Bei der Initialisierung eines B_UniCats-Entities wird Speicher für diese Datenstruktur allokiert. Der Zeiger ent→UniCats aller Entities, die zu diesem UniCats-System gehören, wird dann auf diese Datenstruktur gerichtet, so daß alle Entities des UniCats-Systems Zugriff auf diese gemeinsame Datenstruktur haben. Um im Spielverlauf schnell erkennen zu können, ob in diesem Level überhaupt ein Rechercheraum vorhanden ist, wird zusätzlich der Zeiger world→UniCats auf eine der UniCats-Datenstrukturen des Levels gesetzt.

edict_t	*Regale[3]	die drei B_Regal-Entities
edict_t	*UniCats_Entity	das B_UniCats-Entity
edict_t	*Computer	das B_UniCats_Computer-Entity
edict_t	*Legende[2]	die beiden B_UniCats_Legende-Entities
edict_t	*Abmeldungstrigger	der Abmeldungstrigger
edict_t	*Anmeldungstrigger	der Anmeldungstrigger
edict_t	*Schleusentrigger[2]	die beiden Außentür-Schleusentrigger
edict_t	*SchleuseninnentuerMaster	der Master der Schleuseninnentüren
edict_t	*SchleusenaussentuerMaster	der Master der Schleusenaußentüren
char	*Buchmodell[3][2]	die Büchermodel-Dateipfade; der zweite Feldindex wird benötigt, da aufgrund der vielen Skins zwei Models nötig wurden, der erste unterscheidet die Buchdicke:
A_DICKE_DUENN		(0) bis 199 Seiten
A_DICKE_MITTEL		(1) 200 bis 600 Seiten
A_DICKE_DICK		(2) ab 601 Seiten

float	Dicke[3]	hier wird die jeweilige Dicke obiger Models gespeichert
Buch_t	Buecher[MAX_BUECHER]	
		Datenstrukturen für die Bücherdaten von bis zu MAX_BUECHER (=999) Bücher
int	Buecheranzahl	Momentane Gesamtzahl an Büchern im System
Buch_t	*Sortierungen[4][3][MAX_BUECHER]	
		In diesen Listen werden die obigen Bücher nach Kriterien und deren jeweiligen Ausprägungsklasse sortiert; der erste Feldindex entspricht dabei dem Kriterium:
A_AUSLEIHE		(0) mit dem folgenden zweiten Feldindex als Ausprägungsklasse:
A_AUSLEIHE_SOFORT		(0) sofort ausleihbar
A_AUSLEIHE_WARTEN		(1) später ausleihbar
A_AUSLEIHE_NIE		(2) nicht ausleihbar
A_KATALOG		(1) mit dem folgenden zweiten Feldindex als Ausprägungsklasse:
A_KATALOG_UB		(0) Universitätsbibliothek KA
A_KATALOG_BLB		(1) Badische Landesbibliothek
A_KATALOG_SPRINGER		(2) Springer Verlag
A_SPRACHE		(2) mit dem folgenden zweiten Feldindex als Ausprägungsklasse:
A_SPRACHE_DEUTSCH		(0) deutsch
A_SPRACHE_ENGLISCH		(1) englisch
A_SPRACHE_SONSTIGE		(2) sonstige
A_JAHR		(3) mit dem folgenden zweiten Feldindex als Ausprägungsklasse:
A_JAHR_JUNG		(0) ab 1996
A_JAHR_MITTEL		(1) 1990 – 1995
A_JAHR_ALT		(2) bis 1989
int	Anzahl[4][3]	Die Anzahl der Bücher in den obigen Listen
Eingabe_t	Eingabe	Datenstruktur, um die Daten der Eingabemaske zu speichern
qboolean	Eingabe_aktiv	true, wenn die Eingabemaske aktiviert ist
edict_t	*Wiederholung	Hilfsentity, um die Tastenwiederholung bei der Tastatureingabe zu simulieren
edict_t	*Timer	Hilfsentity für den Timer nach Abschicken der Suchanfrage
qboolean	Suche_aktiv	true vom Abschicken der Suchanfrage an bis zur Ergebnisdarstellung in den Regalen
qboolean	Buecherwechsel	true, wenn die alten Bücherdaten gerade gelöscht wurden und die neuen Daten noch nicht alle einsortiert sind

edict_t edict_t edict_t	*Knopf_Regaldarstellung *Knopf_Farbdarstellung *Knopf_Labeldarstellung	die Auswahlknöpfe, die im Moment bezüglich der Regaldarstellung/Farbdarstellung/Labeldarstellung aktiviert sind
int int int	Regaldarstellung Farbdarstellung Labeldarstellung	momentaner Wert der jeweiligen Darstellungsform; entspricht den obigen vier Kriterien-Konstanten (A_$*$)
int	Regale_aktiv	die Anzahl sich momentan bewegender Regale, wenn gerade die Darstellung angepaßt wird (nicht bei Knopfdruck am Regal)
Buch_t	*Buch_gewaehlt	das momentan ausgewählte Buch, sofern der LaserPointer momentan aktiv ist und seit seiner Aktivierung schon ein Buch getroffen hat (ansonsten NULL)
Buch_t	*Buch_zuletzt_gewaehlt	das zuletzt ausgewählte Buch, wenn Buch_gewaehlt=NULL ist
int	Infomodus	Der momentane Buchinformationsausgabemodus:
	INFOMODUS_AUS	(0) keine Ausgabe
	INFOMODUS_KURZ	(1) kurze Buchinformationsanzeige
	INFOMODUS_LANG	(2) ausführliche Buchinformationsanzeige
edict_t	*UniCats_Benutzer	der momentan an das UniCats-System angemeldete Benutzer (dessen ent$\rightarrow$UniCats-Zeiger wird bei der Anmeldung auch auf diese Struktur gesetzt)
qboolean	Abmeldung_in_Arbeit	wenn true, findet die Abmeldung eines Benutzers gerade statt (kann etwas dauern, da erst alle Regale geleert werden, bevor sich die Schleuseninnentür öffnet)
qboolean	Abmeldung_kann_erfolgen	true, wenn alle Regale leer sind und sich die Schleusentür sofort öffnen kann, wenn der Benutzer sich ihr nähert
qboolean	Schleusentuer_offen	true, wenn eine der Schleusentüren geöffnet ist (dann kann die andere Schleusentür nicht mehr geöffnet werden)
qboolean	showLayout	true, wenn eine Layout-Ausgabe erfolgen soll (für Eingabemaske-Hilfebildschirm)

Strukturtyp Buch_t

In dieser Struktur werden die vom Benutzeragenten übermittelten Daten über die einzelnen Bücher gespeichert. Wird ein Buch-Model in die 3D-Welt gesetzt, so wird dessen Zeiger ent→Buchdaten auf die von ihm dargestellten Buchdaten gesetzt. Für alle anderen Entities bleibt dieser Zeiger immer auf NULL.

long	ID	für jedes Buch eine eindeutige ID-Nummer; alle Bücher mit der gleichen ID-Nummer werden als identisch betrachtet (je nachdem, welche Bücher man als gleich betrachten will, sind dies ein und dasselbe Buch nur aus verschiedenen Katalogen, oder auch verschiedene Übersetzungen oder Auflagen eines Buches oder einfach nur ein in einer Bibliothek mehrfach vorhandenes Buch, das unterschiedliche Verfügbarkeitsdaten aufweist – in jedem Fall liegt die Entscheidung, welche Bücher die gleiche ID-Nummer besitzen bei der Schnittstelle zum Benutzeragenten und wird von diesem übernommen)
char	*Autor	der/die Autor/en des Buches
char	*Titel	der Buchtitel
char	*Verlag	der das Buch herausgebende Verlag
char	*Jahr	das Erscheinungsjahr des Buches
char	*Seitenzahl	die Seitenzahl des Buches
char	*ISBN	die ISBN-Nummer des Buches
char	*Sprache	die Sprache, in der das Buch verfaßt wurde
char	*Katalog	der Katalog, in dem das Buch aufgeführt ist
char	*Verfuegbarkeit	der Verfügbarkeitsstatus des Buches; sofort ausleihbar, später (in x Wochen) ausleihbar, nicht ausleihbar (Präsenzexemplar bzw. das Buch kostet xx DM)
byte	A_Dicke	die „Dicke“-Klasse des Buches (die Konstanten A_DICKE_∗)
int	Index	der Index in der Buecher-Liste in UniCats_t
byte	Klasse[4]	die Klasseneinordnung des Buches (hier werden die obigen zwölf Ausprägungsklassen-Konstanten A_∗ benutzt; Bemerkung: jedes Buch wird bezüglich jedes der vier Kriterien einer Ausprägungsklasse zugeordnet)
int	Klassenindex[4]	der jeweilige Index des Listeneintrags dieses Buches in den Sortierungen[4][3]-Listen in UniCats_t (aufgrund der Klassenbildung ist jedes Buch in nur einer der drei Listen des zweiten Feldindexes vorhanden; die Position des Buches in der Sortierungsliste entspricht dann z.B. bezüglich des Kriteriums Verfügbarkeit *Sortierungen[A_AUSLEIHE][Klasse[A_AUSLEIHE]][Klassenindex[A_AUSLEIHE]]*)
edict_t	*Buch_Entity	das Entity, das diese Buchdaten darstellt

Strukturtyp Eingabe_t

Diese Struktur speichert den momentanen Zustand der Eingabemaske bzw. der Tastatureingabe.

qboolean	SHIFT	true, wenn die Shift-Taste gerade gedrückt ist
int	Wiederholung	das zuletzt eingegebene Zeichen, welches das Wiederholungs-Entity aus UniCats_t bei längerem Drücken einer Taste jeden Server-Frame wiederholt
int	aktiv	speichert den momentan aktiven Bereich der Eingabemaske mit Hilfe folgender Werte:
E_FREITEXT		(0) Freitext
E_AUTOR		(1) Autor
E_TITEL		(2) Titel
E_ISBN		(3) ISBN
E_JAHR		(4) Jahr
E_KATALOG		(5) Katalog
E_SUCHEN		(6) Suchen
E_LOESCHEN		(7) Löschen
E_ABBRECHEN		(8) Abbrechen
E_HILFE		(9) Hilfe
E_UB		(10) Katalog: UB (Universitätsbibliothek KA)
E_BLB		(11) Katalog: BLB (Badische Landesbibliothek)
E_SPRINGER		(12) Katalog: Springer (Springer Verlag)
E_HILFEBILDSCHIRM		(13) der Hilfebildschirm ist aktiv
byte	katalog	speichert die momentan ausgewählten Kataloge mit Hilfe der Flags (es können ja gleichzeitig mehrere Kataloge ausgewählt sein):
K_KATALOG_UB		(1) UB
K_KATALOG_BLB		(2) BLB
K_KATALOG_SPRINGER		(4) Springer
char	Daten[5][EINGABEMAXIMALLAENGE+1]	in den ersten fünf E_∗-Bereichen eingegebene Zeichen, wobei EINGABEMAXIMALLAENGE=251 (250 Zeichen und der Cursor)
int	Marke	die Stelle in dem obigen Daten[aktiv]-String, an dem der Cursor im Moment steht
int	Laenge[5]	die momentane Länge der Daten[5]-Strings
int	Sichtfeld[5]	da nicht alle 251 Zeichen der Stringeingabe auf dem Bildschirm dargestellt werden können, wird hiermit der Index des jeweils ersten der 24 sichtbaren Zeichen aus den Daten[5]-Strings gespeichert

Strukturtyp Regal_t

In dieser Struktur werden alle Informationen eines Regals gespeichert. Wie auch schon beim Fahrstuhl oder dem UniCats-System allokiert das Entity B_Regal bei seiner Initialisierung den Speicher für diese Struktur und setzt im weiteren Verlauf der Regalinitialisierung die ent→Regal-Zeiger aller am Regal beteiligten Entities auf diese Struktur, so daß ebenfalls alle Regalteile auf diese gemeinsame Struktur zugreifen können.

edict_t	*Knopf_hoch	der „nach oben“-Knopf
edict_t	*Knopf_runter	der „nach unten“-Knopf
edict_t	*Boden[3]	die drei Regalböden
int	Index_oberer	diese Variablen speichern den momentanen Index des oberen, mittleren und unteren Regalbodens in obiger Boden-Liste (da die Böden sich bewegen können, ist ein Regalboden mal in der Mitte, mal unten und mal oben)
int	Index_mittlerer	
int	Index_unterer	
edict_t	*Regal_Entity	das B_Regal-Entity, also u.a. die Anzeigetafel
edict_t	*Buecherzahl_Anzeige[3]	die drei B_Regal_Buecherzahl_Anzeige-Entities mit den Indizes:
0		Einer
1		Zehner
2		Hunderter
vec3_t	Boden_runter_ganz	die Vektoren, um die ein Regalboden eine (halbe) Position nach unten bzw. oben gefahren wird
vec3_t	Boden_hoch_ganz	
vec3_t	Boden_runter_halb	
vec3_t	Boden_hoch_halb	
vec3_t	Buecherwiege_oben	der Punkt der oberen linken Regalbegrenzung
vec3_t	Buecherwiege_unten	der Punkt der unteren rechten Regalbegrenzung
vec3_t	Buecherwiege_unten_Richtung	die Richtung (normiert) von der unteren rechten zur unteren linken Regalbegrenzung
vec3_t	Buecherwiege_oben_Richtung	dto. von der oberen linken zur oberen rechten Regalbegrenzung
int	Buecherwiege_Laenge	die Streckenlänge zwischen einer rechten und der jeweiligen linken Regalbegrenzung
qboolean	vollstaendig	true, wenn das Regal vollständig initialisiert und somit einsatzbereit ist
Buch_t	**Buecherliste	Zeiger auf die für dieses Regal (ent→Regal) gültige Sortierungen[ent→UniCats→Regaldarstellung][„Regalindex aus UniCats→Regale“]-Bücherliste

int	Listenlaenge	Listenlänge obiger Buecherliste (= ent→UniCats→Anzahl[dto.][dto.])
edict_t	*Buecher_auf_Boden[3][BUECHERAUFBODENMAX]	Liste der Buch-Entities, die momentan auf dem jeweiligen Regalboden stehen (dabei ist BUECHERAUFBODENMAX=13 die maximale Anzahl an Bücher auf einem Regalboden aufgrund der Beschränkung der Gesamtzahl an gleichzeitig sichtbaren Entities)
int	Anzahl_auf_Boden[3]	Anzahl obiger Buch-Entities auf dem jeweiligen Regalboden
int	Oberer_Index	dies ist der Index der Buecherliste nach dem letzten auf dem Regal dargestellten Buch (zu Beginn ist der Wert auf 0 und kann bis zu Listenlaenge annehmen, wenn das letzte Buch der Buecherliste gerade auf einem Regalboden dargestellt wird)
int	Unterer_Index	dies ist entsprechend der Index vor dem ersten dargestellten Buch (zu Beginn auf -1)
int	Indizes_Markierter_Buecher[MAX_BUECHER]	da das ausgewählte Buch mit einer roten Hülle und alle identischen (gleiche ID) und sichtbaren Bücher mit einer grünen Hülle markiert werden, benötigt man diese Liste, um die markierten Bücher dieses Regals zu speichern
int	Boden_bewegt[3]	Anzahl der sich mit dem jeweiligen Regalboden momentan mitbewegenden Entities (Boden und Bücher), damit die Funktion am Ende der Entity-Bewegung erst ausgeführt wird, wenn alle Teile ihre Bewegung beendet haben
int	Fuellen	speichert die restliche Anzahl an Bewegungen aller Regalböden um eine Position nach oben, da bei einem Darstellungswechsel zuerst die auf dem Regal vorhanden Bücher gelöscht werden müssen, bevor die aus der neu zugeordneten Sortierungen[ ][ ]-Liste auf das Regal gestellt werden können und das Regal zusätzlich noch zweimal nach oben fahren muß, damit auf den beiden sichtbaren Regalböden die Bücher aus der neuen Liste stehen; zu Beginn eines Darstellungswechsel steht Fuellen also auf 3 und steht am Ende eines solchen dann auf 0

qboolean	InBewegung	true, wenn sich dieses Regal gerade bewegt (entweder durch einen Darstellungswechsel oder durch einen Druck auf einen der Regalknöpfe)

Die folgenden Funktionen bezüglich des UniCats-Systems werden in `b_UniCats.c`, `b_Ein-Ausgabe.c`, `b_Kommunikation.c` und in `b_Regal.c` definiert. Um die Funktionen in einer verständlichen Reihenfolge zu erklären, wird versucht, eine Benutzung des UniCats-Systems soweit als möglich Schritt für Schritt nachzuvollziehen:

B_SP_UniCats() Dies ist die Spawn-Funktion des B_UniCats-Entities. In dieser wird (im nullten Server-Frame), wie schon oben erwähnt, die UniCats_t-Struktur allokiert und einige Variablen initialisiert. Desweiteren werden hier alle für das UniCats-System notwendigen Geräusche, Models, etc. gecachet sowie auch das UniCats-spezifische Statusbarlayout (UniCats_Layout) eingetragen.

Spawn-Funktionen der weiteren Entities: B_SP_Regal(), B_SP_UniCats_Auswahlknopf(), B_SP_UniCats_Schleusentrigger(), B_SP_UniCats_Schleusentuer(), B_SP_UniCats_Legende(), B_SP_UniCats_Model(), B_SP_UniCats_Computer(), B_SP_Regal_Begrenzung(), B_SP_Regal_Boden(), B_SP_Regal_Knopf(), B_SP_Regal_Buecheranzahl_Anzeige()

B_Regal_initialisieren() Im ersten Frame, nachdem alle Spawn-Funktionen ausgeführt wurden, sucht das B_Regal-Entity mit Hilfe dieser Funktion alle Regalteile zusammen. Die in dessen Spawn-Funktion allokierte Struktur Regal_t wird initialisiert und die ent→Regal Zeiger aller Regal-Teile werden auf diese Struktur gesetzt. Wurde diese Funktion dann erfolgreich beendet, wird Regal→vollstaendig auf true gesetzt und somit das Regal für einsatzbereit erklärt.

B_UniCats_Teile_suchen() Hier werden dann im zweiten Frame die Entities der UniCats_t-Struktur gesucht und initialisiert sowie deren ent→UniCats-Zeiger auf diese Struktur gerichtet. Ist diese Funktion korrekt terminiert, ist das UniCats-System einsatzbereit.

B_UniCats_Schleusentrigger_beruehrt() Bewegt sich ein Spieler auf eine Schleusentür zu bzw. läuft er in den Bereich eines der Schleusentürtrigger, so wird diese Funktion aufgerufen. Je nachdem welchen der vier Trigger er in welchem Zustand berührt, kann dies unterschiedliche Dinge auslösen. Ist es einer der beiden Schleusenaußentürtrigger, wird nur B_UniCats_Schleusentueren_oeffnen() aufgerufen, um die Tür zu öffnen. Kommt der Spieler von außen an die Innentür und gibt es keinen Systembenutzer, wird die Tür auf die gleiche Weise geöffnet. Geht der Spieler dann durch die offene Schleuseninnentür, wird er am inneren Schleuseninnentürtrigger an das System angemeldet. Kommt der Systembenutzer hingegen von innen an die Schleuseninnentür, so wird, wenn UniCats→Abmeldung_kann_erfolgen auf false steht, erst B_UniCats_Benutzer_Abmeldungsarbeit() ausgeführt, um die Regale zu entleeren, ansonsten ist alles in Ordnung und die Innentür wird geöffnet. Kommt der Benutzer dann durch die Innentür und passiert wieder den äußeren Trigger, wird er vom System abgemeldet. Die Schleuseninnentür kann aber immer nur vom Benutzer des UniCats-Systems geöffnet werden, egal ob von innen oder von außen.

B_UniCats_Schleusentueren_oeffnen() Mit dieser Funktion wird, falls UniCats→Schleusentuer_offen auf false steht, diese Variable auf true gesetzt und die entsprechende Schleusentür geöffnet, ansonsten geschieht nichts, d.h. der Spieler muß vor der zweiten Schleusentür solange warten, bis die andere wieder zu ist (das ist ja eben das gewünschte Schleusenprinzip).

B_UniCats_Schleusentrigger_aktivieren() Nachdem eine Schleusentür wieder geschlossen ist, setzt diese Funktion UniCats→Schleusentuer_offen auf false zurück, damit wieder eine Tür geöffnet werden kann.

B_UniCats_Benutzer_Abmeldungsarbeit() In dieser Funktion werden die Regale und die Bücherlisten (B_Buch_leeren()) entleert, mit Hilfe von B_Buch_Infomodus_aus() die Buchinformationsanzeige beim Benutzer ausgeschaltet und zuletzt die Darstellung angepaßt (B_UniCats_Darstellung_anpassen()). Nachdem die Regale wieder zum Stillstand gekommen sind, wird die Schleuseninnentür geöffnet.

B_UniCats_Schleuseninnentueren_beruehrt() Dies ist die Touch-Funktion der Schleuseninnentür, damit der Spieler eine Bildschirmausgabe erhält, wenn er versucht durch diese Tür zu gehen, diese aber aus irgendeinem Grund nicht öffnet (z.B. System schon von einem anderen Benutzer besetzt oder der Benutzer muß beim Verlassen des Raumes noch warten, bis sich die Regale geleert haben).

B_UniCats_Computer_beruehrt() Ist der Spieler nun im Rechercheraum und an das System als Benutzer angemeldet, kann er durch die Berührung des Computer-Entities diese Funktion aufrufen. Sollte im Moment gerade keine andere Suche stattfinden (UniCats→Suche_aktiv ist false), wird B_Eingabe_starten() aufgerufen.

B_Eingabe_starten() Hier wird das Entity für die Tastaturwiederholungen erzeugt, die Tastatureingabe von Zeichen- und Kommandos aktiviert (UniCats→Eingabe_aktiv auf true), die Spielerbewegung „eingefroren“ und B_Eingabe_initialisieren aufgerufen.

B_Eingabe_initialisieren() In dieser Funktion wird die UniCats→Eingabe-Struktur initialisiert und B_Eingabe_darstellen() aufgerufen.

B_Eingabe_darstellen() Diese Funktion kopiert die momentan sichtbaren Zeichen der Eingabedaten (UniCats→Eingabe.Daten[5]) in die im Statusbarprogramms dafür vorgesehenen Configstrings und setzt je nach Zustand der Eingabe (UniCats→Eingabe.aktiv und UniCats→Eingabe.katalog) die richtigen Bilderindizes in die dafür vorgesehenen Statusvariablen. Dann wird noch die Statusvariable, die die Bildschirmausgabe dieses Teils des Statusbarlayouts steuert, auf 1 gesetzt, so daß im nächsten Frame die Eingabemaske auf dem Bildschirm des Benutzers erscheint. Sollte jedoch der Eingabezustand den Hilfebildschirm erfordern, so wird die zuletzt erwähnte Statusvariable wieder auf 0 gesetzt und B_Eingabe_Hilfe_darstellen() aufgerufen.

B_Eingabe_Hilfe_darstellen() Mit Hilfe dieser Funktion wird das Layout für den gesondert zu behandelnden Hilfebildschirm zusammengestellt, an die Engine übermittelt und die Variable UniCats→showLayout auf true gesetzt, so daß (durch späteres Setzen der richtigen Statusvariable in G_Set_Stats()) dieses Layout angezeigt wird.

B_Eingabe_Buchstabe(), B_Eingabe_Kommando() Nachdem jetzt die Eingabe initialisiert und die Eingabemaske ein erstes Mal auf dem Bildschirm erschienen ist, kann der Benutzer durch Tastatureingabe von Kommandozeichen oder Buchstaben bzw. sonstigen Zeichen seine Suchanfrage zusammenstellen.

Beim Drücken einer Kommando-Taste wird B_Eingabe_Kommando() aufgerufen, wo dann entsprechend dem Eingabezustand und dem aktuellen Kommando reagiert wird (z.B. Cursor um eine Position nach rechts oder ins nächste Eingabefeld wechseln), d.h. der neue Eingabezustand festgelegt oder Cursorpositionierungen in Eingabefeldern durchgeführt werden.

Bei einer anderen Taste wird B_Eingabe_Buchstabe() aufgerufen, wo die entsprechenden Zeichen an die Eingabedaten angehängt werden, falls momentan überhaupt eines der Eingabefelder aktiv ist. Dabei werden die deutschen Sonderzeichen (*ä, ö, ß, . . .*) mit Hilfe der Funktion B_ctoq() gleich in den Quake 2-Zeichensatz konvertiert (s.u.).

In jedem Fall wird am Ende beider Funktionen die ent→nextthink-Zeit des Wiederholungsentities gesetzt, so daß dieses eine halbe Sekunde später damit beginnt, den Tastendruck Frame für Frame erneut zu simulieren und zwar solange, bis die aktuelle Taste losgelassen wird (dann wird ein entsprechendes Kommando erteilt, das die ent→nextthink-Zeit des Wiederholungsentities wieder auf 0 setzt). Weiterhin wird in beiden Fällen dann noch B_Eingabe_darstellen() aufgerufen, um die Bildschirmdarstellung an den neuen Eingabezustand anzupassen. Bei dem Kommando Abbrechen wird die B_Eingabe_beenden()-Funktion aufgerufen; beim Kommando Loeschen wieder B_Eingabe_initialisieren und beim Kommando Suchen die Funktion B_Suche_starten().

B_Eingabe_Wiederholung() Das ist die think-Funktion des Wiederholungsentities. Wird diese ausgeführt, dann wird ein erneuter Tastendruck der momentan noch gedrückten Taste simuliert und die eigene ent→nextthink-Zeit auf den nächsten Frame gesetzt.

B_Eingabe_beenden() Beim Aufruf dieser Funktion wird die Spielerbewegung wieder aktiviert, die Tastatureingabe von Zeichen und Kommandos deaktiviert, das Wiederholungsentity gelöscht sowie die Statusvariable zur Eingabebildschirmausgabe bzw. UniCats→showLayout auf 0 bzw. false gesetzt, damit keine Bildschirmanzeige mehr erscheint.

B_atoq(), B_ctoq() Mit Hilfe dieser Funktionen wird ein ASCII-codierter String bzw. ein Char in den „Quake 2-Zeichensatz“ konvertiert. Der Quake 2-Zeichensatz entspricht dabei in großen Teilen dem ASCII-Code, bis auf unsere Erweiterungen bezüglich der deutschen Sonderzeichen. Werden diesen Funktionen die Sonderzeichen als „C-Chars“ bzw. im „C-String“ übergeben, so werden diese durch die entsprechenden Oktalwerte ersetzt, durch die die Zeichen in der Quake 2-Zeichensatzdatei `conchars.pcx` richtig angesprochen werden, so daß nach einer solchen Konvertierung die korrekte Bildschirmausgabe unter Quake 2 erreicht wird. Taucht ein Zeichen auch im erweiterten Quake 2-Zeichensatz nicht auf, so wird dieses durch ein Ersatzzeichen (einen zentrierten Punkt) ersetzt, damit in der Bildschirmausgabe später keine beliebigen Zeichen erscheinen (dies gilt z.B. für Sonderzeichen der Art ê, ø).

B_qtoa() Diese Funktion ist die Umkehrfunktion zu der obigen Funktion B_atoq(). Ein im Quake 2-Zeichensatz vorliegender String wird wieder an den ASCII-Zeichensatz angepaßt.

B_Suche_starten() Diese Funktion sperrt weitere Suchanfragen, indem UniCats→Suche_aktiv auf true gesetzt wird, startet einen Thread mit der Funktion B_Suche_Kommunikation(), um die Kommunikation mit der Benutzeragentenschnittstelle nichtblockierend durchführen zu können und erstellt das UniCats→Timer-Entity, damit bei möglichen Problemen die Suche nach spätestens 3 min beendet wird.

B_Suche_Kommunikation() Diese Funktion ist recht umfangreich. Zuerst wird die zu sendende Suchanfrage formuliert:

select
from␣*Katalogliste*
where␣Freitext="*Daten*"
and␣Autor="*Daten*"
and␣Titel="*Daten*"
and␣Jahr="*Daten*"
and␣ISBN="*Daten*"

Hierbei ist die *Katalogliste* eine durch Kommata getrennte Auflistung der Kataloge, in denen gesucht werden soll; der Maximalfall entspricht also (ohne irgendwelche Freizeichen): UBKA,BLB,Springer
Die *Daten* stehen für die vom Benutzer eingegebenen Daten in UniCats→Eingabe.Daten, wobei diese noch mit Hilfe der obigen Funktion B_qtoa() konvertiert werden müssen. Anschließend wird die Funktion B_Eingabe_beenden() für den Benutzer aufgerufen, damit die Eingabemaske verschwindet.

Jetzt wird, falls die Cvar Online auf 1 steht, die Kommunikationsverbindung zum Server aufgebaut, der die Anfrage entgegennimmt, die Suchanfrage an diesen übermittelt und dessen Ergebnismenge empfangen. Sollte die Cvar Online auf 0 stehen, wird eine von uns per Hand erzeugte Test-Ergebnismenge aus dem lokalen Dateisystem eingelesen. In jedem Fall liegt bei erfolgreicher Kommunikation die Ergebnismenge in Form eines Strings vor. Die Syntax dieses Ergebnisstrings haben wir wie folgt vereinbart:

0*ID*
1*Titel*
2*Autor*
3*Jahr*
4*Verlag*
5*ISBN*
6*Sprache*
7*Verfügbarkeit*
8*Katalog*
9*Seitenzahl*
0*ID*
1*Titel*
⋮

Hierbei stehen die *kursiv* gedruckten Ausdrücke für beliebige Zeichenfolgen bis zum newline-Zeichen, wobei die gesamte Zeile nicht länger als MAX_ZEILENLAENGE-Zeichen (=256) lang sein sollte. Weiterhin werden im folgenden auch nur MAX_BUECHER (=999)-Bucheinträge berücksichtigt. Die Werte von *ID*, *Jahr* sowie *Seitenzahl* sollten nur aus Ziffern bestehen, die Werte von *Titel*, *Autor*, *Verlag* und *ISBN* sollten „normale" Strings sein, der Wert *Sprache* sollte ein Sprachkürzel nach ISO 639 sein und für den Wert *Verfügbarkeit* ist eine spezielle Syntax vereinbart (*Ziffern* ist dabei eine beliebige Ziffernfolge, deren erste Ziffer $\neq$0):

Zeichen	$\Rightarrow$	Bedeutung
0	$\Rightarrow$	sofort ausleihbar
Ziffern	$\Rightarrow$	in *Ziffern* Wochen ausleihbar
-0	$\Rightarrow$	Präsenzexemplar
-*Ziffern*	$\Rightarrow$	das Buch kostet *Ziffern* DM

Um nun die neuen Buchdaten einlesen zu können, müssen zuerst die alten gelöscht werden. Dazu wird zuerst UniCats→Buecherwechsel auf true gesetzt und dann für jeden vorhandenen Eintrag in der Buecherliste UniCats→Buecher die Funktion B_Buecher_leeren() aufgerufen. Dann wird mit Hilfe der Funktion B_Buch_parse() der Ergebnisstring Buch für Buch durchlaufen und somit die Daten in der Bücherliste eingetragen. Ist der Ergebnisstring durchlaufen oder die maximale Bücherzahl erreicht, werden die Bücher in der Bücherliste mit Hilfe der Vergleichsfunktion B_Buch_compare() aufsteigend nach ID-Werten sortiert. Anschließend werden dann noch die klassenbezogenen UniCats→Sortierungen erstellt und die diesbezüglichen Buchdaten in der Bücherdatenstruktur eingetragen. Zum Schluß wird dann UniCats→Buecherwechsel wieder auf false gesetzt und B_Suche_Timer() aufgerufen.

B_Buch_leeren() Da die Speicherbereiche der einzelnen Daten eines Buches beim Eintragen dynamisch allokiert werden, werden diese hier wieder freigegeben und die Variablen der Buch_t-Struktur alle auf 0 bzw. NULL gesetzt.

B_Buch_parse() In dieser Funktion wird die oben beschriebene Syntax des Ergebnisstrings analysiert und die entsprechenden Daten in die Datenstruktur der Bücherliste eingetragen. Es wird immer nur ein Buch bearbeitet. Dabei werden Strings mit Hilfe von B_atoq() konvertiert und dynamischer Speicherplatz allokiert. Um die Sprachkürzel aufzulösen wird die Funktion B_SprachkuerzeltoSprache() aufgerufen.

B_SprachkuerzeltoSprache() Diese Funktion erhält ein Sprachkürzel, sucht die passende Sprache dazu (i.d.R. in Englisch) aus und gibt einen neu allokierten Speicherbereich mit dem Sprachstring zurück.

B_Buch_compare() Hier werden die ID-Daten zweier Bücher miteinander verglichen.

B_Suche_Timer() Diese Funktion wird entweder aufgerufen, weil der Timer abgelaufen ist, oder weil die Suche erfolgreich beendet wurde. Ist der Timer abgelaufen, so wird der Thread beendet. Andernfalls wird der Timer angehalten und B_UniCats_Darstellung_anpassen() aufgerufen. In jedem Fall wird UniCats→Suche_aktiv auf false gesetzt und der Timer entfernt.

B_UniCats_Auswahlknopf_benutzt() Diese Funktion wird aufgerufen, wenn der LaserPointer einen der zwölf Auswahlknöpfe trifft. Sofern dieser Knopf nicht der in seiner Darstellungsgruppe gerade ausgewählte ist, wird der Ausgewählte abgewählt (seine Textur angepaßt), dieser Knopf selbst ausgewählt (ebenfalls die Textur angepaßt, zusätzlich noch als gewählt in der UniCats-Struktur gespeichert) und das Kriterium der betreffenden Darstellungsform angepaßt. Anschließend wird das UniCats-Entity so eingestellt, daß die Funktion B_UniCats_Darstellung_anpassen() in 3 s aufgerufen wird (damit der Benutzer noch Zeit hat, die Auswahl der anderen Darstellungsformen anzupassen, denn nach jedem Aufruf dieser Funktion wird die Zeitspanne wieder auf 3 s eingestellt).

B_UniCats_Darstellung_anpassen() Sind die Regale noch aktiv (UniCats→Regale_aktiv ≠ 0), wird diese Funktion um 1 s verzögert erneut aufgerufen. Ansonsten wird UniCats→Regale_aktiv auf 3 gesetzt, für jedes Regal B_Buecher_Markierung_aus() aufgerufen, die Regal→Buecherliste an die neue Regaldarstellung angepaßt, die weiteren Parameter initialisiert und Regal→Fuellen auf 3 gesetzt. Anschließend wird einmal B_Buecher_Markierung_ein() aufgerufen, um in allen Regalen die markierungen wieder zu aktivieren, jedoch diesmal in den neuen Listen. Dann wird jedes Regal mit B_Regal_hoch() in Bewegung gesetzt und die Texturframes der Bücheranzahl-Anzeige sowie der Ausprägungsanzeige angepaßt. Abschließend werden dann noch die Texturframes der beiden Legenden auf den neusten Stand gebracht.

B_Buecher_Markierung_ein() In dieser Funktion werden alle Bücher, die den gleichen ID-Wert aufweisen wie das momentan ausgewählte Buch, in den jeweiligen Regalen mit deren Index in der Regal→Buecherliste in die Liste Regal→Indizes_Markierter_Buecher eingetragen und diese jeweils mit einem Wert von -1 abgeschlossen.

B_Markierungen_anpassen() Diese Funktion aktualisiert die „Markierungen“ eines Regals. Zum einen sind dies die Frames und die Benutzbarkeit der Regalknöpfe aufgrund der momentan dargestellten Bücherliste; sind im Regal in einer Richtung noch bisher nicht auf dem Regal stehende Bücher vorhanden, so wird dem Benutzer ermöglicht, diese ins Regal zu stellen, indem er den Regalknopf dieser Richtung mit dem LaserPointer beschießt; dabei wird dem Benutzer diese Möglichkeit signalisiert, indem die Texturframes des Knopfes von 0, 1 auf 2, 3 geändert werden (EF_ANIM01 bzw. EF_ANIM23). Zum anderen werden die Markierungen der Bücher aktualisiert; steht ein Buch aus der Liste Indizes_Markierter_Buecher sichtbar auf dem Regal, so wird diesem mit Hilfe des EF_COLOR_SHELL eine grüne Hülle verpaßt (bzw. rote Hülle, wenn es sich um das momentan ausgewählte Buch selbst handelt), ist ein Buch aus der Liste nicht sichtbar, dann beginnt der Pfeil des entsprechenden Knopfes zu blinken (EF_ANIM_ALL).

B_Buecher_Markierung_aus() Diese Funktion schaltet alle durch B_Markierungen_anpassen() eingeschalteten Effekte bezüglich der Büchermarkierung aus und setzt die Liste Regal→Indizes_Markierter_Buecher aller Regale zurück.

B_Regal_hoch(), B_Fuellen_hoch(), B_Regal_hoch∗() Durch den Aufruf der Funktion B_Regal_hoch() (wird auch beim Beschießen des entsprechenden Regalknopfes mit dem LaserPointer ausgeführt) wird der „nach oben“ Knopf in eine gedrückte Position bewegt, weitere Anfragen durch ‚Regal→InBewegung‘ auf true verhindert und alle Regalböden um eine Position nach oben befördert. Dieses Befördern geschieht in einzelnen Schritten mit Hilfe der B_Regal_hoch_∗()-Funktionen und der die Bewegung einleitenden Funktion B_Regal_Boden_bewegen() (damit auch die Bücher auf dem Regalboden mit nach oben fahren):

Zuerst geht der oberste sichtbare Regalboden um eine halbe Position nach oben. Dort wird die zweite Weghälfte in Auftrag gegeben, gleichzeitig aber noch der mittlere Regalboden beauftragt, eine ganze Position nach oben zu fahren. Ist nun der oberste Regalboden ganz oben im Regalgehäuse angekommen (für den Benutzer nicht sichtbarer Bereich), dann werden durch B_Regal_Buecher_loeschen() alle Bücher auf diesem Regalboden entfernt und die in der Bücherliste folgenden Bücher auf dem untersten Regalboden (der unten im Regalgehäuse im nicht sichtbaren Bereich residiert) eingesetzt. Dazu wird zuerst die Bücheranzahl auf dem untersten Boden ermittelt (aufgrund der Regalbreite, der Bücherliste und der maximalen Bücheranzahl auf einem Regalboden begrenzt), die Buch-Entities an den jeweiligen Positionen mit den entsprechenden Orientierungen erzeugt (u.a. mit Hilfe von B_Spawn_Buch()), deren ent→Buchdaten-Zeiger auf die entsprechenden Datenstrukturen gerichtet und auf dem Regal eingeordnet (Regal→Buecher_auf_Boden-Liste, Teamchain mit Regalboden). Dann werden die Markierungen entsprechend angepaßt (B_Markierungen_anpassen()) und der unterste Boden um eine ganze Position nach oben bewegt.

Kommt dann der unterste Regalboden an seiner Endposition an, wird das Regal wieder in die Grundstellung gebracht, d.h. der oberste Boden wird entfernt und auf dem untersten Platz eingefügt. Sollte Regal→Fuellen $\neq$ 0 sein, so wird die Variable dekrementiert und, wenn sie jetzt immer noch nicht 0 ist, B_Fuellen_hoch() angestoßen bzw., wenn sie dann 0 sein sollte, noch B_Buch_Info_aktualisieren() aufgerufen, der entsprechende Regalknopf in seine Ausgangslage befördert (dieser ruft am Ende seiner Bewegung dann B_Regal_weiter() auf) und, falls dies gerade das Leeren des Regals vor der Benutzerabmeldung war, die Schleuseninnentür geöffnet. Ist Regal→fuellen() aber schon von Anfang an 0 (bei Bewegunng durch „normale“ Betätigung des Regalknopfes), wird einfach nur der Knopf in seine Ausgangslage versetzt (und wie oben schon erwähnt anschließend B_Regal_weiter() aufgerufen).

B_Regal_Boden_bewegen() Diese Funktion bewegt einen Regalboden mitsamt seinen Büchern um den angegebenen Weg.

B_Regal_Buecher_loeschen() Diese Funktion entfernt die Bücher eines Regalbodens aus der 3D-Welt und paßt die internen Regaldaten dementsprechend an.

B_Spawn_Buch() Eine Spawn-Funktion für Bücher, bei der u.a. auch die Bounding Box des Buches berechnet wird.

B_Regal_fuellen() Diese Funktion ruft einfach B_Regal_hoch() auf, wird aber benötigt, damit sie als think-Funktion agieren kann (B_Regal_hoch() hat andere Parameterliste).

B_Regal_weiter() Diese Funktion ermöglicht wieder weitere Funktionsaufrufe zur Regalbewegung (Regal→InBewegung auf false).

B_Regal_runter*() Dieser Funktionenkomplex funktioniert entsprechend den oben erläuterten B_Regal_hoch*()-Funktionen, nur daß hier die Möglichkeit des „Füllens“ entfällt.

B_Regal_blockiert() Diese Funktion ist die blocked-Funktion des Regals und verhält sich dementsprechend.

B_Buch_benutzen() Wird ein Buch mit dem LaserPointer getroffen, so wird diese Funktion aufgerufen. Findet gerade der Bücherwechsel statt (UniCats→Buecherwechsel auf true) oder sind die Regale gerade aktiv (UniCats→Regale_aktiv) oder ist das getroffene Buch immer noch dasselbe wie im letzten Frame, geschieht überhaupt nichts. Ist es ein anderes Buch als im letzten Frame, dann wird B_Buecher_Markierung_aus() aufgerufen, UniCats→Buch_gewaehlt auf dieses Buch gesetzt und anschließend wieder B_Buecher_Markierung_ein() sowie B_Buch_Infomodus_aktualisieren() ausgeführt. War der LaserPointer im letzten Frame jedoch ausgeschaltet (UniCats→Buch_gewaehlt auf NULL), so ergeben sich folgenden Möglichkeiten: Wurde das momentan ausgewählte Buch (UniCats→Buch_zuletzt_gewählt) erneut getroffen, so wird B_Buch_Infomodus_wechseln() aufgerufen; war bisher noch überhaupt kein Buch getroffen, so werden B_Buecher_Markierung_ein() und B_Buch_Infomodus_wechseln() aufgerufen; wurde ein anderes Buch als das momentan ausgewählte getroffen, dann wird verfahren wie im Fall, als im letzten Frame ein anderes Buch getroffen war. Allen obigen Fällen ist gemeinsam, daß am Ende dieser Funktion UniCats→Buch_gewählt auf das soeben getroffene Buch gesetzt ist.

B_Buch_Infomodus_wechseln() Diese Funktion setzt die für die kurze bzw. ausführliche Ausgabe der Buchinformationen über die Statusbar notwendigen Statusvariablen fest (dabei wird der Ausgabemodus immer gewechselt, d.h. ist zuvor der kurze Modus aktiviert, wird nun der ausführliche eingeschaltet und umgekehrt). Abschließend wird B_Buch_Info_aktualisieren() aufgerufen.

B_Buch_Info_aktualisieren() Diese Funktion schreibt je nach Infomodus die entsprechenden aktuellen Buchdaten in die dafür vorgesehenen Configstrings, damit die Daten korrekt auf dem Bildschirm erscheinen können.

B_Buch_Infomodus_beenden() Diese Funktion beendet die Anzeige, indem die betreffenden Statusvariablen auf 0 gesetzt werden und anschließend noch mit Hilfe von B_Buecher_Markierung_aus() alle Buchmarkierungen ausgeschaltet werden. Alle weiteren Variablen bezüglich der Büchermarkierungen werden dann noch zurückgesetzt.

B_UniCats_Model_die() Abschließend noch eine Funktion, die mit dem UniCats-System eigentlich nichts tu tun hat. Sie wird aufgerufen, wenn das mit einem ent→health-Wert versehene B_UniCats_Model-Entity „getötet“ wird, also dessen Gesundheit unter 0 abfällt.

C Übersicht über die verwendeten Programme

Erstellung und Kompilierung der DLL-Datei:

- Microsoft Visual C++ v5.0
  `http://www.microsoft.com`

Erstellung und Kompilierung des Levels:

- WorldCraft v1.6a *(Shareware)*
  `http://www.worldcraft.com`
- QBSP3 *(Freeware)*
  `ftp://ftp.idsoftware.com/idstuff/quake2/`
- QVIS3 *(Variante von Steven Boswell, Freeware)*
  `ftp://quake2:fragfest@wdunltd.ml.org/qbspvis.zip`
- ArghRad v1.50 *(Freeware)*
  `http://www.planetquake.com/arghrad/`

Erstellung der Texturen:

- PhotoShop v5.0
  `http://www.adobe.de/products/photoshop`
- Wally v1.25b *(Freeware)*
  `http://home.telefragged.com`
- SkyPaintGE v1.03 *(Trial Version)*
  `http://www.wasabisoft.com`

Erstellung der Models:

- Quake2 Model Editor v0.90b *(Freeware)*
  `http://www.rmdsoftware.com/q2modeller`
- NST v0.9b3 *(Freeware)*
  `http://planetquake.com/q2pmp`
- 3D Studio MAX v2.5
  `http://www.3dstudio.com`

Sonstiges:

- PakExplorer v1.2 *(Freeware)*
  `http://quakestuff.telefragged.com`

D Internet-Adressen

D.1 Quake 2

Nachrichten zu Quake und anderen 3D-Spielen:

- Blue's News
  `http://bluesnews.com`
- Redwood's 3D-News
  `http://redwood.stomped.com`
- Q-Continuum *(Deutschsprachige Seite)*
  `http://www.q-continuum.de`

DLL-Programmierung:

- Quake DeveLS - The Quake Developer's Library Site
  `http://planetquake.com/qdevels`
- Quake 2 DLLs
  `http://www.quake2.com/dll`

Level-Editing:

- The Forge - The official Worldcraft editing site
  `http://planetquake.com/worldcraft`
- Rust - Game Editing and Design
  `http://www.gamedesign.net`
- Quake Workshop2
  `http://www.quake2.com/qworkshop`
- QBSP3, QVIS3 and map-making
  `http://24.0.172.162/whatis/quake2/bspvis.html`

Texturerstellung:

- *Tutorial zu Quake1-Texturen*
  `http://www.qoole.com/tutorial/texture`
- Wally's Home
  `http://home.telefragged.com`

Model-Erstellung:

- Quake 2 Modeling Studio
  `http://www.quake2.com/modeling`
- KrAy-ZeE's Modeling Guide
  `http://planetquake.com/modeling`
- Thinking outside the box *(Tutorial von Paul Steed, Mitarbeiter von id Software)*
  `http://www.loonygames.com/content/1.9/totb`
- Bubbah's Monster Models
  `http://planetquake.com/epoch/bubbah`
- 666's Skins
  `http://www.quake2.com/skins`

D.2 Digital Libraries

- SenseMaker
  `http://sensemaker.stanford.edu`
- VRML-Document-Finder
  `http://fabdo.fh-potsdam.de`
- LyberWorld
  `http://www.darmstadt.gmd.de/~hemmje/Activities/Lyberworld`
- ETD Digital Library Projekt
  `http://csgrad.cs.vt.edu/~nkipp/classes/dl/dl.html`
- UniCats
  `http://wwwipd.ira.uka.de/~unicats/`

Literaturverzeichnis

[1] BALDONADO, Michelle Q W.: *An Interactive, Structure-Mediated Approach to Exploring Information in a Heterogeneos, Distributed Environment*, Stanford University, Dissertation, Dezember 1997

[2] BALDONADO, Michelle Q W. ; WINOGRAD, Terry: Hi-Cites: Dynamically Created Citations with Active Highlighting. In: *Proceedings of ACM CHI 98 Conference on Human Factors in Computing Systems* Bd. 1, 1998, S. 408–415. – http://www.acm.org/pubs/articles/proceedings/chi/274644/p408-baldonado/p408-baldonado.pdf

[3] BOSWELL, Steven: QBSP3, QVIS3 and map-making. – http://24.0.172.162/whatis/quake2/bspvis.html

[4] CHRISTOFFEL, M. ; PULKOWSKI, S. ; SCHMITT, B. ; LOCKEMANN, P.: Electronic Market: The Roadmap for University Libraries and Members to Survive in the Information Jungle. In: *SIGMOD Record (ACM Special Interest Group on Management of Data), Special Section on Electronic Commerce* 27 (1998), December, Nr. 4, S. 68–73. – ISSN 0163–5808

[5] DÄSSLER, Rolf ; PALM, Hartmut: *Virtuelle Informationsräume mit VRML: Informationen recherchieren und präsentieren in 3D*. dpunkt Verlag, 1998

[6] DIX, Alan ; FINLEY, Janet ; ABOWD, Gregory ; BEALE, Russell: *Mensch Maschine Methodik*. Prentice Hall, 1995

[7] FOLEY, James D. ; VAN DAM, Andries [u. a.] : *Computer Graphics: Principles and Practice*. Addison Wesley, 1995

[8] HEMMJE, Matthias: LyberWorld - A 3D Graphical User Interface for Fulltext Retrieval. In: *Proceedings of CHI 95*, 1995

[9] ID SOFTWARE: Quellcode zu Quake2. – ftp://ftp.idsoftware.com/idstuff/quake2/source/

[10] JAKFROST: The Console. – http://planetquake.com/console

[11] KAIN, Nicholas J. ; WINZIG, Thomas: The Quake Map Specs. – http://www.niklata.com/quake/qmapspec.html

[12] KIPP, Neill A. [u. a.] : A Virtual Digital Library Interface with a Spatial Metaphor. – http://csgrad.cs.vt.edu/~nkipp/classes/dl/dl.html

[13] NEUMANN, Dieter: *Spezifikation von Benutzerschnittstellen: Konstruktion konfigurierbarer und konsistenter Benutzerschnittstellen*. VDI-Verlag, 1996

[14] SCHMITT, Alfred: Unterlagen zur Vorlesung *Mensch-Maschine-Dialog I*. Universität Karlsruhe, WS 98/99

[15] SUN MICROSYSTEMS: Homepage. – http://www.sun.com

[16] SUN MICROSYSTEMS: Java Technology Home Page. – http://www.javasoft.com

[17] SYRBE, Rudolf: Vorlesung *Anthropotechnik/Ergonomie als Basiswissen*. Universität Karlsruhe, WS 98/99