Fachhochschule Landshut

Fachbereich Informatik

Game-Engine für eine 3D-Weltraumsimulation

Diplomarbeit

Autor: Markus Liehmann

Betreuer: Prof. Dr. Gudrun Schiedermeier

Erklärung zur Diplomarbeit

(gemäß §31, Abs. 7 RaPO)

Name, Vorname des Studierenden: Liehmann, Markus

Fachhochschule Landshut
Fachbereich Informatik

Hiermit erkläre ich, dass ich die Arbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt, sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

_____ _____

(Datum) (Unterschrift des Studierenden)

Thema:	Game-Engine für eine 3D-Weltraumsimulation
Student:	Markus Liehmann
Studiengang:	Informatik
Aufgabensteller:	Prof. Dr. Gudrun Schiedermeier
Durchgeführt in:	FH Landshut
Semester der Abgabe:	10
Gesperrt bis:	-

Zusammenfassung:

Ziel der vorliegenden Diplomarbeit ist eine Game-Engine für eine Weltraumsimulation zu erstellen, welche für die dreidimensionale Darstellung OpenGL verwendet. Die Objekte der Spielwelt sollen möglichst einfach in Aussehen und Verhalten zu ändern sein und sich außerdem physikalisch korrekt im Weltraum verhalten, d.h. die Bewegung läuft im Vakuum und unter Nullgravitation ab. Der Spieler soll dabei sein Raumschiff per Hauptantrieb und Steuerdüsen bewegen und drehen können.
Im Rahmen der vorliegenden Diplomarbeit werden die theoretischen Grundlagen einer Game-Engine und deren Implementierung anhand eines 3D-Asteroids-Klons erläutert. Auf die dabei verwendeten externen Libraries wird ebenso eingegangen wie auf die Konfigurationsschnittstellen der Engine. Desweiteren werden auftretende Probleme während der Durchführung der Diplomarbeit angesprochen, sowie mögliche Verbesserungen und Erweiterungen der Game-Engine aufgezeigt.

Inhaltsverzeichnis:

Vorwort

Die Geschichte der Videospiele geht bis ins Jahr 1958 zurück. Damals, noch weit vor den ersten kommerziellen Videospielen, versuchte ein Physiker namens William A. Higinbotham mittels eines elektronischen Aufbaus, Führungen in seinem Institut, dem Brookhaven National Laboratory, einem US Nuklear-Forschungslabor in Upton, New York, für Besucher etwas schmackhafter zu gestalten. Seine Idee war einen kleinen analogen Labor-Computer zum Zeichnen und Anzeigen einer Flugbahn eines bewegten Balls auf dem Oszilloskop zu verwenden. Mit der Unterstützung von Robert V. Dvorak entwickelte er anschließend ein Spielsystem, das sie "Tennis for Two" nannten welches somit ein Vorläufer des Klassikers "PONG" war. Wie später in der Begriffsdefinition von "Game-Engine" erörtert wird, handelte es sich hierbei um die erste Game-Engine.
Das welterste, vollinteraktive Videospiel sollte 1962 "Spacewar!" werden, das von Wayne Witanen, J. Martin Graetz und Steve Russell für den Tag der offenen Tür am MIT entwickelt wurde. Andere Programmierer gaben ihre Unterstützung, wie etwa eine Sinus/Cosinus Routine von Alan Kotok, ein Programm mit Namen "Expensive Planetarium" von Peter Samson zum Erzeugen des Sternenfeldes sowie die Gravitationseffekte, erstellt von Dan Edwards. Die Game-Engine von "Spacewar!" zeichnete sich durch 2D-Vektor-Grafik und sehr flexible und vielfältige Einstellungen aus. Ebenfalls erwähnenswert ist die Einführung eines neuen Eingabegerätes, nämlich des ersten Spiele-Joysticks, der von "Spacewar!"-Fans zusammengebaut wurde, um die Schalter des Originals zu ersetzen.
Das erste Videospielpatent bekamen 1968 Ralph Baer, Bob Tremblay, Bob Solomon und Bill Rusch für ihr Spielsystem, das zwischen einigen Schießspielen für ein neuentwickeltes Lichtgewehr als auch Ping-Pong, Volleyball und Football umgeschaltet werden konnte.
Der Klassiker "PONG" von Atari entstand 1972 im selben Jahr in dem Atari gegründet wurde. Ziel des Spieles war mit strichförmigen Schlägern einen eckigen "Ball" ins Aus des Gegners zu schlagen. Als Eigenschaft der Game-Engine von "PONG" könnte 2D-Darstellung mit rudimentären physikalischen Berechnungen genannt werden (simple Reflektion und Bewegung des Balles).
Mit "PONG" begründete Atari den Videospielemarkt und setzte seine Bestrebungen um neue technische Errungenschaften fort.

Ab diesem Zeitpunkt begannen Spiele und deren Game-Engines fortlaufend komplexer in den Bereichen audiovisueller Darstellung und Spiellogik zu werden. 1974 wurde mit "Shark Jaws" von Atari das erste Videospiel mit animierten Charakteren veröffentlicht.
"Speed Freak" der Firma Vektorbeam, erschienen 1977, war ein simples Autorennen, das zur Visualisierung Vektorgrafik verwendete und dessen Game-Engine im weitesten Sinne als die erste 3D-Engine bezeichnet werden konnte.
Asteroids, auf dem die Implementierung der Game-Engine im Rahmen dieser Diplomarbeit basiert, wurde 1979 von Atari veröffentlicht und stellte als perfekte Synergie zwischen Einfachheit und intensivem Spiel einen Meilenstein im Arcade-Videospielebereich dar. Der Spieler musste dabei Gesteinsbrocken durch Schüsse seines Raumschiffes zerstören, welches durch Drehung ausgerichtet und durch Schub bewegt werden konnte. Die Asteroiden brechen dabei in immer kleinere auf, bis sie letztendlich ganz zerstört sind. Dem Spieler konnte neben den Asteroiden auch ein durch eine simple KI gesteuertes UFO gefährlich werden. Um sich aus der Gefahr schnell zu entfernen blieb dem Spieler als letzte Möglichkeit der Hyperspace, der das Raumschiff auf eine zufällige Position auf dem Spielfeld setzt.
Eine ansprechende KI wurde das erste mal im Spiel "Puckman" von Namco implementiert (in den USA unter dem Namen "Pac Man" eingeführt). Der Spieler musste mit seiner Spielfigur in einem Labyrinth alle Punkte fressen, ohne dabei von einem der vier Geister erwischt zu werden. Von diesen Geistern hatte jeder eine eigene KI Routine, wie z.B. dem Spieler den Weg abzuschneiden, ihn direkt zu verfolgen usw.

Der Boom der Homecomputer 1984 versetzte, zusammen mit anderen Faktoren, der Videospieleindustrie den Todesstoß. Einzig Nintendo wagte mit dem NES (Nintendo Entertainment System) eine Wiederbelebung des Marktes, welche allerdings erst eine neue Ära der Konsolen einläutete, als die Homecomputer Ende der 80er langsam den PC's wichen.
Aufgrund der Einschränkungen bei den frühen Mainframes und Mikrocomputern dominierte unter den ersten Computerspielen textuelle Darstellung, welches allerdings nicht unbedingt von Nachteil sein muss, wie sich später herausstellte. So erschienen 1972 die ersten Text-Adventures mit "Hunt the Wumpus", von Gregory Yob, und "Adventure", von Willie Crowther. MUD (Multi User Dungeon), von Roy

Trubshaw 1979 geschrieben, setzt den Grundstein für Multi User Online Games, welche allerdings erst Mitte der 90er wirklich populär werden sollen.
Grafik in Computerspielen tauchte 1980 mit "Mystery House" von On-Line Systems (später: Sierra On-Line) auf, ein Adventure, welches aus Text und einfachen Strichen zur grafischen Darstellung bestand. Richard Garriott programmierte 1979 das erste echte CRPG (Computer Role Playing Game) mit Namen "Akalabeth", dessen Engine ASCII für die Darstellung der Spielkarte und eine Pseudo-3D-Ansicht der Höhlen verwendete.
Durch Verbesserung der Hardware wurde es möglich, die Grafik von Computerspielen immer detaillierter zu gestalten. 1987 erschien mit "Maniac Mansion" von Lucasfilm Games (heute Lucas Arts) das erste Point'n'Click-Adventure, das anstatt eines Parsers für Texturkommandos ein ausgeklügeltes Benutzerinterface verwendete, bei dem mit der Maus Befehle angeklickt und zu Sätzen geformt werden konnten.
Der Begriff "3D-Engine" erfuhr seine Prägung erst mit Erscheinen der populären 3D-Shooter "Wolfenstein 3D" (1992) und "Doom" (1993), beides Spiele von id Software. "Doom" unterstützte als erstes Spiel konsequent einen Mehrspielermodus über Modem, Nullmodem-Kabel oder LAN.
Weitere Informationen zur Geschichte der Video- und Computerspiele können im 8Bit-Museum (vgl. Link [1]) eingeholt werden.

Im weiteren Verlauf entwickelten sich unter den Computerspielen verschiedene Genres, wie Adventures, Shooter, CRPGs (Computer Role Playing Games), Simulationen, Strategiespiele, Sportspiele, Jump'n'Run usw. und diverse Genre-Mixes. Bei jedem Genre oder Genre-Mix muss überlegt werden, welche Bedingungen die jeweils zugrundeliegende Game-Engine erfüllen soll. Es müssen Punkte wie KI, audiovisuelle Darstellung, Spiellogik, Mehrspielertauglichkeit etc. beachtet werden.
Im Falle einer 3D-Weltraumsimulation, die dem Thema der vorliegenden Diplomarbeit zugrunde liegt, muss dabei spezieller Wert auf die dreidimensionale Darstellung der Szene und auf eine realistische Physik gelegt werden. Eine der ersten 3D-Weltraumsimulationen war "Elite", 1985 für den Commodore 64 erschienen. Dieses Spiel bediente sich einer Vektorgrafik, um die 3D-Szenerie zu rendern. Zur Interaktion mit der Spielwelt standen eine Vielzahl Sonnensysteme,

Planeten, Raumstationen etc. zur Verfügung. Der Spieler konnte dabei mit verschiedenen Gütern handeln oder spezielle Aufträge annehmen. Da die damalige Hardware im Vergleich zu heutigen Standards recht dürftig war, besaß das Spiel eine sehr grob implementierte Physik.
Es war nicht möglich, im zeitlichen Rahmen der Diplomarbeit ein komplettes Wirtschaftssystem, Sonnensysteme, Raumstationen, mehrere Schiffstypen usw. zu implementieren, weswegen ein 3D-Asteroids-Klon als Implementierungsgrundlage ausgewählt wurde. Aus denselben zeitlichen Gründen musste ebenfalls auf eine KI und auf einen Mehrspielermodus verzichtet werden. Für ein komplettes Spiel wären außerdem zusätzlich zur Engine noch Level Designer und Art Designer nötig.
Die Diplomarbeit wurde in vier Kapitel unterteilt. Im ersten Kapitel werden die theoretischen Hintergründe einer Game-Engine beleuchtet. Dazu gehören Begriffsdefinition und grundsätzliche Ansätze und Überlegungen zum Design. Das zweite Kapitel deckt eine Einführung in die verwendeten Open-Source Libraries SDL (Simple DirectMedia Layer), SDL_mixer (Sound-Library basierend auf SDL) und der ODE (Open Dynamics Engine) ab. Kapitel drei beschäftigt sich mit der Implementierung der Game-Engine. Dabei wird detailliert eine Besonderheit der Engine, nämlich der Planetoiden-Algorithmus zur Erstellung dynamisch berechneter Asteroiden, vorgestellt. Anschließend wird das Datenmanagement und die Klassenhierarchie, mit Überblick über die einzelnen verwendeten Klassen, beschrieben. Die Dokumentation zu den verwendeten Schnittstellen, durch die die Game-Engine extern unter Verwendung von Textdateien konfiguriert werden kann, befindet sich im vierten Kapitel. Ein abschließendes Fazit befasst sich mit Problemen während der Erstellung der Game-Engine sowie möglichen Verbesserungen und Erweiterungen.

Kapitel 1: Theoretische Grundlagen einer Game-Engine

Was ist eine Game-Engine? Da es zu dieser Frage keine eindeutige Definition von "Game-Engine" gibt, soll in diesem Kapitel der Versuch unternommen werden, eine entsprechende Begriffsdefinition zu finden. Außerdem sollen einige Ansätze und Überlegungen genannt werden, welche beim Design einer Game-Engine beachtet werden sollen.

1.1 Begriffsdefinition

Der Begriff "Game-Engine" könnte sinngemäß als "Spiele-Motor/-Maschine" bzw. "Spiel-Mittel/-Werkzeug" übersetzt werden, also als ein Gebilde, welches für ein Spiel unerlässlich ist, da es dieses sozusagen "antreibt", und somit das Spielen ermöglicht.

1.1.1 Definition "Spiel"

Eine allgemeine Definition wäre:

"Spiel ist eine allgemeine Aktivität des Menschen, die ohne Zwang und Zweck um ihrer selbst ausgeübt wird."
[Prof. Dr. Vernooij „Spiel in seiner Bedeutung für schulisches Lernen", vgl. Link [11]]

Es gibt keine einheitliche Definition. Je nach theoretischer Position muss die subjektive Beteiligung bzw. Zielsetzung sowie die Motivation des Spielenden berücksichtigt werden.

1.1.2 Definition "Engine"

"Anything used to effect a purpose; any device or contrivance; an agent. --Shak."
[Webster's Revised Unabridged Dictionary, vgl. Link [2]]

Eine Engine kann also (unter anderem) ein abstraktes Gebilde sein, welches verwendet wird, um mit ihm ein bestimmtes Ziel zu erreichen.

1.1.3 Definition "Game-Engine"

Der Begriff "Game-Engine" bezieht sich ausschließlich auf Spiele, die als Medium einen Rechner voraussetzen. Unter Einbeziehung der oben genannten Definitionen könnte nun für diese Arbeit folgende allgemeine Definition des Ausdrucks "Game-Engine" formuliert werden:

"Eine Game-Engine ist eine Schnittstelle zwischen Mensch und Spiel, die sich eines Rechners als Medium bedient."

D.h. ein Computerspiel ist zunächst losgelöst vom Rechner zu betrachten. Es hat eigene Regeln, Figuren, Darstellungen usw. Der Rechner wird lediglich als Medium verwendet, um das Spiel für den Spielenden nutzbar zu machen. Die Game-Engine kümmert sich hierbei um die audiovisuelle Darstellung, um die Durchführung der Spielregeln, verwaltet die Spielfiguren und reagiert auf Aktionen des Benutzers. Vergleichbar hierzu wäre beim Brettspiel "Schach" das Material, aus dem das Schachbrett und die Figuren gemacht sind, sowie deren Aussehen bzw. die Darstellung, wie sie der Mensch wahrnimmt, Teil dieses abstrakten Gebildes namens "Game-Engine". Weitere Bestandteile wären in diesem Zusammenhang die Schachregeln und die physikalischen Eigenschaften unserer Welt.

1.2 Ansätze und Überlegungen zum Design einer Game-Engine

Es gibt viele Faktoren, die beim Bau einer eigenen Game-Engine zu bedenken wären. Je nach Komplexität des Spieles, für das eine Game-Engine entwickelt werden soll, kann der Sourcecode schnell mehrere MByte groß werden, an dem mehrere Entwickler beschäftigt werden. Bei besonders grossen Projekten kann es auch nötig sein, dass verschiedene Programmierer an einem einzelnen Subsystem der Game-Engine arbeiten müssen, beispielsweise am physikalischen Bereich der Engine. In diesem Fall werden Subsysteme teilweise ebenfalls als "Engine" bezeichnet, z.B. Rendering Engine, AI (Artificial Intelligence) Engine oder Physics Engine.
Im Folgenden werden einige der Design-Faktoren dargestellt und diskutiert.

1.2.1 Objektorientierung

Soll in der geplanten Engine ein objektorientierter Ansatz verfolgt werden? Diese Frage ist eigentlich grundlegend für jedes Software-Projekt. Zu diesem Thema wurde bereits sehr viel Literatur veröffentlicht, weswegen hier nur eine kleine Zusammenfassung der Vor- und Nachteile beider Ansätze dargelegt wird.

Eine prozedurale Vorgehensweise hat zwar Vorteile in Hinsicht auf Effizienz und Schlankheit des Quellcodes, allerdings verliert sie in grösseren Projekten die Fähigkeiten der Flexibilität, Wartbarkeit und Übersichtlichkeit.
Dies sind jedoch die Vorteile eines objektorientierten Ansatzes. Der Overhead wird zwar durch Klasseneinteilung erhöht, was sich negativ auf die Performance auswirkt, jedoch kann durch Objektorientierung das Projekt besser modularisiert werden, wodurch es leichter zu erweitern und zu warten ist. Im Vergleich zu einem prozeduralen Ansatz fällt bei einem objektorientierten Ansatz mehr Sourcecode an, dieser kann allerdings bei geschickter Klasseneinteilung (Vererbung nicht zu vergessen) sehr viel übersichtlicher strukturiert werden.

Zur Durchführung der Implementierung wurde ein objektorientierter Ansatz in C++ gewählt. Dieser beruht auf der SimpEngine, welche im Buch "OpenGL Game Programming" vorgestellt wird.

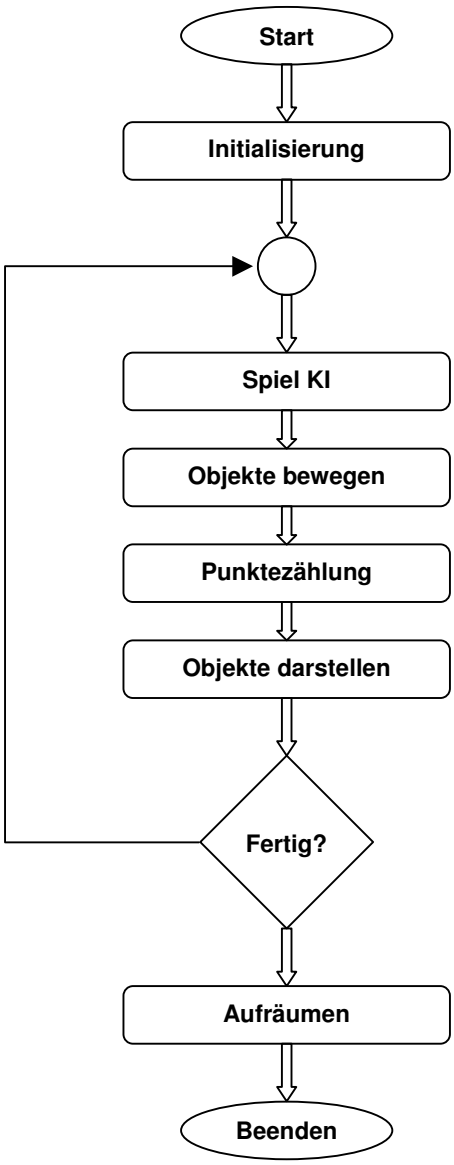
1.2.2 Verwendung externer Libraries von Drittanbietern

Ein anderer Punkt, der beim Design einer Game-Engine beachtet werden sollte, ist die Verwendung von externen Programmbibliotheken. Natürlich kann jede einzelne Funktion selbst implementiert werden. Jedoch ist dies bei Projekten, die plattformunabhängige Lauffähigkeit besitzen sollten, zu zeitraubend, da diverse Funktionen mehrfach implementiert werden müssten. Ebenfalls ist es nicht effizient, selber Programmcode zu erzeugen, der bereits einmal von Dritten geschrieben und getestet wurde (das Rad muss nicht immer erneut erfunden werden). Nachteile von externen Bibliotheken sind zum einen etwaige Lizenzgebühren, die dafür zu entrichten sind, zum anderen die Abhängigkeit von den Entwicklern der Bibliotheken und deren Wartung.
Hierbei bieten sich Open-Source Bibliotheken zur Verwendung an, da der Quellcode zur Verfügung steht und keine Kosten für Lizenzen anfallen. Beispiele für Open-Source Libraries, die für Game-Engines interessant sein könnten, wären SDL (Simple DirectMedia Layer), OpenAL (Open Audio Library), ODE (Open Dynamics Engine).
Im Falle von 3D-Rendering kann auf die Verwendung externer Bibliotheken kaum verzichtet werden, handele es sich nun um OpenGL oder um Direct3D.

Um eine eventuelle Portierung der Game-Engine zu erleichtern wurden für die Implementierung SDL, SDL_mixer und ODE verwendet.

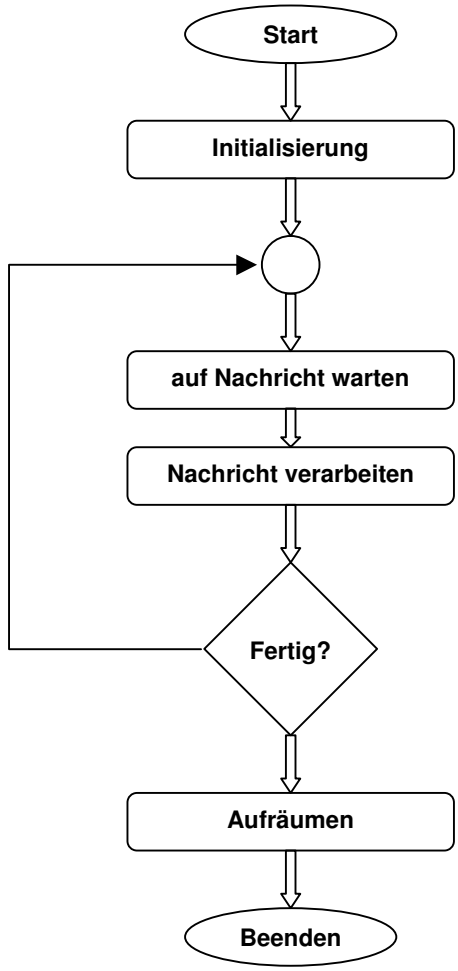
1.2.3 Nachrichtenschleife

Die Grundlage eines Computerspieles bildet eine Schleife mit Abbruchkriterium (Quit). In Zeiten, als Betriebssysteme keine Verwaltung mehrerer Prozesse beherrschten (MS DOS), sah der Ablauf eines Spieles schematisiert wie folgt aus:



Im Gegensatz zu Multi-Thread Applikationen musste hier nicht auf einen Time Slice für die Ausführung gewartet werden. Der Programmierer hatte volle Kontrolle über nahezu jede Operation, die der Prozessor ausführt.

Für Applikationen, die unter einem Betriebssystem laufen, das einen Mehrprozessbetrieb unterstützt, sieht die Schleife etwas anders aus:



Bei einem Multithread-Betriebssystem werden Ereignisse den verschiedenen Applikationen vom Betriebssystem zugeordnet. Somit ist der einzige Zweck der Hauptschleife, diese Ereignisse zu behandeln (Nachrichtenschleife). Die Applikation wartet hierbei auf eine Nachricht vom Betriebssystem, welche ihm mitteilt, dass ein Ereignis eingetreten ist, und leitet diese Nachricht weiter an eine interne Nachrichtenbehandlung.

Das Problem ist nun, dass auf Multithreading-Systemen Applikationen nebeneinander arbeiten müssen und dabei, was Speicherverwaltung und CPU-Zeit betrifft, vom Betriebssystem verwaltet werden. Ein Spiel will allerdings soviele Ressourcen für sich beanspruchen, wie möglich. Würde es das im Mehrprozessbetrieb tun, käme das System zum Stillstand. Also muss ein Kompromiss gefunden werden.
Um nun beispielsweise Tastaturkommandos zu verarbeiten, bestünde die Nachrichtenschleife einer typischen Applikation aus dem Warten auf diese Nachricht (Betätigung der Tastatur), der Übersetzung der Nachricht (Tastaturkommando aus der Message extrahieren) und der Verarbeitung bzw. Weiterleitung der Nachricht (Tastaturkommando interpretieren).
Für ein Spiel ist dies allerdings nicht ausreichend, weil es somit auf die Benachrichtigung durch das Betriebssystem angewiesen ist. Die Applikation muss

somit in der Nachrichtenschleife abfragen, ob sich Messages im Queue der Applikation befinden, und falls nicht, soll der Prozess nicht einschlafen, sondern mit dem nächsten Spielzyklus beginnen. Den oben genannten Kompromiss könnte folglich eine Nachrichtenschleife mit aktivem Warten (busy waiting) bilden. Um dies zu erreichen könnte zwar ein Weg über Interrupts und ISRs (Interrupt Service Routines) beschritten werden, jedoch, wie oben erwähnt, sollte das Ressourcen-Management dem Betriebssystem überlassen werden, um die Lauffähigkeit anderer Applikationen nicht zu beeinflussen.

Ein anderer Ansatz statt "busy waiting" wäre, einen Event-Handler und Callback-Routinen zu verwenden. Dabei arbeitet die Engine in einer Schleife den Spielzyklus ab, während die Ereignisbehandlung über Callback-Funktionen, die in der Game-Engine implementiert werden, mit dieser kommuniziert. Der Event-Handler kann auch in einem eigenen Thread gestartet werden. Unter MS Windows wäre z.B. eine sog. Hook denkbar, welche bestimmte Events (z.B. Tastaturkommandos) an die Game-Engine umleitet. Jedoch wirkt sich diese Vorgehensweise negativ auf die Performance aus, da sämtliche Messages doppelt gefiltert werden, einmal durch das Betriebssystem, und anschließend durch die Hook.
In Bezug auf die Nachrichtenschleife und den Game-Cycle sollte auch über die Art der Timer-Implementierung entschieden werden. Soll die Game-Engine mit maximal möglichen FPS (Frames Per Second) laufen, so wird ein neuer Spielzyklus gestartet, sobald eine bestimmte Zeit vergangen ist, deren Minimum durch die Timer-Granularität bestimmt wird. Werden dagegen die FPS vorgegeben, so wird ein Game-Cycle erst gestartet, wenn mindestens die durch den FPS-Wert vorgegebene Zeit abgelaufen ist.

Bei der Implementierung der hier vorliegenden Game-Engine wurde ein Ansatz mit aktivem Warten verfolgt, da die verwendete SDL-Library keinen Callbackmechanismus für die Ereignisbehandlung zur Verfügung stellt. Im Gegensatz dazu baut z.B. die GLUT-Library auf Callback-Routinen für die Ereignisbehandlung auf. Mit *glutKeyboardFunc()* kann beispielsweise eine Funktion in GLUT registriert werden, die bei Keyboard-Events aufgerufen wird.

1.2.4 Data-Driven Engine

Beim Design einer Game-Engine ist ebenfalls wichtig, ob die Daten von ihr entkoppelt werden sollen oder ob es nützlicher ist, sie im Quellcode zu verankern. Die Daten von der Engine zu trennen bringt Vorteile für das Design des eigentlichen Spieles selbst. Sind viele Designer bei der Erstellung eines Computerspiels beteiligt, so ist es von Vorteil, wenn die Daten (z.B. Modeldaten, Texturen, Sound etc.) direkt abgeändert werden können, ohne dass die Engine neu kompiliert werden muss. Auf diese Weise können auch Teile des Source Codes selber entkoppelt werden, z.B. durch Erstellung einer Scripting Engine (später dazu mehr).
Soll die Engine für mehrere Spiele genutzt werden, bleibt nahezu keine andere Möglichkeit, als eine Data-Driven Engine zu entwickeln. Dies ist natürlich wiederum abhängig von der Komplexität der Spiele, für die die Engine Verwendung finden soll. Einziger Nachteil einer Data-Driven Engine ist das Debugging. Da das Verhalten der Game-Engine stark von den verwendeten Daten abhängt, wird das Debugging mit zunehmender Entkopplung der Daten immer schwieriger. Abhilfe schaffen hier selbst erstellte Debugging Tools, was natürlich wieder mehr Arbeit bedeutet.
Hardcoded Data kann sich lohnen, wenn die Engine nur für ein einziges Spiel entwickelt wird, und nicht viel Designarbeit für das Spiel nötig ist, ansonsten ist allerdings ein Data-Driven Ansatz zu bevorzugen.

Die implementierte Game-Engine soll flexibel konfigurierbar sein, weswegen überwiegend ein Data-Driven Ansatz verfolgt wurde.

1.2.5 Triggering und Scripting

Ein auf Trigger basierendes System ist ein System, welches direkt in der Engine eingebaut wird, festgelegte Konditionen evaluiert und entsprechend reagiert. Im Gegensatz dazu interpretiert eine Scripting Engine ein vorgegebenes Script nach einer festgelegten Syntax, und bestimmt damit das Verhalten der Objekte im Spiel. Trigger bieten den Vorteil, dass ein komplettes Set an Konditionen und Reaktionen spezifiziert und damit sehr gut ein Data-Driven Ansatz für die Game-Engine verfolgt werden kann. Außerdem liefert es überdies einen guten objektorientierten Ansatz, da Objekten bestimmte Eigenschaften zugewiesen werden können, wodurch das System einfach zu erweitern ist. Da keine Syntax erlernt werden muss, ist ein Trigger-based-System für Designer auch leichter zu handhaben, da es verständlicher und leichter zu dokumentieren ist.
Mit zunehmender Komplexität der Game-Engine (und auch der Spiele, für die sie entwickelt wird) macht es jedoch mehr Sinn eine Scripting Engine zu entwerfen, da sie mehr Funktionalität und bessere Skalierbarkeit bei großen Projekten liefert. Eine solche Engine ist auch eine gute Möglichkeit um Source Code aus der Game-Engine zu entkoppeln und somit das Verhalten der Engine sehr flexibel extern zu bestimmen.

Da im Rahmen der Diplomarbeit aus zeitlichen Gründen die Implementierung einer komplexen Game-Engine nicht möglich war, wurde ein einfaches, auf Trigger basierendes System verwendet.

1.2.6 Tools

Zu Beginn des Designs der Game-Engine sollte auch überlegt werden, ob und welche Tools für die Engine sinnvoll sind. Handelt es sich lediglich um ein kleines Projekt, kann auf Tools wie Level-Editor, Model-Editor, Script-Editor usw. verzichtet werden, insbesondere wenn kein Data-Driven Ansatz verfolgt wird. Bei größeren Projekten empfiehlt es sich Schnittstellen für Tools festzulegen und diese auch gleichzeitig mit der Engine zu entwickeln, um es komfortabler zu machen, den Spielinhalt zu verändern, ohne die Engine abzuändern. Wenn die Game-Engine für mehrere Spiele verwendet werden soll, ist der Entwurf von Tools dringend notwendig für eine effiziente Arbeitsweise. Wichtig sind hierbei auch gute Debugging Tools, da sehr viele Daten ausgelagert werden und somit ein reines Debugging des Quellcodes nicht mehr ausreichend ist.

Der zeitliche Rahmen der Diplomarbeit erlaubte nicht die Erstellung von Tools, wie z.B. eines Level-Editors für die Weltraumsimulation. Das einzige Tool, welches verwendet wird, ist ein Texteditor zum erstellen der Konfigurationsdateien.

Kapitel 2: Einführung in die verwendeten Open-Source Libraries

Die Implementierung wurde unter Windows 2000 Professional mit Visual Studio 6.0 in C++ durchgeführt. Wie unter Punkt 1.2.2 bereits erwähnt wurde, sollte es jedoch möglich sein, die Game-Engine möglichst einfach auf andere Betriebssysteme zu portieren. Aus diesem Grund wurde zur Visualisierung und zum Event-Handling die SDL-Library, für Sound und Musik SDL_mixer und für physikalische Berechnungen sowie Kollisionsabfrage die ODE ausgewählt.
Im Folgenden werden diese Open-Source Libraries vorgestellt. Es wird nur auf die für die Implementierung der Game-Engine benötigte Funktionalität eingegangen.

2.1 SDL (Simple DirectMedia Layer)

2.1.1 Überblick

Die SDL-Library ist eine Plattform-übergreifende Multimedia-Programmierschnittstelle (API) für Spiele, Spiele-SDKs (Software Development Kits), Emulatoren, Demos und andere Multimedia-Anwendungen. Sie kann für Videoausgabe, Ereignisbehandlung, Audioausgabe, Abspielen von Audio-CDs, Threads, Timer und Konvertierung der Bytereihenfolge verwendet werden. Portiert wurde die SDL für eine Vielzahl an Betriebssystemen, u.a. Linux, Windows, BeOS und MacOS. Sie wurde in C entwickelt und arbeitet nativ mit C++. Anbindungen für einige andere Programmiersprachen sind ebenfalls verfügbar, wie z.B. Ada, Eiffel, Java, Perl und PHP. Autor der SDL ist Sam Lantinga, welcher von 1999 bis 2001 leitender Programmierer bei Loki Entertainment Software war und seit 2001 als Softwareingenieur bei Blizzard Entertainment arbeitet (vgl. Link [5]).
Obwohl andere Libraries, wie z.B. GLUT (OpenGL Utility Toolkit), auch hätten verwendet werden können, wurde die SDL aufgrund ihrer Flexibilität und Funktionalität ausgewählt.
Für die Implementierung wurde die SDL Version 1.2.6 verwendet.

2.1.2 Initialisierung

Die SDL besteht aus verschiedenen Subsystemen, nämlich Audio, Video, Timer, CDROM und Joystick. Bevor die SDL-Funktionalität zur Verfügung steht, muss die Library mit *SDL_Init()* initialisiert werden.

Z.B. wird durch

SDL_Init (SDL_INIT_TIMER | SDL_INIT_AUDIO | SDL_INIT_VIDEO);

die SDL mit den Timer-, Audio- und Video-Subsystemen initialisiert, d.h. es ist nun möglich Funktionen für die audiovisuelle Darstellung sowie für die Zeitmessung aufzurufen.
Zur Initialisierung weiterer Subsysteme zu einem späteren Zeitpunkt dient *SDL_InitSubSystem()*. Wird die SDL nicht mehr benötigt, so sollen durch *SDL_Quit()* alle benötigten Ressourcen wieder freigegeben werden. *SDL_QuitSubSystem()* nimmt dies für einzelne Subsysteme vor.
Bei Fehlerausgaben und für das Debugging kann mit *SDL_GetError()* der letzte aufgetretene Fehler als String ausgelesen werden.

2.1.3 Video

Die SDL Video Funktionen können erst verwendet werden, wenn zuvor die Library mit SDL_INIT_VIDEO initialisiert wurde. Für die Implementierung der Game-Engine wird nur die OpenGL-Funktionalität der SDL benötigt.

Folgender Code erzeugt ein Fenster, auf das mit OpenGL gezeichnet werden kann:

```
SDL_GL_SetAttribute (SDL_GL_DEPTH_SIZE, 16);
SDL_GL_SetAttribute (SDL_GL_DOUBLEBUFFER, 1);
SDL_SetVideoMode (1024, 768, 32, SDL_OPENGL | SDL_FULLSCREEN);
```

Durch *SDL_GL_SetAttribute()* können einzelne Attribute für OpenGL gesetzt werden, wie im Beispiel die Bit-Größe des Tiefenpuffers und die Unterstützung von Doppelpufferung. *SDL_SetVideoMode()* versucht ein Fenster mit den angegebenen Werten zu erzeugen. Das mit diesem Code generierte Fenster hätte eine Auflösung von 1024x768 Pixeln mit 32 Bit Farbtiefe im Vollbildmodus.
Ist die Doppelpufferung aktiviert, können mit *SDL_GL_SwapBuffers()* der versteckte und der momentan dargestellte Puffer getauscht werden.

Da eine Darstellung des Mauszeigers während des Spiels nicht erwünscht ist, muss dieser mit

```
SDL_ShowCursor (SDL_DISABLE);
```

abgeschaltet werden. Der Cursor ist zwar noch vorhanden, wird jedoch nicht mehr gezeichnet, d.h. die Applikation erhält auch weiterhin Nachrichten über Mausbewegungen.

2.1.4 Window Management

Die SDL stellt einige wenige Funktionen bereit, um Eigenschaften von Fenstern zu verändern, wie z.B. Titel, Symbol etc.

Das folgende Beispiel setzt Titelname und Icon für das Fenster und sorgt dafür, dass alle Eingaben an das Fenster umgeleitet werden:

```
SDL_WM_SetCaption ("Titelname", NULL);
SDL_WM_SetIcon(SDL_LoadBMP("icon.bmp"), NULL);
SDL_WM_GrabInput(SDL_GRAB_ON);
```

Das Grabbing für die Eingabe muss aktiviert sein, um relative Mauskoordinaten zu erhalten, selbst wenn der Mauszeiger den Rand des Fensters erreicht.

2.1.5 Timer

In SDL werden einige plattformübergreifende Funktionen für Timer bereitgestellt. Für die Granularität sollte man sich an mindestens 10 ms orientieren. Die Clock-Ticks mancher Plattformen sind kürzer, jedoch ist dieser Wert am meisten verbreitet. SDL_Init muss mit SDL_INIT_TIMER ausgeführt werden, bevor die Timer-Funktionalität zur Verfügung steht.

Es gibt zwei Möglichkeiten Timer einzubauen, entweder die Abfrage der Zeit, die seit der SDL-Initialisierung vergangen ist, mit *SDL_GetTicks()* oder die Implementierung einer Callback-Funktion, welche in bestimmten Intervallen aufgerufen und durch *SDL_AddTimer()* registriert wird. Durch letzteres können mehrere voneinander unabhängige Timer erzeugt und durch *SDL_RemoveTimer()* wieder entfernt werden. Für *SDL_GetTicks()* ist wichtig zu beachten, dass nach ca. 49 Tagen ein Überlauf auftritt, da der Rückgabewert 32 Bit groß ist.
Um ein Programm für eine bestimmte Zeit anzuhalten kann *SDL_Delay()* verwendet werden.

2.1.6 Event Handling

Die Ereignisbehandlung ist notwendig um Benutzereingaben zu verarbeiten. Sie wird automatisch durch *SDL_Init(SDL_INIT_VIDEO)* initialisiert. Intern werden alle zu behandelnden Events in einer Warteschlange (Event Queue) gespeichert.
Das Kernstück bildet SDL_Event, eine Union aus verschiedenen Arten von Ereignissen, wie Keyboard- oder Maus-Events.

Durch folgenden Code könnte die Message-Queue auf eine gedrückte Taste abgefragt werden:

```
SDL_Event event;
while(SDL_PollEvent(&event))
{
        switch(event.type)
        {
                case SDL_KEYDOWN:
                        printf ("Eine Taste wurde gedrückt!\n");
                        break;
        }
}
```

SDL_PollEvent() holt das nächste anliegende Event und entfernt es aus der Queue. Aufgrund von 'type' kann anschließend bestimmt werden, um welches Ereignis es sich handelt.

Das folgende Beispiel setzt ein Ereignis an das Ende der Warteschlange:

```
SDL_Event event;
event.type = SDL_QUIT;
SDL_PushEvent (&event);
```

Hier wird lediglich ein Event zum Beenden der Applikation hinzugefügt. Analog können auch andere Events, wie Mausbewegungen etc., erzeugt werden.

2.2 SDL_mixer

2.2.1 Überblick

Bei SDL_mixer handelt es sich um eine Bibliothek, welche auf der SDL aufbaut. Mit ihr ist es möglich, Soundfiles verschiedener Formate einzulesen, miteinander abzumischen und auszugeben. Entwickelt wurde sie von Sam Lantinga, Stephane Peter und Ryan Gordon (vgl. Link [6]). Für MP3-Unterstützung wird zusätzlich die SMPEG-Library benötigt.
Wegen ihrer einfachen Handhabung und da sie direkt auf SDL aufsetzt wurde sie für die Implementierung der Game-Engine herangezogen. OpenAL wäre zwar ebenfalls für die Soundausgabe geeignet gewesen, jedoch ist der Umstieg auf diese Library erst sinnvoll, wenn 3D-Sound verwendet werden soll, was für die implementierte Game-Engine nicht nötig war.
Für die vorliegende Arbeit wurde Version 1.2.5 der SDL_mixer-Library verwendet.

2.2.2 Initialisierung

Generell muss vor Verwendung von SDL_mixer das Audio-Subsystem der SDL initialisiert sein.
Durch Initialisierung der SDL_mixer-Library wird der Frequenzbereich, Anzahl der Kanäle (Mono oder Stereo), Größe des Puffers und Format der Soundausgabe festgelegt.

int Mix_OpenAudio (44100, AUDIO_S16SYS*, 2, 4096);*

Hierbei werden 44100 Hz Stereo für die Ausgabe festgelegt, was CD-Qualtiät entspricht. AUDIO_S16SYS gibt an, dass die Wiedergabe in Signed Integer mit der Byte-Ordnung des Systems erfolgen soll. Die Puffergröße beträgt 4 KB und soll nicht zu klein gewählt werden, da sonst Aussetzer bei der Soundwiedergabe auftreten können. Wird sie zu groß angegeben kann eine zeitliche Versetzung beim Abspielen des Sounds festgestellt werden. Wird SDL_mixer nicht mehr verwendet, so sollten mit *Mix_CloseAudio()* alle belegten Ressourcen wieder freigegeben werden.

2.2.3 Sound Wiedergabe

SDL_mixer unterstützt die Ausgabe von Sound-Samples, die im WAVE-, AIFF-, RIFF-, OGG- oder VOC-Format vorliegen.
Das folgende Beispiel soll das Laden und die Ausgabe eines Samples demonstrieren:

```
Mix_Chunk* chunk;
Mix_AllocateChannels (16);
chunk = Mix_LoadWAV ("sound.wav");
Mix_PlayChannel (-1, chunk, 0);
Mix_FreeChunk (chunk);
```

Zunächst muss mit *Mix_AllocateChannels()* die Zahl der Kanäle für das Abmischen der Sounds angegebenen werden. Anschließend wird aus der Datei 'sound.wav' das gewünschte Sample in den Speicher geladen und mit *Mix_PlayChannel()* einmal ohne Wiederholung (letzter Wert 0) auf dem nächsten verfügbaren Kanal (erster Wert -1) abgespielt. Da das Sample nicht mehr benötigt wird, kann es mit *Mix_FreeChunk()* wieder freigegeben werden.
Die Wiedergabe von Samples kann auf einem beliebigen Kanal erfolgen, jedoch ist es nicht möglich zwei auf dem gleichen Kanal abzuspielen.
Zum Starten bzw. Stoppen von Kanälen können Fade-In- bzw. Fade-Out-Effekte verwendet werden. Man kann jeden Kanal auch pausieren und weiterführen, wie im folgenden Beispiel dargestellt:

```
Mix_FadeInChannel (7, chunk, -1, 3000);
SDL_Delay (10000);
Mix_Pause (7);
SDL_Delay (5000);
Mix_Resume (7);
SDL_Delay (10000);
Mix_FadeOutChannel (7, 3000);
```

Kanal 7 spielt das Sample 'chunk' in einer Endlosschleife (Schleifenzahl -1) ab. Dabei wird die Lautstärke über drei Sekunden (3000 ms) stetig von 0 bis zum Maximum erhöht. Nach zehn Sekunden (*SDL_Delay(10000)*) wird der Kanal pausiert und in fünf Sekunden wieder weiter abgespielt. Zehn Sekunden später wird die Lautstärke des Kanals über drei Sekunden hinweg stetig bis 0 reduziert und der Kanal angehalten.
Durch Aufruf von *Mix_HaltChannel()* könnte der Kanal sofort ohne Verzögerung angehalten werden. Wird als Kanalnummer -1 übergeben, werden alle Kanäle, die gerade aktiv sind, gestoppt.

2.2.4 Sound Effekte

SDL_mixer stellt mit Mix_RegisterEffect und Mix_SetPostMix zwar Funktionen zur Verfügung, um eigene Effekt-Funktionen einzubauen und diese auf einzelne Kanäle bzw. direkt auf die Soundausgabe nach der Abmischung (post mix) anzuwenden, jedoch wurde zur Implementierung nur der bereits in der SDL_mixer-Library vorhandene Distanz-Effekt verwendet.
Ein Effekt für die Positionierung einer Geräuschquelle ist ebenfalls enthalten, für echtes 3D-Audio sollte allerdings OpenAL verwendet werden, da es eine bessere Funktionalität und größere Genauigkeit bereitstellt.

Der Distanz-Effekt kann wie folgt auf einem Kanal registriert werden:

Mix_SetDistance (7, 127);

Die Distanz wird für Kanal 7 auf 127 gesetzt. Die Werte müssen im Intervall [0; 255] sein, wobei 0 minimale Entfernung und maximale Lautstärke bedeutet und somit den Effekt wieder vom Kanal entfernt.

2.2.5 Musik Wiedergabe

Eine besondere Stellung in der SDL_mixer-Library nimmt die Musikausgabe ein. Für diese sind ein eigener Kanal und eigene Funktionen verfügbar. Unterstützt werden die Formate WAVE, MOD, MIDI und OGG.

Das folgende Beispiel lädt ein Musikstück 'music.ogg' in den Speicher und spielt es in einer Endlosschleife ab:

```
Mix_Music* music;
music = Mix_LoadMUS("music.ogg");
Mix_PlayMusic (music, -1);
```

Alle Funktionen, die zum Abspielen von Samples dienen, existieren in analoger Form auch für Musik.

2.3 ODE (Open Dynamics Engine)

2.3.1 Überblick

Diese Open-Source Bibliothek stellt Funktionen für die Simulation artikulierter Rigid Body Dynamics zur Verfügung, d.h. es können verschiedene dreidimensionale Körper beliebiger Form und Masse miteinander durch unterschiedliche Gelenke (Joints) kombiniert werden. Die ODE übernimmt dabei alle physikalischen Berechnungen sowie die Kollisionsabfrage. Entwickelt wurde die Library von Russell Smith (vgl. Link [9]).
Da die ODE bereits in Open-Source Spielen wie "Scorched 3D" oder "Racer" erfolgreich zum Einsatz kam, wurde sie auch für die Implementierung der Game-Engine verwendet.
Als Annäherung an die Objekte in der implementierten 3D-Weltraumsimulation dienen ausschließlich Sphären. Joints sind hierbei nur für die Kollisionsabfrage nötig (Contact Joints), weswegen die weiteren Arten der Gelenke nicht erläutert werden.
Die Implementierung wurde mit ODE Version 0.039 durchgeführt.

2.3.2 World

Das World-Objekt fungiert in der ODE als Container für Rigid Bodies und Joints. Objekte, die sich in unterschiedlichen Worlds befinden, können nicht miteinander in Interaktion treten. Alle Objekte einer Welt existieren zum selben Zeitpunkt, somit kann es für manche Applikationen notwendig sein, mehrere World-Objekte zu kreieren, um Simulationen in unterschiedlichen Zeitebenen ablaufen zu lassen.

Eine neue Welt kann mit *dWorldCreate()* erzeugt und mit *dWorldDestroy()* zerstört werden. Mit *dWorldStep()* wird ein Simulationsschritt mit angegebener Zeitdauer durchgeführt, d.h. alle Objekte werden entsprechend ihrer Geschwindigkeit und der wirksamen Kräfte bewegt. Der Vektor der Gravitationskraft wird mit *dWorldSetGravity()* gesetzt, die Voreinstellung ist Nullgravitation.

2.3.3 Rigid Bodies

Alle Körper, die an der Simulation teilnehmen, müssen der World hinzugefügt werden. Der nachfolgende Code fügt eine Kugel in eine Welt mit Nullgravitation ein und addiert eine Kraft und ein Drehmoment, die für den nächsten Worldstep auf die Kugel wirken sollen:

```
dWorldID world;
dBodyID body;
dMass mass;
const float* position;
world = dWorldCreate ();
body = dBodyCreate (world);
dBodySetPosition (body, 1.0f, 2.0f, 3.0f);
dMassSetSphereTotal (&mass, 0.05f, 1.0f);
dBodySetMass (body, &mass);
dBodyAddRelTorque (body, 5.7f, 8.12f, 2.0f);
dBodyAddRelForce (body, 71.45f, 12.2f, 5.9f);
dWorldStep (1.0f);
position = dBodyGetPosition(m_ODEBody);
```

Nach der Generierung der Welt für die Simulation wird der Rigid Body mit *dBodyCreate()* hinzugefügt. *dBodySetPosition()* gibt dem Body die gewünschte Position. Die Form des Körpers wird durch seine Masse bestimmt. Da für dieses Beispiel nur eine Kugel generiert wird, kann die Masse mit *dMassSetSphereTotal()* gesetzt werden. Einheiten sind nebensächlich, denn sie hängen vom gewählten Maßstab für das Koordinatensystem ab. Mit *dBodySetMass()* wird dem Body die gewünschte Masse zugewiesen. Die Sphäre hat also nun die Eigenschaften einer Kugel mit Masse 0.05 und Radius 1.0. Anschließend werden die Vektoren für das Drehmoment und einer linear wirkenden Kraft als Vektoren addiert, die relativ zum Koordinatensystem des Bodies angegeben werden. Die Größe des Drehmoments ergibt sich aus der Länge des Vektors, die Achse aus dem Vektor selber. Um Kräfte in globalen Koordinaten zu spezifizieren, kann *dBodyAddForce()* bzw. *dBodyAddTorque()* aufgerufen werden.

Sämtliche akkumulierten Kräfte werden nach dem nächsten Worldstep auf 0 gesetzt. In diesem Beispiel wurde ein Simulationsschritt mit einer Dauer von 1.0 Sekunden durchgeführt und danach mit *dBodyGetPosition()* die neue Position der Kugel ausgelesen.
Obwohl es jederzeit möglich ist, direkt die Position und Rotation für Körper zu setzen, sollten die jeweiligen Funktionen nur zur Initialisierung verwendet werden, um die Integrität der Simulation nicht zu kompromittieren. Die Angabe von Rotationen erfolgt in Quaternionen oder in 3x3 Matrizen.

2.3.4 Kollisionsabfrage

Die ODE besteht aus zwei Teilen. Zum einen aus der zuvor beschriebenen Simulation durch eine World und in dieser enthaltenen Bodies, zum anderen aus der Kollisionsabfrage. In dieser ist die Form eines Körpers wichtig, um herauszufinden, ob sich zwei Körper berühren.

2.3.4.1 Geometrie Objekte (Geoms)

Ein Geometrie Objekt kann die geometrische Figur eines einzelnen Rigid Bodies repräsentieren (z.B. eine Sphäre oder ein Quader) oder andere Geoms beinhalten. Letzteres ist ein spezielles Form und wird "Space" genannt. Geoms können auf Kollisionen überprüft werden und liefern 0 oder mehr Kontaktpunkte zurück. Es gibt zwei Arten von Geoms, nämlich "placeable Geoms" und "non-placeable Geoms". Erstere besitzen einen Positionsvektor und eine Rotationsmatrix, die während der Simulation verändert werden können. Position und Rotation von non-placeable Geoms können nicht manipuliert werden, da diese für statische Eigenschaften der Umgebung verwendet werden, wie z.B. Boden und Wände.

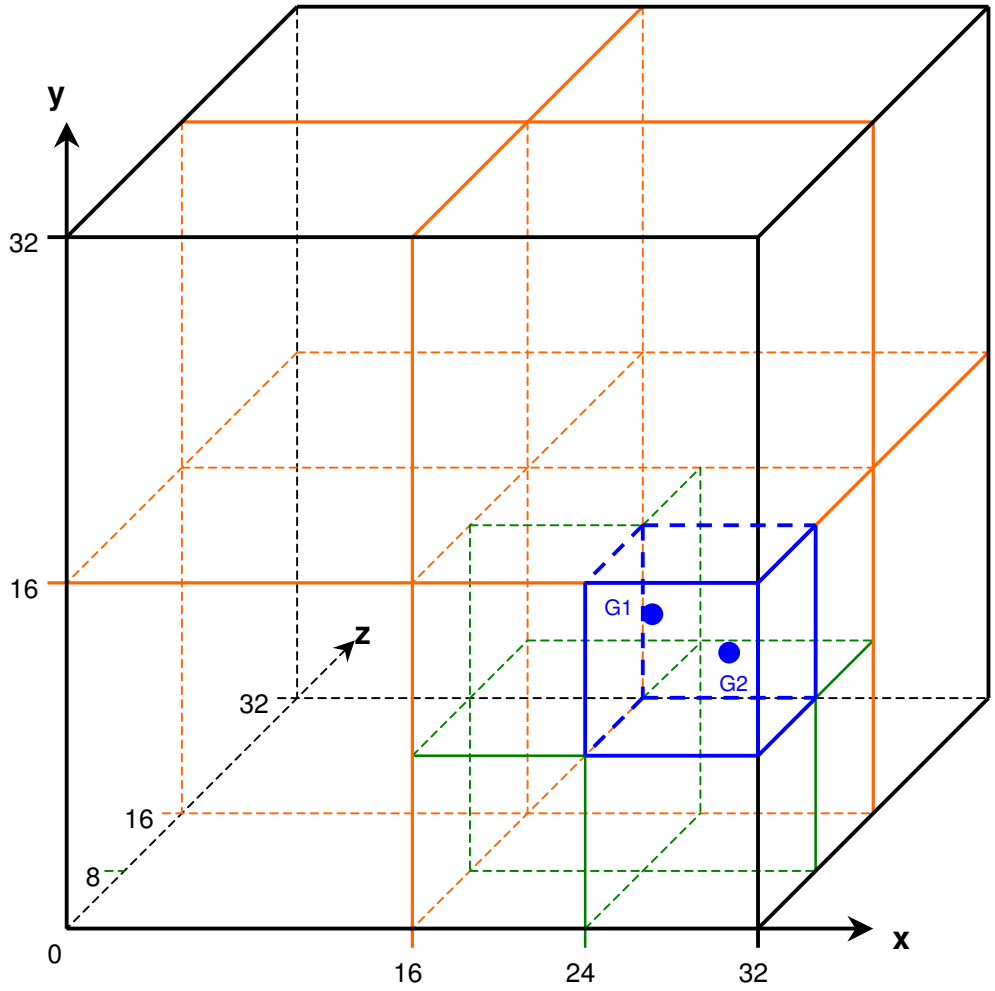
Geometrie Objekte werden in verschiedene Klassen eingeteilt, wie "Sphere", "Box" etc. Spaces sind spezielle optionale Geoms, die andere Geoms enthalten können. Sie entsprechen somit der World im Simulationsteil der ODE. Ohne Spaces müssen sämtliche Geoms miteinander auf Kollision geprüft werden. Durch Spaces und Space-Hierarchien kann die Zahl der benötigten Vergleiche verringert werden.

Im Folgenden wird eine Sphäre (placable) und eine Ebene (non-placable) in einem Hash Space erzeugt:

```
dSpaceID space;
dGeomID sphere;
dGeomID plane;
space = dHashSpaceCreate (0);
dHashSpaceSetLevels (space, 0, 4);
sphere = dCreateSphere (space, 1.0f);
plane = dCreatePlane (space, 0, -1.0f, 0, -5.0f);
```

Die Verwendung eines Hash Spaces beschleunigt die Kollisionsabfrage. Dieser Space teilt den dreidimensionalen Raum in verschieden große Zellen ein. Im Beispiel wird die Größe der Zellen mit *dHashSpaceSetLevels()* gesetzt. Maximum ist hierbei $2^4 = 16$, Minimum $2^0 = 1$, d.h. der Raum wird in Zellen mit 16 Einheiten Kantenlänge eingeteilt. Diese Zellen werden wiederum in kleinere mit 8 Einheiten eingeteilt, danach in 4 usw. bis das Minimum erreicht ist. Mit Hilfe der dadurch aufgespannten Octtrees (Baum in dem jeder Knoten acht Folgeknoten hat) kann sehr schnell verglichen werden, ob zwei Geoms Kandidaten für eine Kollision sind.
Der Sphäre wird ein Radius von 1.0 zugewiesen, die Ebene befindet sich im Beispiel senkrecht zur Y-Achse und schneidet diese bei -5.
Um einen Nutzen aus der Kollisionsabfrage zu ziehen, muss mit *dGeomSetBody()* dem Geom ein Body zugewiesen werden. Anschließend besitzen beide dieselbe Position und Rotation.

Das folgende Beispiel soll die Kollisionsabfrage mit Hilfe eines Octtrees verdeutlichen:



Die beiden Geoms G1 und G2 befinden sich im blau markierten Kubus.
Für dieses Beispiel wäre der Hash Space mit einem Minimum von 3 und einem Maximum von 5 initialisiert worden, d.h. der schwarz gezeichnete Würfel bildet die Wurzel des Octtrees und besitzt eine Kantenlänge von $2^5 = 32$. Die orange hervorgehobene Einteilung repräsentieren die acht Folgeknoten, nämlich acht Würfel mit einer Kantenlänge von $2^4 = 16$. Durch die kleinste Einteilung (grün) in Kuben mit Kantenlänge $2^3 = 8$ kann nun festgestellt werden, welche Geoms sich in gleichen Zellen befinden und Kandidaten für eine Kollision sind.

2.3.4.2 Kontaktpunkte und Contact Joints

Kollidieren zwei Objekte miteinander werden ein oder mehrere Kontaktpunkte erzeugt und durch Contact Joints repräsentiert. Contact Joints sind temporäre Gelenkstellen, die dazu dienen, das Verhalten zwei kollidierender Geoms zu beeinflussen. Nach jedem Worldstep sollten die erzeugten Contact Joints wieder gelöscht werden, da ansonsten Objekte ggf. aneinander "kleben" bleiben.
Nun können mit *dCollide()* paarweise sämtliche Geoms auf Kollisionen überprüft werden, was jedoch zu einer Ausführungszeit von $O(n^2)$ für n Geoms führt. Im Gegensatz dazu sinkt die benötigte Zeit auf O(n) wenn Hash Spaces verwendet werden.

Das folgende Beispiel soll eine Kollisionsabfrage in einem Hash Space 'space' demonstrieren:

```
dJointGroupID contactgroup;
dSpaceCollide (space, NULL, &nearCallback);
dWorldStep (world, 0.01f);
dJointGroupEmpty (contactgroup);
```

Eine Joint Group ist ein Container für verschiedene Joints und wird für die Callback-Funktion benötigt, die durch *dSpaceCollide()* aufgerufen wird, wenn zwei Geoms Kandidaten für eine Kollision sind. Nach dem nächsten Simulationsschritt werden die erzeugten Contact Joints durch *dJointGroupEmpty()* wieder gelöscht.

Die verwendete Callback-Funktion kann z.B. so aussehen:

```
void nearCallback (void *data, dGeomID o1, dGeomID o2)
{
        dContact contact[10];
        int n = dCollide (o1, o2, 10, &contact[0].geom, sizeof(dContact));

        if (n > 0)
        {
                for (int i = 0;  i < n;  i++)
                {
                        contact[i].surface.mode = dContactBounce;
                        contact[i].surface.bounce = 1.0f;
                        dJointID c = dJointCreateContact (world,
                                contactgroup, &contact[i]);
                        dJointAttach (c, dGeomGetBody (o1), dGeomGetBody (o2));
                }
}
```

Diese Funktion erhält als Parameter von *dSpaceCollide()* die IDs der beteiligten Geoms sowie einen Pointer auf ggf. notwendige benutzerspezifische Daten. Die Anzahl der Kontaktpunkte wird mit *dCollide()* bestimmt.
Hier im Beispiel wird das Maximum an generierten Kontaktpunkten auf zehn festgelegt. Sind Kontaktpunkte vorhanden, erhält jeder Kontaktpunkt physikalische Eigenschaften, wie z.B. hier durch die Angabe von 'bounce', die bestimmt, wie stark die Geoms voneinander abprallen. Abschließend wird aus dem Kontaktpunkt ein Contact Joint erzeugt, der Joint Group hinzugefügt und beide Geoms mit *dJointAttach()* aneinander befestigt.

Kapitel 3: Implementierung der Game-Engine

Als Name für die 3D-Weltraumsimulation wurde "ArP" (Artificial Planetoids) gewählt. Die "ArP-Engine" genannte Game-Engine unterstützt frei wählbare Maus- und Tastatureingaben und stellt ein konfigurierbares OpenGL-Fenster zur Verfügung. Für das Spielfeld, der "Arena", wird eine Skybox bereitgestellt, die für die Darstellung des Weltalls verwendet werden kann. Explosionen können durch das implementierte Partikelsystem visualisiert werden. Eine Besonderheit der ArP-Engine ist die Generierung zufallsgesteuert berechneter Planetoiden-Models. Der Algorithmus hierzu wird im folgenden Unterpunkt erläutert. Anschließend erfolgt eine Darstellung des Datenmanagements der Game-Engine und eine Übersicht über die einzelnen implementierten Klassen und ihrer Funktionalität.

3.1 Planetoiden-Algorithmus

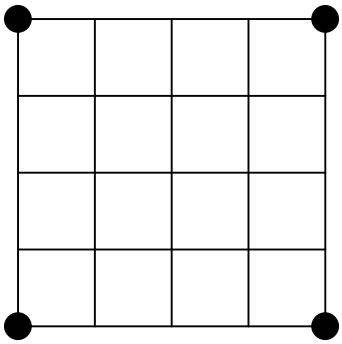
Die Berechnung der Vertices für das Rendering eines Planetoiden erfolgt in zwei Schritten. Zuerst wird durch den Diamond Square Algorithmus ein Gitter mit Höhenangaben erzeugt, das eine fraktale Landschaft beschreibt. Danach werden vier dieser Gitter auf eine besondere Sphäre, einem OQTM (Octahedral Quaternary Triangular Mesh), projeziert, wodurch eine Form ähnlich der eines Felsbrockens entsteht.

3.1.1 Diamond Square Algorithmus

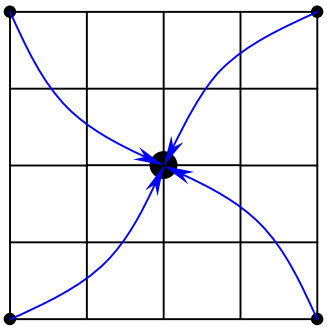
Der Diamond Square Algorithmus ist ein fraktaler Algorithmus, d.h. die durch ihn generierten Strukturen zeichnen sich durch Selbstähnlichkeit aus. Er wird zur Berechnung zufälliger Landschaften und Plasmawolken angewandt und wurde zuerst 1982 in "Computer Rendering of Stochastic Models" von Alain Fournier, Don Fussell, und Loren Carpenter beschrieben (vgl. Link [4]).

Der Algorithmus gliedert sich in folgende Schritte:

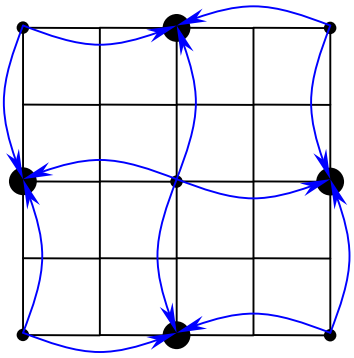
1. Die vier Eckpunkte eines Gitters werden mit Zufallswerten gefüllt.



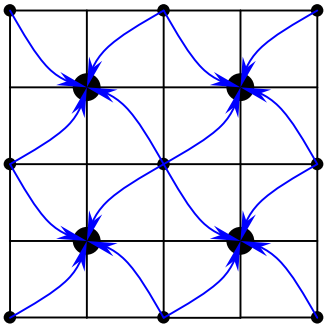
2. Aus diesen Eckpunkten wird der Mittelwert bestimmt, ein zufälliger Wert addiert und dem Mittelpunkt des Gitters zugewiesen. Dies ist der erste "Diamond Step", da durch diesen Schritt eine diamantförmige Struktur auf dem Gitter entsteht (allerdings momentan noch nicht ersichtlich, da die vier entstandenen Diamantformen über das Gitter hinausgehen).

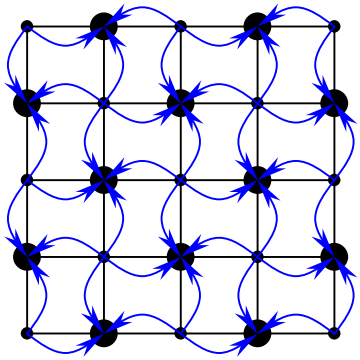


3. Aus den vier Punkten der Diamanten wird jeweils ein Mittelwert generiert und ein zufälliger Wert addiert. Dies ist der erste "Square Step", da hierdurch ein Muster aus Quadraten auf dem Gitter erzeugt wird.



4. Der Diamond Step wird nun auf jedes einzelne der Quadrate angewandt und darauffolgend wieder der Square Step. Dies wird solange fortgeführt, bis das gesamte Gitter mit Werten gefüllt ist.



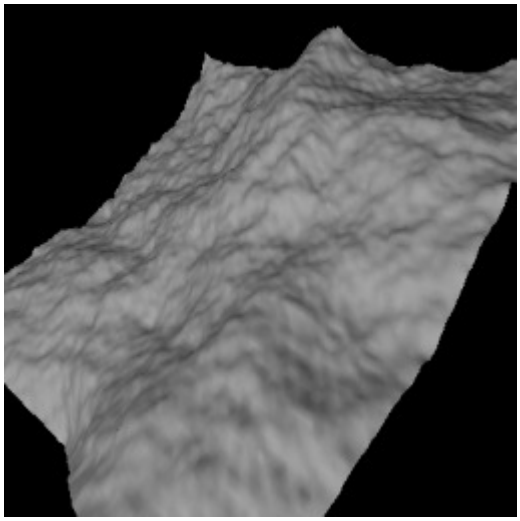


Es können nur Gitter mit 2^n+1 Kantenlänge erzeugt werden. Die Iterationstiefe wird durch n angegeben.

Folgende Parameter sind für den Diamond Square Algorithmus nötig:

Iteration:	Bestimmt die Größe des Gitters
Initialisation:	Bestimmt das Intervall für die Zufallswerte, mit denen die Eckpunkte des Gitters initialisiert werden
Roughness:	Bestimmt den Initialwert für das Intervall, aus dem die Zufallswerte stammen, die den Mittelwerten hinzugefügt werden; desto größer die Roughness, desto zerklüfteter das Terrain

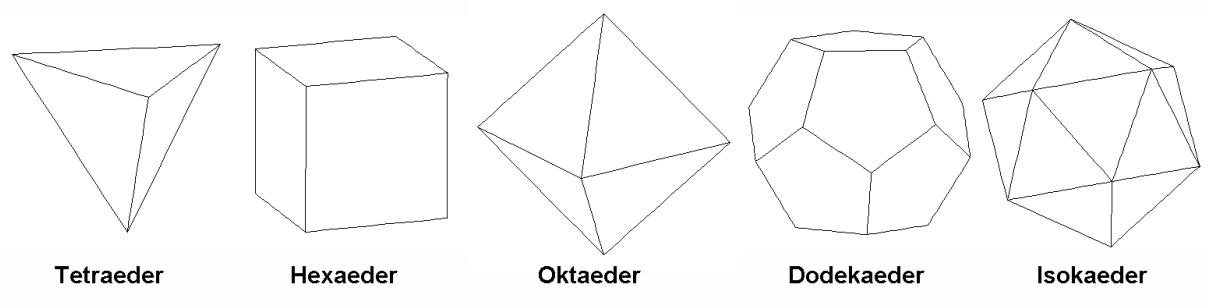
Pro Iterationsschritt wird eine neue Roughness berechnet, indem der aktuelle Wert quadriert wird. Je höher der momentane Iterationsschritt ist, desto kleinere Änderungen an den berechneten Mittelwerten werden vorgenommen. Der Betrag der Roughness muss zwischen 0 und 1 liegen um sicherzustellen, dass das grobe Aussehen der Landschaft durch den Initialisation- und Roughness-Wert vorgegeben wird, und keine großen Änderungen im Detail des Terrains durchgeführt werden.



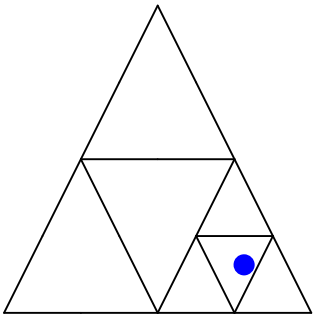
Mit Diamond Square erzeugte Landschaft
(Iteration 7, Roughness [-0.5; 0.5], Initialisation [-0.5; 0.5])

3.1.2 Octahedral Quaternary Triangular Mesh (OQTM)

Eine Sphäre kann durch iterative Unterteilung in regelmäßige, untereinander kongruente Vielecke angenähert werden. Als Grundlage wird einer der platonischen Körper verwendet:



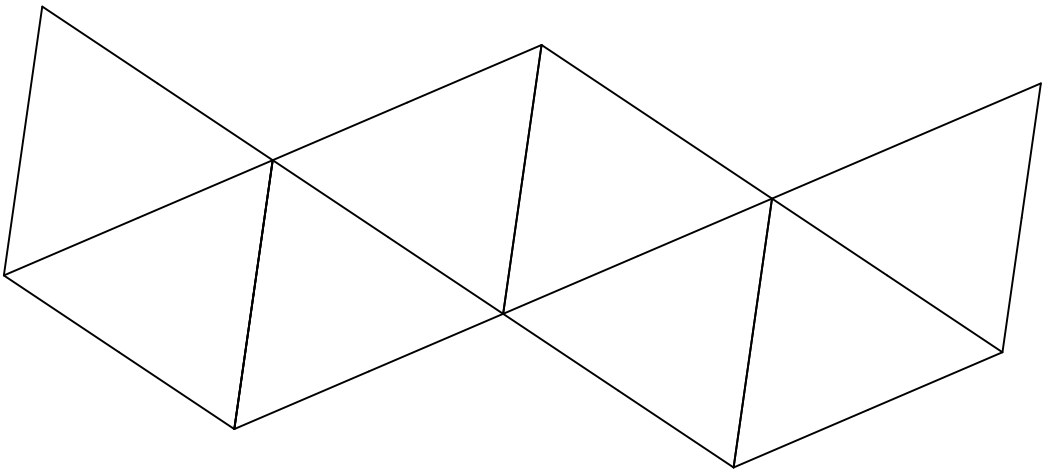
Als "Quaternary Triangular Mesh" wird eine Unterteilung in gleichseitige Dreiecke ausgehend von einem Tetraeder, Oktaeder oder Isokaeder bezeichnet. QTMs können verwendet werden, um Positionen von Punkten auf einer Sphäre (wie z.B. auf der Erde) beliebig genau angeben zu können. Dies wird durch Verwendung von Quadtrees erreicht, die sich bei der Zerlegung der Dreiecke ergeben:



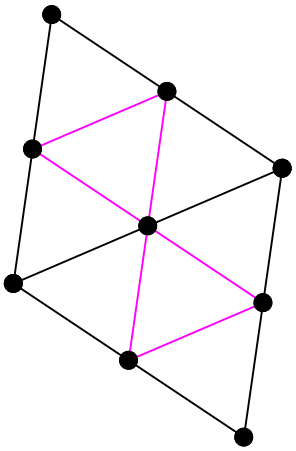
Die Position des Punktes im großen Dreieck kann durch eine immer feinere Zerlegung beliebig genau angenähert werden (hier: 2. Iteration der Zerlegung). Da jedes Dreieck in vier Teildreiecke zerfällt, kann ein Quadtree zur Positionsbestimmung aufgespannt werden.

(Anm.: Quadtrees sind Bäume, bei denen jeder Knoten vier Folgeknoten besitzt)

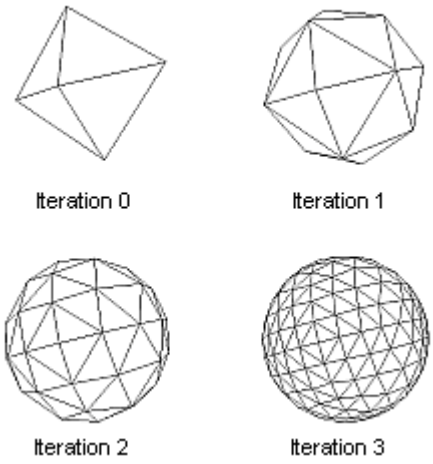
Speziell für die Planetoiden ist der Octahedral Quaternary Triangular Mesh interessant, dessen Grundform der Oktaeder ist. Aufgeklappt würde dieser wie folgt aussehen:



Entnimmt man nun ein Viertel des Oktaeders und unterteilt jeweils beide Dreiecke in weitere vier gleichseitige Dreiecke entsteht folgendes Bild:



Wie bei dieser ersten Iteration der Zerlegung ersichtlich ist, kann der OQTM durch vier Gitter repräsentiert werden, die jeweils eine Kantenlänge von 2^n+1 Knoten bei n Iterationen besitzen.
Diese Eigenschaft kann in Verbindung mit dem Diamond Square Algorithmus ausgenutzt werden.

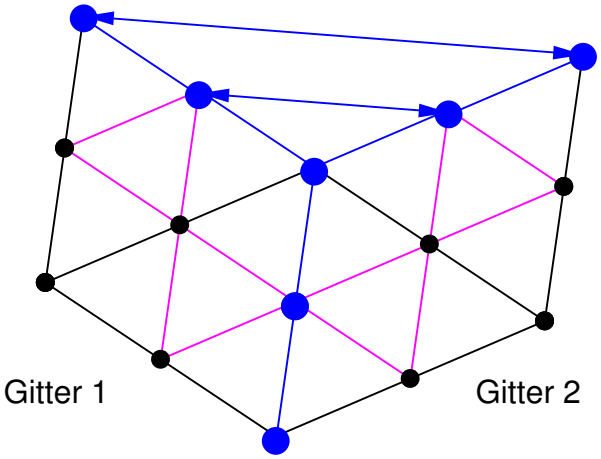


Verschiedene Iterationsstufen eines OQTM

3.1.3 Berechnung des Planetoiden

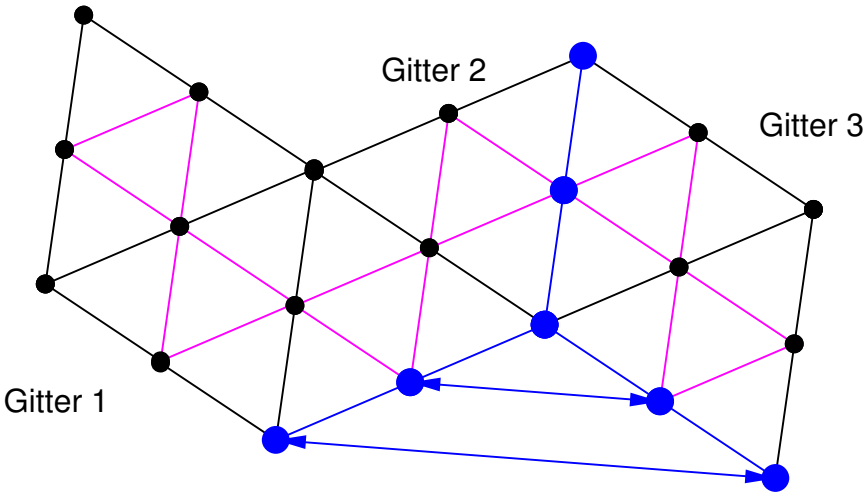
Der Planetoid selber kann nun dadurch erzeugt werden, indem die Höhenangaben des Diamond Square Terrains als Abstände zum Mittelpunkt des OQTMs verwendet werden. Dazu müssen vier Gitter erzeugt und deren Ränder ggf. mit den Werten des angrenzenden Gitters initialisiert werden, um eine stetige Oberfläche zu erreichen. Eine entsprechende Modifikation des Diamond Square Algorithmus ist hierfür notwendig.

Das erste Gitter wird ohne Einschränkungen mit Diamond Square generiert. Das zweite Gitter muss die Ränder, die es mit dem ersten gemein hat, durch die Werte des ersten Gitters initialisieren:

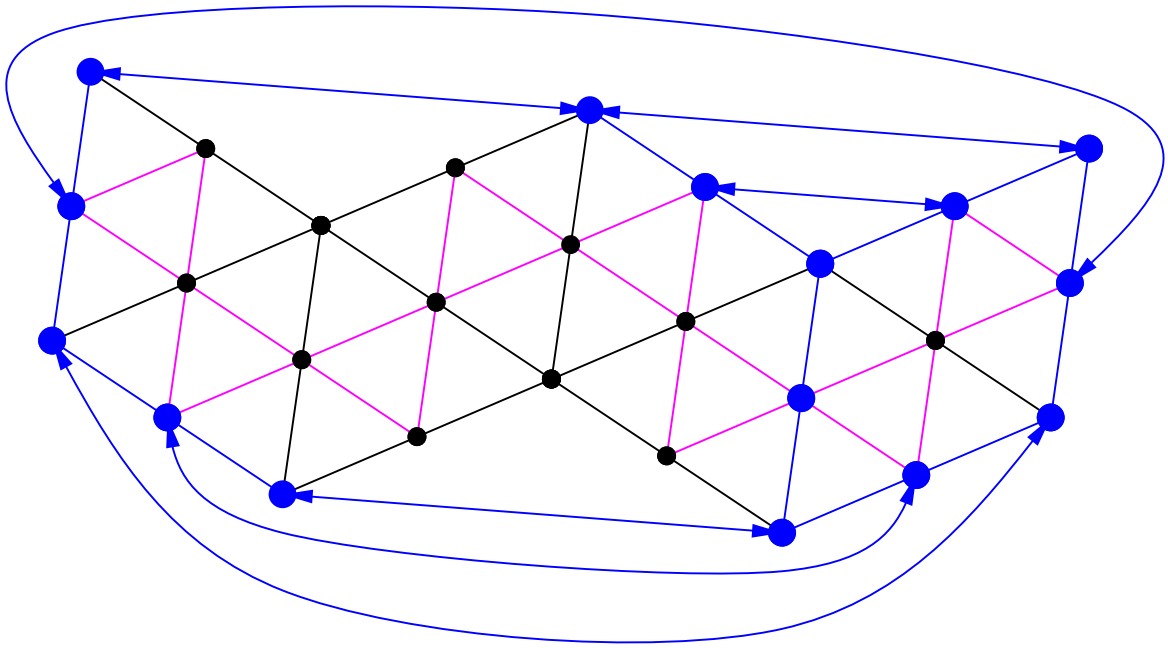


"↔" : Gitterpunkte besitzen dieselben Werte

Das dritte Gitter muss die entsprechenden Werte des zweiten übernehmen:



Das letzte Gitter muss mit den entsprechenden Werten des ersten und dritten Gitters initialisiert werden:

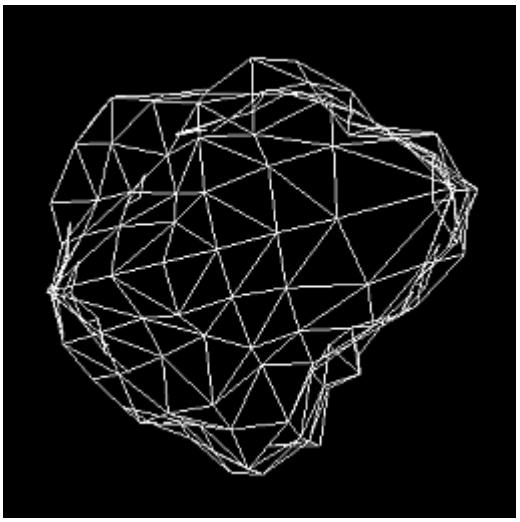


Als nächstes müssen die Koordinaten der Vertices des OQTMs bestimmt werden. Ausgehend von einem Oktaeder, dessen Eckpunkte auf der Einheitskugel liegen, wird jede seiner acht Seiten je nach gewünschter Iteration in Dreiecke zerlegt und die dadurch neu gewonnenen Vektoren der Punkte normalisiert. Führt man nun eine Skalarmultiplikation der Vektoren des OQTM mit den Höhenangaben in den Diamond Square Gittern durch, so erhält man die Vertices des gewünschten Planetoiden. Die Anzahl der darzustellenden Dreiecke berechnet sich wie folgt:

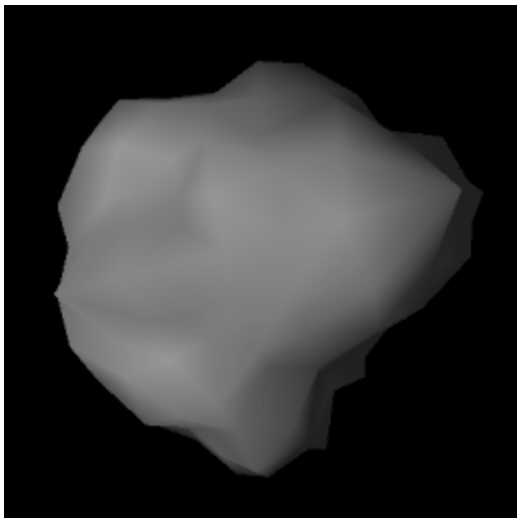
$N_{\text{Polygonzahl pro Gitter}} = 2^{2n+1}$;

$N_{\text{Polygonzahl gesamt}} = 2^{2n+1} * 4 = 2^{2n+3}$;

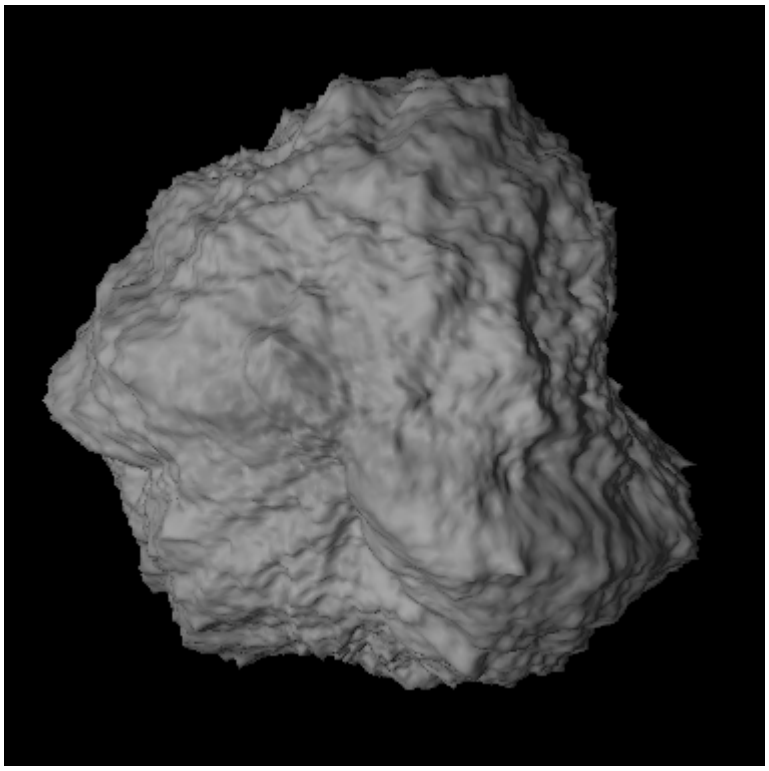
n: Iterationstiefe



Planetoid Wireframe in 3. Iteration



Derselbe Planetoid beleuchtet, ohne Textur



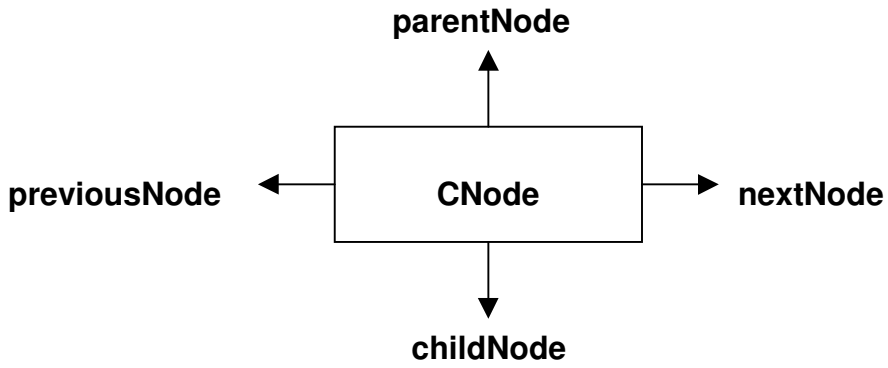
Planetoid in 7. Iteration mit 131072 Polygonen; für ein Spiel momentan utopisch

3.2 Datenmanagement der Spielwelt

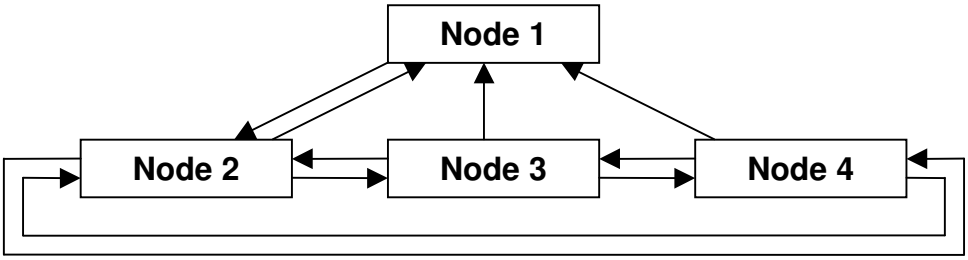
Die Datenverwaltung einer geladenen Spielwelt (Level) wurde aus der in "OpenGL Game Programming" dargestellten SimpEngine übernommen. Alle Objekte des Levels werden in einem Baum aus zyklisch verketteten Listen gespeichert. Die Klasse CNode repräsentiert einen Knoten dieser Baumstruktur und die davon abgeleitete Klasse CObject stellt die kleinste mögliche Entität der Spielwelt dar.

3.2.1 Baumstruktur mit CNode

Ein einzelnes CNode-Objekt besitzt Zeiger auf den vorhergehenden und nächsten Knoten in der Liste sowie ggf. auf einen Parent- und Child-Knoten:



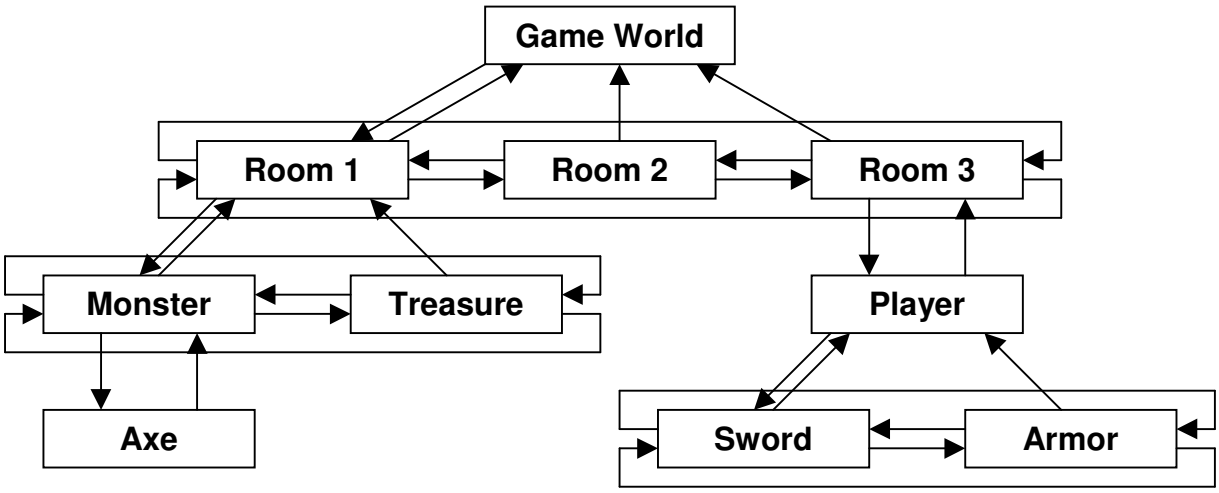
Jeder Parent kann zwar mehrere Kinder, aber nur einen Verweis auf ein Child haben. Der Child ist der Kopf der verketteten Liste der Kinder. Eine Verknüpfung mehrerer CNode-Objekte kann folgendermaßen aussehen:



Node 1 ist hierbei der Parent, Node 2 der Child, und Node 3 und 4 werden "Siblings" von Node 2 genannt.

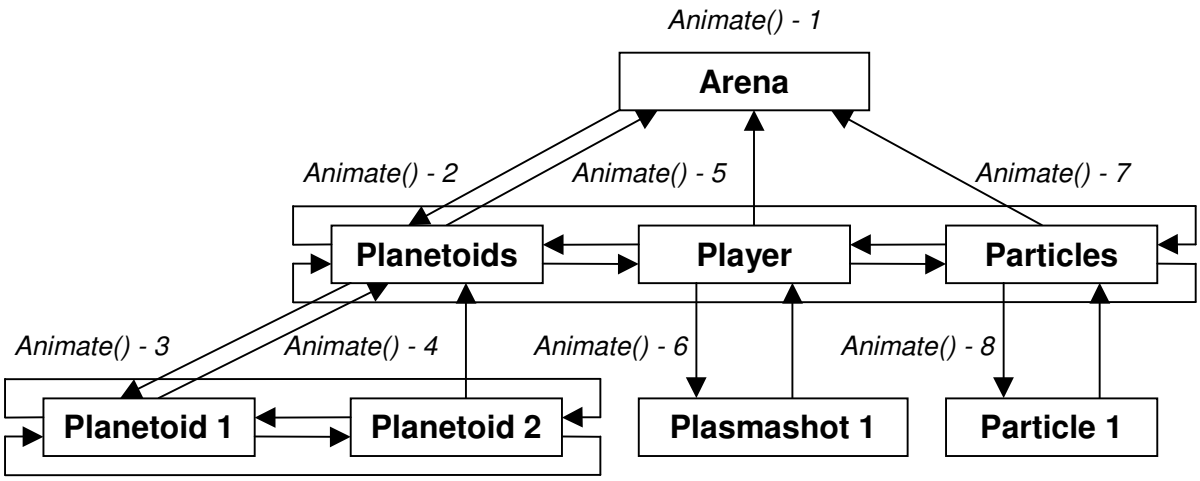
3.2.2 Datenmanagement mit CObject

CObject kann alles in der Spielwelt repräsentieren, das gezeichnet oder animiert wird. Die Klasse ist von CNode abgeleitet, wodurch eine Objekt-Hierarchie aufgebaut werden kann, die die Datenverwaltung der Game-Engine vereinfacht.
Die folgende Abbildung demonstriert eine mögliche Hierarchie in einer fiktiven Spielwelt:



Die Wurzel bildet hierbei "Game World". Diese Spielwelt besteht aus drei Räumen, welche verschiedene Objekte wie Gegner, Spieler usw. beinhalten können. An jedes Objekt können weitere angefügt werden, wie z.B. hier Schwert und Rüstung für den Spieler.
Eine wichtige Eigenschaft von CObject sind die virtuell deklarierten Funktionen für die Vorbereitung (*OnPrepare()*), Animation (*OnAnimate()*), Darstellung (*OnDraw()*) und Kollision (*OnCollide()*) von Objekten. Diese werden durch die in CObject definierten Funktionen *Prepare()*, *Animate()*, *Draw()* und *ProcessCollisions()* aufgerufen. *Animate()* wird für physikalische Berechnungen der Objekte verwendet, *Prepare()* für KI und Statusänderungen, *ProcessCollisions()* für die Kollisionsabfrage und *Draw()* für die graphische Darstellung. Jede dieser Funktionen führt ihre Aktion für das Objekt selber, sowie für Child und Siblings aus. Eigene Objekte müssen nun von CObject abgeleitet werden und die virtuellen Funktionen je nach Bedarf definieren. Durch diese Vorgehensweise können z.B. infolge einen Aufrufs von *Animate()* an das Wurzelobjekt alle Knoten eines Baums bzw. Teilbaums animiert werden.

Das folgende Diagramm soll den Ablauf eines Animationsaufrufs in Bezug auf die Baumstruktur verdeutlichen. Es spiegelt eine mögliche Objekt-Hierarchie in ArP wieder:



Dieser Aufbau ist außerdem wichtig für den Rendervorgang. Alle Partikel sind transparent und müssen für die verwendete Blending-Technik als letztes gezeichnet werden, weswegen "Particles" das letzte Kind in der Liste von "Arena" ist.
In einer Objekt-Hierarchie werden für jedes Child die relativen Koordinaten zum Parent angegeben, wodurch komplexere Strukturen aus mehreren Objekten möglich sind, ohne sämtliche Koordinaten global spezifizieren zu müssen.

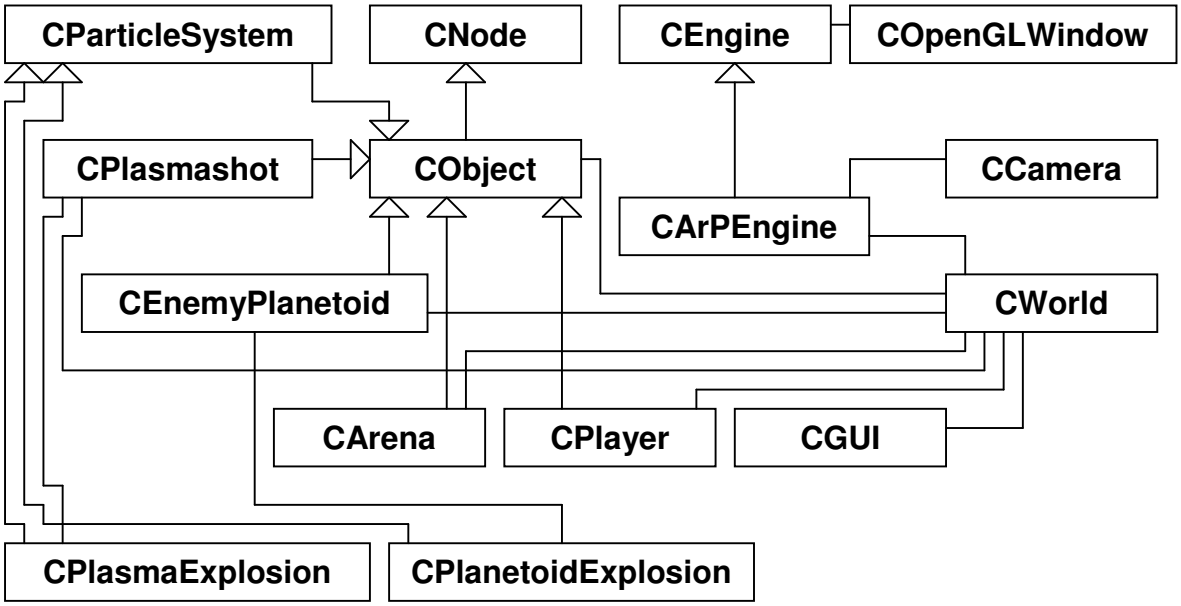
Der CObject-Klasse wurde für die Verwendung in ArP zwei Datenmember hinzugefügt, zum einen eine Matrix zur Angabe der Rotation (OpenGL-Format), zum anderen eine ID, um die Klasse der beteiligten Objekte in der Kollisionsbehandlung zu identifizieren.

3.3 Klassenhierarchie

Die Klassenhierarchie von ArP basiert auf dem Ansatz der SimpEngine in "OpenGL Game Programming". Da statt DirectX und dem Windows Management von Microsoft die SDL verwendet wurde, musste dieser Ansatz entsprechend modifiziert werden.

3.3.1 Klassendiagramm der Hauptklassen

Zur besseren Übersicht wurden die Klassen in Haupt- und Hilfsklassen unterteilt. Die Hauptklassen bestimmen die eigentliche Struktur der ArP Engine. Folgendes Klassendiagramm soll die Zusammenhänge verdeutlichen:



Im Nachfolgenden werden die einzelnen Klassen näher erläutert.

3.3.2 Hauptklassen

3.3.2.1 CEngine

Die CEngine Klasse bildet die Basis der Game-Engine, sie initialisiert die SDL und SDL_mixer Libraries. Dem Konstruktor von CEngine muss ein String für eine Konfigurationsdatei übergeben werden, aus der die Video- und Soundeinstellungen eingelesen werden können. Über COpenGLWindow wird ein Fenster für das Rendering bereitgestellt. Die Funktion *EnterMessageLoop()* bildet die Nachrichtenschleife, welche anstehende Messages einliest und in der Game-Engine mit Hilfe virtueller Funktionen, wie z.B. *OnMouseMove()* oder *OnKeyDown()*, verteilt. Während der Abarbeitung der Nachrichtenschleife wird auch die virtuelle Funktion *GameCycle()* ausgeführt, die einen neuen Spielzyklus startet.
Der Ablauf des in CEngine definierten Spielzyklus sieht wie folgt aus:

```
OnPrepare ();
world->Prepare ();
camera->Animate ();
world->Animate (deltaTime);
world->Draw (camera);
```

D.h. zuerst wird die Engine für den folgenden Game Cycle vorbereitet, anschließend die aktuelle Spielwelt. Hierbei erfolgt die Kollisionsabfrage und es wird überprüft, ob das Level beendet wurde. Darauffolgend werden Kamera und die Spielwelt animiert und durch einen Aufruf von *Draw()* des CWorld-Objektes die Szene vom Blickwinkel der Kamera aus gezeichnet.
Der angegebene Spielzyklus kann ggf. in einer abgeleiteten Klasse neu definiert werden.

3.3.2.2 CArPEngine

CArPEngine bildet die Hauptschnittstelle zu ArP. Die Klasse ist von CEngine abgeleitet und definiert die zur Nachrichtenbearbeitung benötigten virtuellen Funktionen. In diesen wird die Verhaltensweise von ArP auf Benutzereingaben bestimmt, wie z.B. die Steuerung des Raumschiffs. Aus der Hauptkonfigurationsdatei wird der Name für die Levellist sowie benutzerdefinierte Key-Bindings eingelesen. Die Levellist bestimmt die Reihenfolge der zu erstellenden CWorld-Objekte, welche als Übergabeparameter einen String für die Konfigurationsdatei des entsprechenden Levels benötigen. Das erste CWorld-Objekt ist dabei immer eine spezielle Spielwelt, die den Hintergrund für den Startbildschirm bildet.

3.3.2.3 COpenGLWindow

Diese Klasse repräsentiert ein Fenster, auf das mit OpenGL gezeichnet werden kann. Sie initialisiert die SDL-Library und sorgt dafür, dass der Mauszeiger deaktiviert und die Benutzereingabe bei aktivem Fenster auf dieses umgeleitet wird.

3.3.2.4 CWorld

CWorld stellt ein Level in ArP dar. Es wird ein Objektbaum aus Spieler, Arena, Planetoiden usw. aufgebaut und verwaltet. Alle für die Darstellung und Logik der Spielwelt notwendigen Daten werden vorher aus der jeweiligen Level-Konfigurationsdatei ausgelesen. Diese Daten werden in Datenmembern abgelegt, welche allen Objekten der Spielwelt zur Abfrage bereitstehen. Eine besondere Form von CWorld ist die Startwelt, die eine spezielle Konfigurationsdatei benötigt und als Hintergrund für das Hauptmenü dient.

3.3.2.5 CCamera

CCamera bildet die Kamera, durch deren Blickwinkel die Szene gerendert wird. Ein CCamera-Objekt kann an ein beliebiges CObject gebunden werden und übernimmt dessen Position und Rotation.

3.3.2.6 CArena

Diese von CObject abgeleitete Klasse repräsentiert das Spielfeld. Das CArena-Objekt bildet die Wurzel des CObject-Baumes. Es werden die Ebenen für die ODE definiert und das Aussehen des Würfels, in dem sich Spieler und Planetoiden befinden. Ausserdem wird die Skybox gezeichnet, die mit beliebigen Texturen versehen werden kann und den Weltraum simulieren soll, d.h. es ist ein Würfel, der sich mit der Kamera bewegt und immer hinter allen Objekten gezeichnet wird.
Die Positionierung der OpenGL-Lichtquellen findet ebenfalls in CArena statt.

3.3.2.7 CGUI

Durch CGUI wird die graphische Benutzeroberfläche gezeichnet, also HUD (Head Up Display), Texte für den Startbildschirm usw. Zu diesem Zweck wird eine 256x256 Pixel große Textur eingelesen, welche die entsprechenden ASCII-Zeichen beinhaltet. Die Textur wird in 256 16x16 Pixel große Teilbereiche zerlegt, die jeweils ein Zeichen bestimmen. Diese Fragmente dienen nun als Grundlage für OpenGL Display Lists, welche ein bestimmtes Zeichen auf dem Bildschirm rendern, indem sie ein Quadrat mit dem Zeichen als Textur darstellen. Dadurch kann nun jeder beliebige String auf dem Bildschirm visualisiert werden.

3.3.2.8 CPlayer

Das Raumschiff des Spielers wird durch CPlayer dargestellt. Die Klasse ist von CObject abgeleitet und wird in der ODE durch eine Kugel sowohl in der Simulation, als auch in der Kollisionsabfrage angenähert. CPlayer reagiert auf Benutzereingaben durch Addition von Kräften zum ODE-Objekt. Kollidiert der Spieler mit einem Planetoiden, so verlieren beide einen bestimmten Betrag ihrer Startenergie. Sinkt diese auf 0, ist der Spieler zerstört und das Spiel verloren.

3.3.2.9 CEnemyPlanetoid

CEnemyPlanetoid verkörpert die Planetoiden, die vom Spieler abgeschossen werden müssen, um das Level zu gewinnen. Sie werden ebenso wie der Player von CObject abgeleitet und als Kugel in der ODE gehandhabt. Es gibt drei Arten von Planetoiden, nämlich klein, mittel und groß. Asteroiden, welche ihre Startenergie verloren haben, spalten sich in zwei neue der nächst kleineren Art auf, bis sie letztendlich zerstört werden können. Die vier Teile, die für die Darstellung des Planetoiden nötig sind (siehe 3.1.3), werden aus Performancegründen in einer Display List mit Triangle Strip gezeichnet und gespeichert.

3.3.2.10 CPlasmaShot

Drückt der Spieler die Feuertaste, so wird ein CPlasmaShot-Objekt erzeugt und in die Objekt-Hierarchie eingefügt. Diese Klasse ist ebenfalls von CObject abgeleitet und als Sphäre in der ODE präsent. Bei einem Zusammenstoß mit einem Planetoiden wird von beiden ein Betrag der Energie abgezogen und der Plasmaschuß entfernt, wenn seine Energie 0 erreicht. Zur Darstellung wird ein texturiertes, per Blending gezeichnetes Quadrat verwendet, das wie ein Partikel zur Kamera ausgerichtet wird. Dieser Effekt wird "Billboarding" genannt.

3.3.2.11 CParticleSystem

Die Klasse CParticleSystem bildet das Framework für verschiedene Partikeleffekte und ist von CObject abgeleitet. Ein Partikelsystem besteht aus mehreren Partikeln, die emittiert werden. Das Verhalten und die Darstellung muss in einer davon abgeleiteten Klasse definiert werden. Jeder Partikel besitzt verschiedene Eigenschaften wie Farbe, Lebensdauer, Position etc. Das Partikelsystem und die Partikeleffekte wurden größtenteils aus "OpenGL Game Programming" übernommen.

3.3.2.12 CPlanetoidExplosion und CPlasmaExplosion

Beide Klassen sind von CParticleSystem abgeleitet und bestimmen das Aussehen der Explosionen von Plasmaschüssen und Planetoiden. Die einzelnen Partikel werden ähnlich dargestellt wie die Plasmaschüsse, also durch Billboarding und Blending. CPlanetoidExplosion verwendet einen als Parameter übergebenen Geschwindigkeitsvektor, um die zufällige Initialgeschwindigkeit emittierter Partikel zu beeinflussen. Wird hierfür die Geschwindigkeit des explodierenden Asteroiden angegeben, entsteht ein Effekt, der simuliert, die Partikel würden in Richtung des Asteroiden weiterfliegen.
In CPlasmaExplosion werden den einzelnen Partikeln Zufallsgeschwindigkeiten radial von der Mitte der Explosion ausgehend zugewiesen.

3.3.3 Hilfsklassen

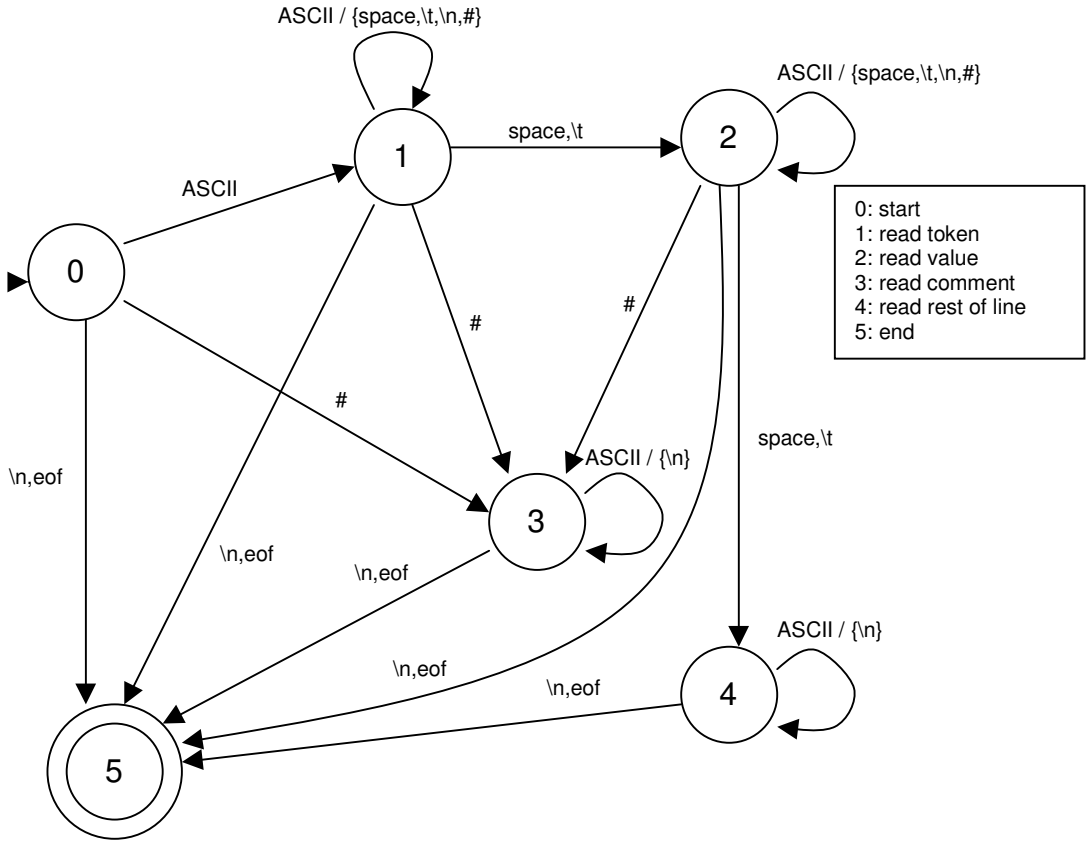
3.3.3.1 CVector

In dieser Klasse sind Funktionen und überladene Operatoren für Vektorarithmetik definiert. Sie wurde vollständig aus "OpenGL Game Programming" übernommen.

3.3.3.2 CConfigFile

CConfigFile wird verwendet um die Konfigurationsdateien einzulesen. Folgender Automat wurde implementiert um die Dateien zu parsen:

K = {0, 1, 2, 3, 4, 5};
Σ = {alle ASCII-Zeichen, eof};
s = 0;
F = {5};
Auf eine Auflistung der Übergangsfunktion δ wird verzichtet.



Eine Zeile einer Konfigurationsdatei muss also aus einem Token und einem Value, getrennt durch ein Leer- oder Tabulatorzeichen bestehen. Mit '#' werden Kommentare markiert, jedoch wird anschließend der Rest der Zeile ignoriert.

3.3.3.3 CDiamondSquare

Diese Klasse verkörpert ein Diamond Square Gitter (siehe 3.1.1). Das Gitter wird mit den Angaben von Iteration, Initialisation, Roughness, Modus und Größe erzeugt. Der Modus gibt dabei an, welche Ränder des Gitters nicht berechnet werden sollen. Dies ist vor allem für die Erstellung des Planetoiden durch einen OQTM wichtig (siehe 3.1.3). Die Größe des Gitters ist notwendig, wenn Normalenvektoren berechnet werden sollen.

3.3.3.4 CPlanetoid

CPlanetoid setzt aus den Diamond Square Gittern und einem OQTM den Planetoiden zusammen (siehe 3.1.3). Als Parameter müssen Iteration, Initialisation, Roughness und Radius angegeben werden. Die durch CDiamondSquare erzeugten Höhenangaben für die Vertices werden auf den Radius des Planetoiden skaliert um sicherzustellen, dass die Größe des Asteroiden maximal der des Radius entspricht. Außerdem wird ein Verschiebungsvektor kalkuliert, indem alle Vertices addiert und der Mittelwert gebildet wird. Das Gleiche wird mit den Beträgen der Vektoren vorgenommen, um den Radius der Bounding-Sphere zu bestimmen, welche in der ODE als Repräsentation des Planetoiden verwendet wird. Der Verschiebungsvektor wird für ein versetztes Zeichnen benutzt, um die Sphäre besser an den Planetoiden anzugleichen.

3.3.3.5 CTexture

Mit CTexture können Texturen im TGA-Format eingelesen und verwaltet werden. Die Funktionalität wurde aus der CTexture-Klasse, welche in "OpenGL Game Programming" beschrieben wird, übernommen.

Kapitel 4: Konfiguration der Game-Engine

Als Schnittstelle für die Konfiguration von ArP werden mehrere Textdateien verwendet. In "arp.cfg" befindet sich die Hauptkonfiguration der Engine, "menu.cfg" beinhaltet Einstellungen für das Demo-Level, das im Hintergrund des Startbildschirms abgespielt wird. Die Dateinamen für die Level-List und die einzelnen Levels sind variabel. Jede Zeile einer Konfigurationsdatei muss genau ein Token und einen Value, durch Leerzeichen oder Tabulatoren getrennt, beinhalten. Aus diesem Grund muss darauf geachtet werden, dass bei Pfadangaben und Dateinamen keine Leerzeichen vorkommen. Kommentare werden durch '#' gekennzeichnet, wobei allerdings der Rest der Zeile anschließend ignoriert wird.
Während der Laufzeit von ArP aufgetretene Fehler werden in "error.txt" gespeichert. Das Log-File "log.txt" liefert Informationen über Aktionen der Game-Engine.

4.1 Hauptkonfiguration

4.1.1 Video und Sound

Konfigurationsbeispiel für "arp.cfg":

width	*1024*
height	*768*
bits	*32*
fullscreen	*true*
music_max_volume	*1.0*
sound_max_volume	*1.0*
sound_frequency	*44100*
sound_stereo	*true*
sound_channels	*16*
sound_buffer	*4096*

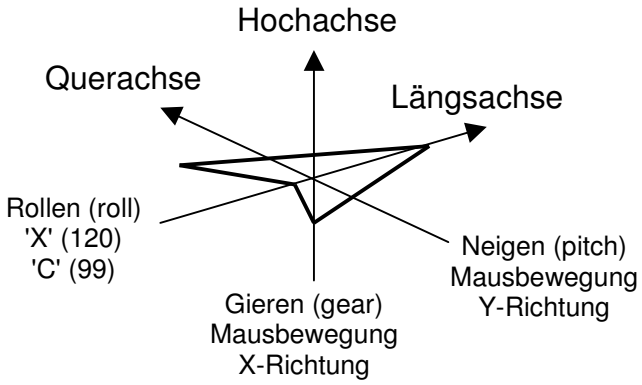
Durch diese Angaben wird ArP im Vollbildmodus mit einer Auflösung von 1024x768 Pixeln und 32 Bit Farbtiefe gestartet. Das Maximum für Sound und Musik wird mit 1.0 auf volle Lautstärke gestellt (0 wäre Stille).
Soundwiedergabe erfolgt in CD-Qualität (44,1 kHz Stereo, 16 Bit wird automatisch verwendet). 16 Kanäle werden für die Abmischung allokiert und die Puffergröße für die Wiedergabe beträgt 4 KB. Letzteres sollte an das System angepasst werden, auf dem ArP läuft. "Stottert" die Soundausgabe, so sollte der Puffer erhöht werden, tritt sie zeitversetzt ein, muss der Puffer verringert werden.

4.1.2 Input-Konfiguration

Die Steuerung des Raumschiffs kann auf beliebige Tasten gelegt werden. Die Angabe der belegten Tasten muss entsprechend der Werte in der Typdefinition von SDLKey erfolgen.
Hier ein Beispiel für die Konfiguration:

mouse_sensitivity	*0.5*
invert_mouse	*false*
bind_shoot	*122*
bind_thrust	*32*
bind_brake	*118*
bind_roll_left	*120*
bind_roll_right	*99*
bind_increase_mousesens	*270*
bind_decrease_mousesens	*269*
bind_music	*109*
mouse_button_left	*shoot*
mouse_button_right	*thrust*
mouse_button_middle	*brake*

Für die Maussteuerung wird die Empfindlichkeit auf 0.5 gesetzt und die Invertierung in Y-Richtung deaktiviert. Ersteres kann auch während der Laufzeit eingestellt werden. Die Auswirkung variiert je nach verwendetem Modell der Maus.
In diesem Beispiel wurde für Schuss die SDLKey 122 ausgewählt, was zwar in der SDL der Z-Taste entspricht, auf deutschen Keyboards jedoch 'Y'. Beschleunigung und Abbremsen der Drehbewegung des Schiffes erfolgt mit Space (32) bzw. 'V' (118).
Die Steuerung des Schiffes soll durch diese Darstellung erläutert werden:



Die Mausempfindlichkeit kann während des Spiels mit Keypad-Minus (269) bzw. Keypad-Plus (270) geändert und die Musik mit 'M' (109) an- bzw. abgeschaltet werden.
Im Beispiel wird auf die linke Maustaste "Schuss", auf die rechte "Beschleunigen" und auf die mittlere "Abbremsen" gelegt. Die Rollbewegung des Schiffes kann ebenfalls bei Bedarf auf die Maustasten gelegt werden.

4.1.3 Weitere Einstellungen

Mit 'show_fps' wird die Anzeige der Frames per Second im Spiel aktiviert. Dies ist besonders nützlich für Performance-Messungen.
Der String, welcher als Value für 'crosshair' angegeben wird, stellt das Fadenkreuz des Raumschiffes dar. Er wird zentriert in der Mitte des Bildschirms gezeichnet.
'level_list' gibt den Namen der Level-List-Datei an, die im Folgenden näher erläutert wird.

4.2 Einzelne Spielwelten (Levels)

Ein Spiel besteht aus mehreren Levels, deren Eigenschaften in separaten Konfigurationsdateien verfügbar sein müssen. Die Reihenfolge und Namen der einzelnen Levels werden in der Level-List spezifiziert.

4.2.1 Level-List

Folgendes wäre ein Beispiel für eine Level-List:

level *level1.cfg*
level *level2.cfg*

Nur Values für Token 'level' werden beachtet. Die Reihenfolge in der Liste bestimmt auch die Reihenfolge, in der die Levels im Spiel geladen werden.

4.2.2 Level-Config

In den einzelnen Level-Config-Files werden alle nötigen Daten für eine Spielwelt angegeben, wie Soundeinstellungen, Texturdaten, ODE-Daten der Spiel-Objekte usw. Die Datei 'menu.cfg' ist für ein besonderes Level nötig, das bei Programmstart geladen wird. In diesem Level ist kein Spieler vorhanden und Asteroiden verlieren durch Kollision keine Energie. Jedes fehlende oder mit falschen Werten angegebene Token wird mit Default-Werten initialisiert.
Die Konfiguration eines Levels ist zu umfangreich, um sie hier in allen Details darzustellen. Deswegen sollen die wichtigsten Einstellungen im Folgenden grob umrissen werden.

4.2.2.1 Audiovisuelle Darstellung der Spielwelt

In jedem Level können die zu ladenden Musik-, Sound- und Texturdateien angegeben werden. Sind die Files nicht vorhanden, weil sie nicht eingelesen werden konnten oder im falschen Format vorliegen, so wirkt sich das auf die Lauffähigkeit des Programms nicht aus. Als Formate für Sound werden WAVE, AIFF, RIFF, OGG und VOC unterstützt, für Musik WAVE, MOD, MIDI und OGG. Texturen müssen als TGA in einer Auflösung von $2^n * 2^n$ ($n > 0$) Pixeln vorliegen.
Für Sound und Musik können pro Level separate Einstellungen für die Lautstärke vorgenommen werden, deren Maximum sich jedoch an den in der Hauptkonfiguration spezifizierten Werten orientiert.

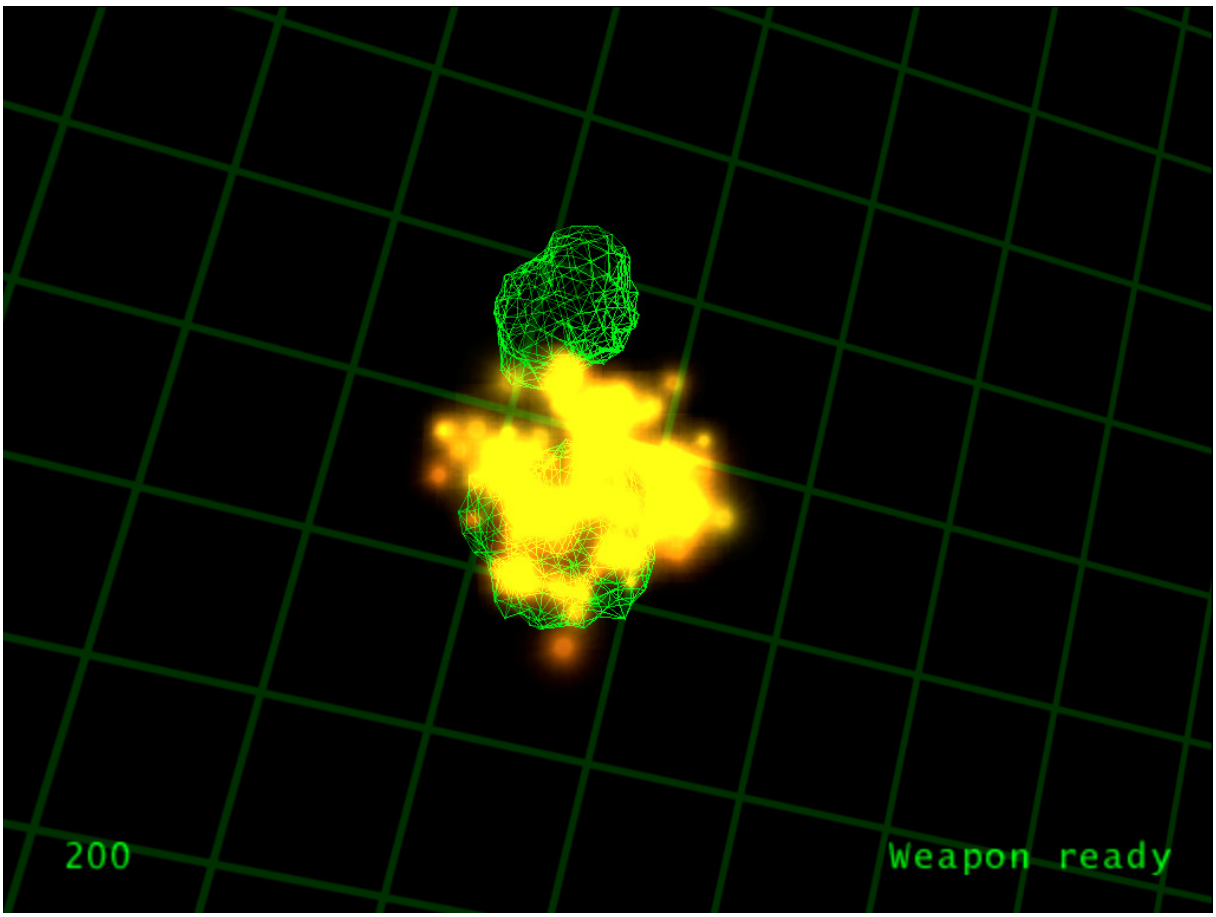
Bis zu drei direktionale OpenGL-Lichtquellen können aktiviert werden. Positionsvektor und Farbe kann für jede Lichtquelle einzeln gewählt werden, letzteres auch für das Ambient-Lighting. Sämtliche Farben werden im RGB- bzw. RGBA-Format angegeben, wobei der Alphakanal nur für Objekte mit Transparenz, wie z.B. Partikel, von Nutzen ist.

Jede der drei Klassen von Planetoiden benötigt die Werte Initialisation, Komplexität (entspricht der Iterationstiefe) und Roughness für den Planetoiden-Algorithmus (siehe 3.1). Außerdem kann für die Asteroiden eine Darstellung als Wireframe oder mit Transparenz aktiviert werden. Die entsprechende Textur wird auf jedes Viertel des Planetoiden projeziert. Das Tiling dafür kann ebenfalls vorgegeben werden, falls die Textur in einem Viertel mehrfach wiederholt werden soll. Neben der Farbe für Asteroiden ist auch die Angabe für ein Farbmodell und ein Alphawert für die Explosionen möglich. Das Farbmodell besteht aus einer Primär- und einer Sekundärfarbe, wodurch $2^3 = 8$ Modelle zur Auswahl stehen.

Die folgenden Einstellungen würden hellgrüne Planetoiden mit transparenter Wireframe Darstellung erzeugen. Als Farbmodell für Explosionen wurde Rot/Grün gewählt:

planetoid_transparency	*true*
planetoid_wireframe	*true*
planetoid_tex_tiling	*0*
planetoid_color_r	*0.0*
planetoid_color_g	*1.0*
planetoid_color_b	*0.0*
planetoid_color_a	*1.0*
planetoid_expl_colors	*rg*
planetoid_expl_alpha	*1.0*

Das Ergebnis würde beispielsweise so aussehen:



Für die Font-Textur zur Darstellung der GUI, den Plasmaschuss sowie die Arena können ebenfalls RGBA-Werte in die Konfigurationsdatei eingetragen werden. Wird eine Skybox verwendet, so ist es sinnvoll, für die Arena Transparenz zu aktivieren. Als Farbe der Plasmaexplosionen wird die Farbe des Plasmaschusses verwendet.

4.2.2.2 ODE Einstellungen

Die Einstellungen für die ODE können anhand eines Beispiels erläutert werden:

plasma_force	*1000.0*
plasma_mass	*0.05*
player_thrust	*0.5*
player_mass	*0.025*
planetoid_1_mass	*0.05*
planetoid_1_max_force	*100.0*
planetoid_1_max_torque	*12.5*

Die Einheiten der einzelnen Werte sind vernachlässigbar, wichtig ist die Relation zueinander. 'plasma_force' und 'plasma_mass' geben die Masse und die Kraft, mit der ein Plasmaschuss aus dem Raumschiff abgegeben wird, an. Für den Spieler sind überdies die Beschleunigung des Raumschiffes, 'player_thrust', sowie die Masse des Raumschiffes, 'player_mass', wichtig. Die Angaben für Planetoiden geben neben der Masse einen Maximumwert für den zufälligen Kraftstoß und das Drehmoment zu Beginn des Levels vor. Jede Klasse (1: klein, 2: mittel, 3: groß) von Asteroiden kann separat konfiguriert werden.
Zwei weitere Parameter können für die ODE spezifiziert werden. Das Token 'bounce' bestimmt das Abprallverhalten, wobei 1 das Maximum ist, d.h. bei Kollisionen prallen die beteiligten Objekte mit voller Geschwindigkeit voneinander ab. Die Gleitreibung wird durch 'coulomb_friction' vorgegeben und kann Werte zwischen 0 (keine Reibung) und 'infinity' (unendliche Reibung) annehmen.

4.2.2.3 Spiellogik

Die Spiellogik bestimmt die Regeln des Levels, d.h. wieviele Asteroiden sind im Level enthalten, welche Energie besitzen diese usw.
Auch hierfür ein kleines Beispiel um die Einstellungsmöglichkeiten zu verdeutlichen:

health_loss	*50*
plasma_health	*50*
plasma_frequency	*1.0*
plasma_lifetime	*10.0*
plasma_arena_bounce	*true*
player_health	*200*
planetoid_1_quantity	*0*
planetoid_1_health	*50*

'health_loss' gibt den Energieverlust für beide Beteiligten in einer Planetoid-Planetoid-, Plasma-Planetoid- oder Player-Planetoid-Kollision vor. Der Plasmaschuss des Spielers besitzt eine Startenergie 'plasma_health' und eine Lebensdauer 'plasma_lifetime', nach deren Ablauf sich der Schuss selbst zerstört. Die Schussfrequenz 'plasma_frequency' bestimmt, wie lange der Spieler warten muss bevor er das nächste Mal schießen kann. Das Token 'plasma_arena_bounce' steuert das Verhalten von Plasma-Arena-Kollisionen. Entweder der Plasmaschuss wird reflektiert oder geschluckt. Erreicht die Energie des Spielers, dessen Startwert durch 'player_health' spezifiziert wird, 0, so ist das Spiel verloren. Jeder der drei Arten Planetoiden kann eine Anzahl und eine Startenergie vorgegeben werden, hier im Beispiel für Planetoiden der Klasse 1 (klein). Die Asteroiden werden zufällig im Level verteilt, allerdings mit einem bestimmten Abstand zum Spieler.

Fazit

Das Ziel der Diplomarbeit, eine Game-Engine für eine 3D-Weltraumsimulation zu erstellen, wurde erreicht. Die Game-Engine ist lauffähig und die 3D-Weltraumsimulation spielbar. Ein komplettes Spiel benötigt allerdings mehr als eine funktionsfähige Game-Engine. Die einzelnen Levels müssen zusammengestellt, der Schwierigkeitsgrad bestimmt, Texturen für Skybox, Planetoiden etc. gezeichnet und Samples für Musik und Soundeffekte kreiert werden.
Im Verlauf der Diplomarbeit wurde Wissen um Entwurf und Implementierung einer Game-Engine sowie im Umgang mit Open-Source Libraries erarbeitet. Die SDL bildet eine sehr gute Oberfläche für die Entwicklung von Multimedia Anwendungen. Sie ist flexibel und vielseitig einsetzbar. Gleiches gilt für die Physik-API ODE, obwohl der eigentlich Einsatzbereich, nämlich die Simulation von Körper, die durch Gelenke verbunden sind, bei der Implementierung der Game-Engine nur angeschnitten wurde.

Schwierigkeiten ergaben sich beim Planetoiden-Algorithmus. Zu Beginn wurde versucht, die Koordinaten einer auf üblichem Weg erstellten Sphäre so zu modifizieren, dass eine gesteinsähnliche Struktur entsteht. Durch die Einteilung in Längen- und Breitengrade besitzt die Kugel an den Polen eine höhere Auflösung und Detailstufe als am Äquator, was für einen Asteroiden nicht geeignet ist. Aus diesem Grund wurden QTMs für die Annäherung an eine Sphäre ausgewählt. Aus den fünf platonischen Körpern eignete sich der Oktaeder am besten für die Verwendung des Diamond Square Algorithmus, mit dem die zufallsgesteuerte fraktale Oberfläche des Planetoiden gebildet werden sollte. Denkbar wären zwar ebenfalls Tetraeder (2 Gitter), Hexaeder (6 Gitter) oder Isokaeder (10 Gitter), jedoch bildet der Oktaeder ein gutes Mittelmaß der benötigten Diamond Square Gitter und passt in seiner Grundform besser zu einer Sphäre mit Längen- und Breitengraden.

Der Ansatz für die SimpEngine, der in "OpenGL Game Programming" vorgestellt wird, musste für die implementierte Game-Engine modifiziert werden. Die OpenGLWindow-Klasse wurde nicht als Basis der Engine verwendet, da zur Fenstererzeugung auf SDL zurückgegriffen wurde. Außerdem sollte das Fenster nur ein Mittel der Engine zur Visualisierung sein, nicht die Engine selber.

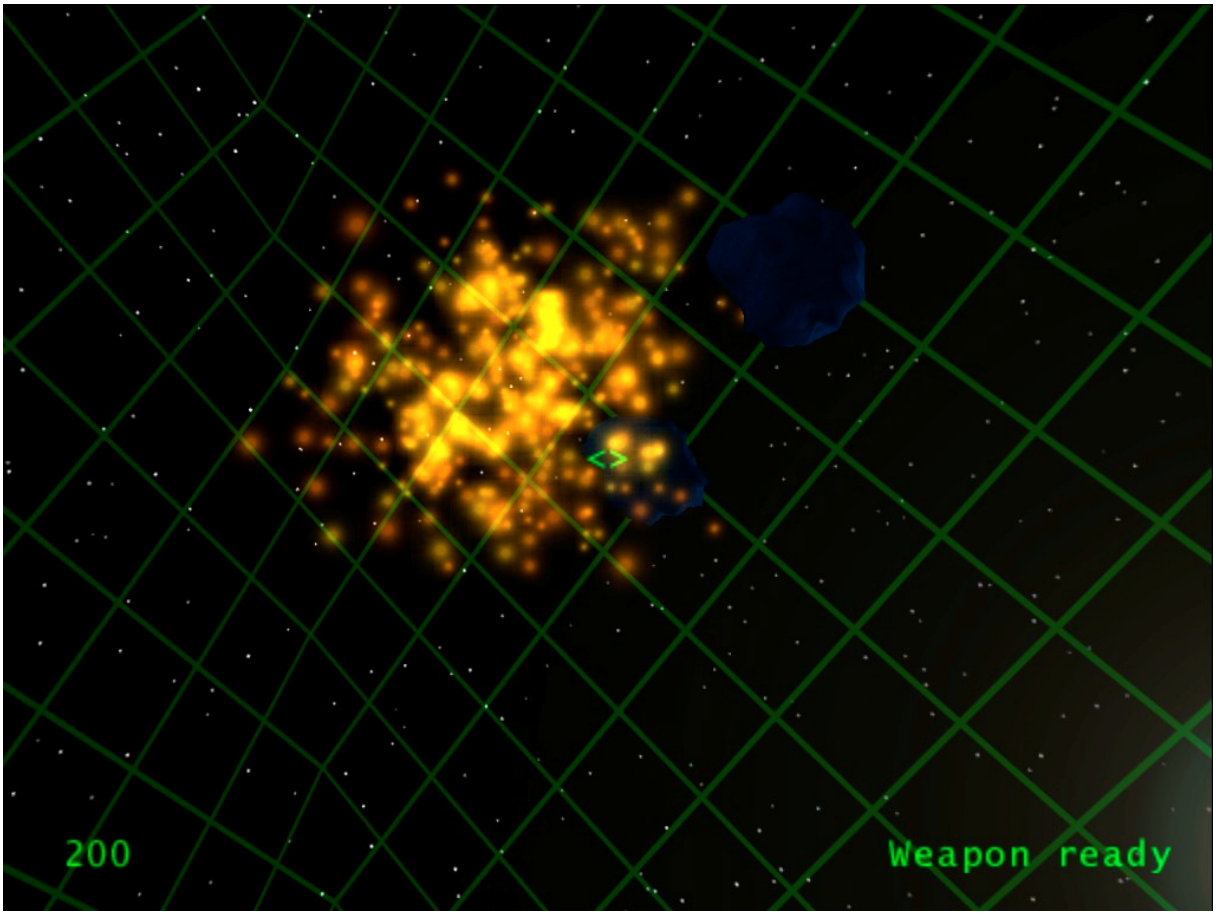
Ebenfalls komplett abgeändert werden musste die Kamera. In der Simpengine wird die Kamera nicht frei im 3D-Raum bewegt, sondern zweidimensional über die Landschaft geschoben. Der Spieler bewegt dabei die Kamera und an dieser befestigt das Player-Objekt. In ArP sollte die Kamera jedoch auf ein beliebiges Objekt plaziert werden können und der Spieler direkt das Player-Objekt steuern können.

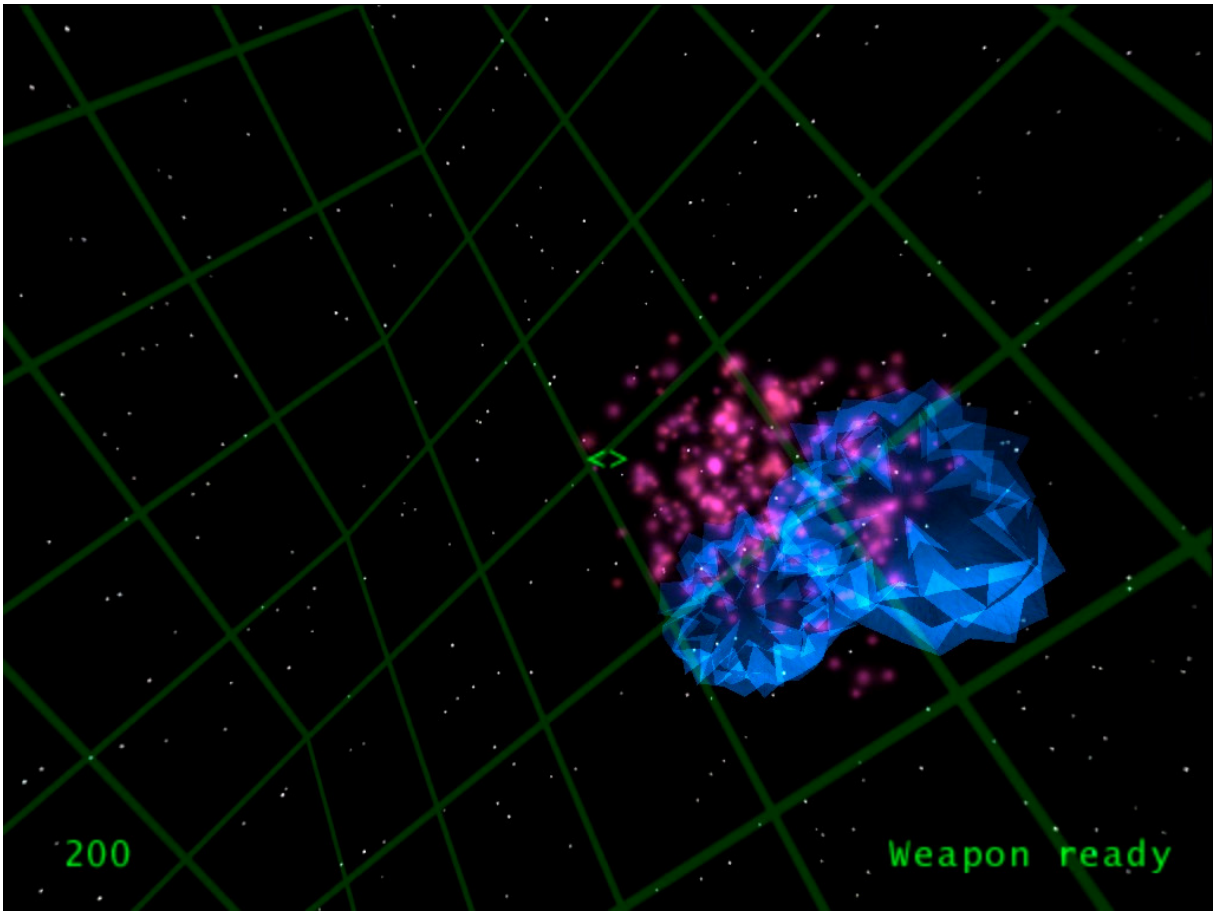
Was beim Arbeiten mit OpenGL Probleme verursachte, war die Tatsache, dass Matrizen an OpenGL in transponierter Form übergeben werden müssen. Die Rendering-Reihenfolge transparenter Objekte muss ebenfalls für die verwendete Blending Technik beachtet werden. Sie werden gezeichnet während der Tiefenpuffer schreibgeschützt ist, um keine Tiefeninformationen zu hinterlassen. Aus diesem Grund muss darauf geachtet werden, zuerst alle undurchsichtigen und anschließend alle transparenten Objekte zu zeichnen.

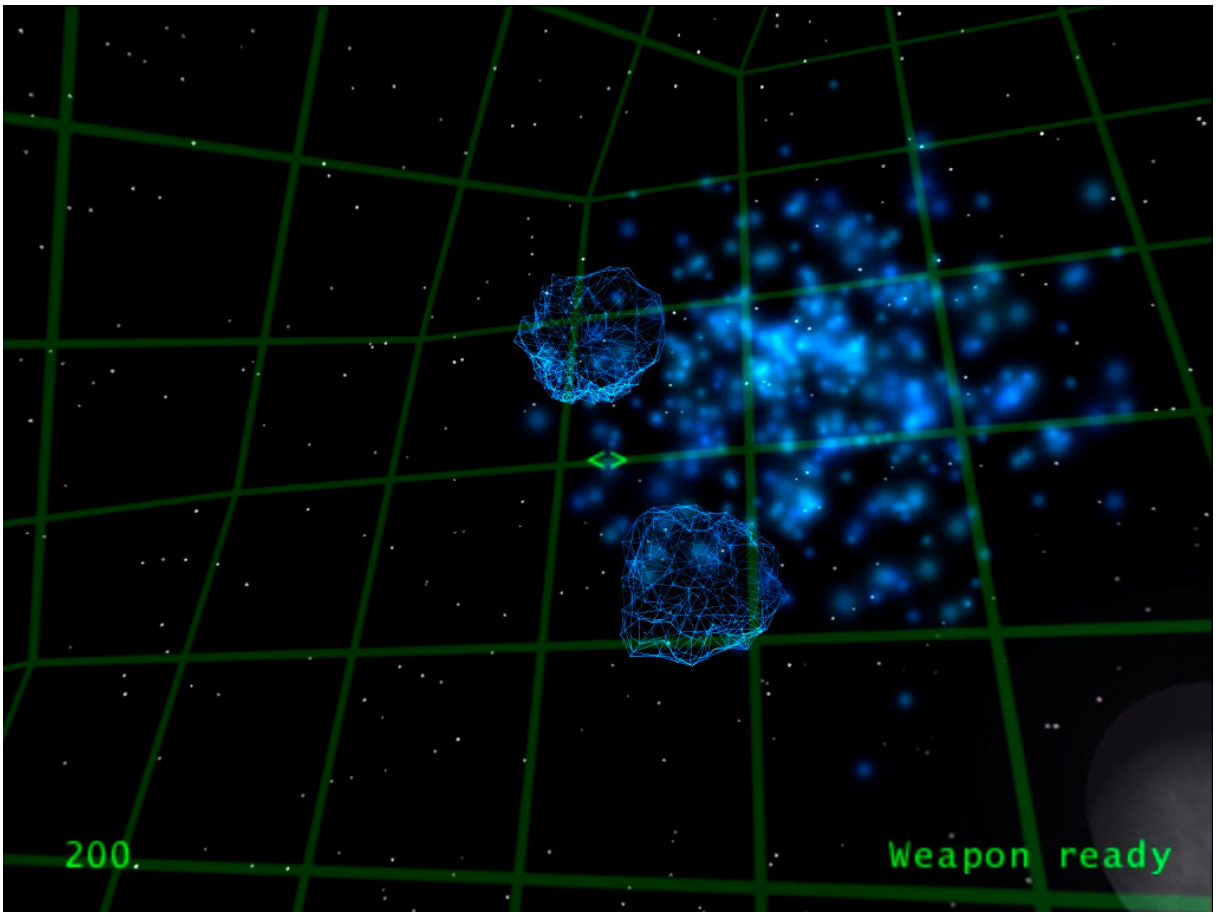
Für die Game-Engine gäbe es vielfältige Verbesserungsmöglichkeiten. Z.B. könnte anstatt der SDL_mixer Bibliothek OpenAL verwendet werden, wodurch echter 3D-Sound möglich wäre. Model-Support wäre sinnvoll um Objekte einzufügen, die durch ein externes Model-Programm erzeugt wurden. Diverse grafische Effekte, wie Multitexturing, Lens Flares, Bump Mapping, Schattenberechnung etc. könnten ebenso zusätzlich implementiert werden.
Genauso vielfältig sind die Erweiterungsmöglichkeiten. Im Original "Asteroids" gab es ein UFO, das den Spieler immer wieder angriff. Etwas ähnliches könnte in ArP eingebaut werden, jedoch wird hierfür eine funktionierende KI benötigt. Interessant wäre auch ein Mehrspielermodus, bei dem mehrere Spieler entweder gegeneinander in Raumschiffen antreten oder kooperativ die Planetoiden zerstören. Für ein Arcade-Spiel wären auch "Power-Ups" üblich, also Bonusgegenstände, die das Schiff aufrüsten, reparieren oder verstärken können. Sollte mit der Engine eine "echte" Weltraumsimulation realisiert werden, so müssen Planeten, Raumstationen, verschiedene Schiffstypen, evtl. ein Handelssystem etc. eingebaut werden.

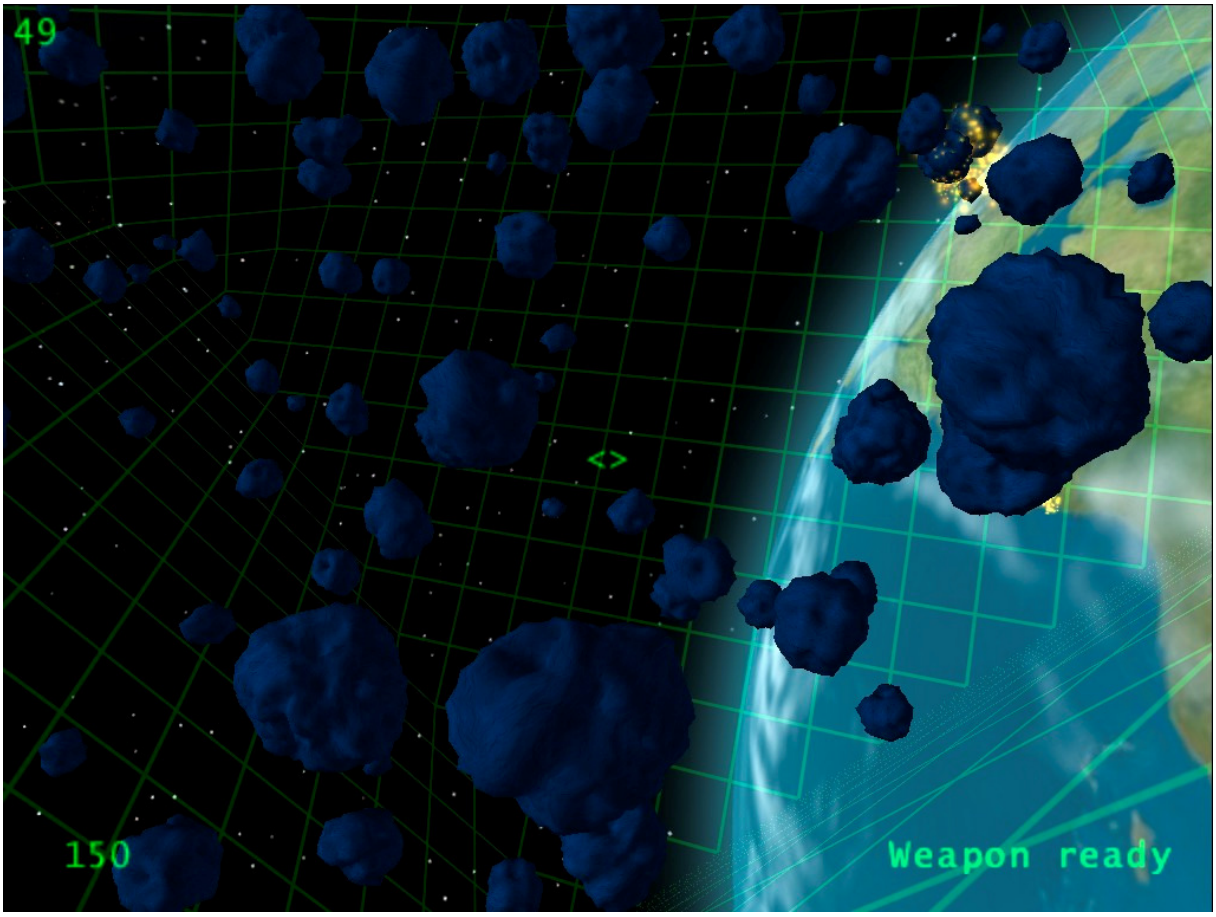
An der Game-Engine kann also selbst nach Abschluss der Diplomarbeit weitergearbeitet werden, wobei sich ein Open-Source Projekt anbieten würde. Außerdem sollte die Engine ausgiebig auch auf anderen Rechnerkonfigurationen getestet und optimiert, sowie auf andere Betriebssysteme portiert werden.

Im Folgenden einige Screenshots zur Demonstration der Game-Engine:









(Skybox-Texturen von DamianTV, Planetoid-Textur von http://infinitefish.com/)

Anhang

A. Performance-Tabelle der Game-Engine

Testsystem:

Prozessor:	Athlon XP 1800+
Hauptspeicher:	512 MB
Grafikkartenprozessor:	nVidia GeForce 3 Ti 200
Grafikkartenspeicher:	64 MB
Betriebssystem:	Windows 2000 Professional Service Pack 4
ArP Fensterauflösung:	1024x768 Pixel, Vollbild

Da die Polygonzahl hauptsächlich für die FPS verantwortlich und ausschließlich von der Komplexität und Anzahl der Planetoiden abhängig ist, wurde mit Klasse 1 (klein) Planetoiden getestet.

Komplexität / Anzahl	**2** (128 Polygone)	**3** (512 Polygone)	**4** (2048 Polygone)	**5** (8192 Polygone)
100	100 FPS	100 FPS	50 FPS	12 FPS
200	100 FPS	100 FPS	33 FPS	10 FPS
300	100 FPS	50 FPS	33 FPS	5 FPS
400	100 FPS	50 FPS	25 FPS	5 FPS
500	100 FPS	50 FPS	12 FPS	3 FPS

Bei hohen FPS kann keine genaue Aussage getroffen werden, da die Granularität des SDL-Timers unter Windows 2000 10 ms beträgt, also jeder Frame mindestens 10 ms (entspricht 100 FPS) andauert. Somit sind z.B. Werte zwischen 100 und 50 nicht möglich.

B. Dokumentation zur verwendeten Funktionalität der Libraries

B.1 SDL

B.1.1 Initialisierung

Bevor andere Funktionen der SDL verwendet werden können, muss diese initialisiert werden. Das geschieht durch SDL_Init:

int SDL_Init (Uint32 flags)

-1 als Rückgabewert signalisiert dabei einen Fehler, 0 eine erfolgreiche Initialisierung. 'flags' muss dabei aus einer oder mehrerer folgender Konstanten bestehen, welche ggf. bitweise oder-verknüpft werden können:

SDL_INIT_TIMER	Timer Subsystem initialisieren
SDL_INIT_AUDIO	Audio Subsystem initialisieren
SDL_INIT_VIDEO	Video Subsystem initialisieren
SDL_INIT_CDROM	CDROM Subsystem initialisieren
SDL_INIT_JOYSTICK	Joystick Subsystem initialisieren
SDL_INIT_EVERYTHING	alles oben genannte initialisieren
SDL_INIT_NOPARACHUTE	keine fatalen Fehler abfangen
SDL_INIT_EVENTTHREAD	wird in der SDL-Dokumentation nicht erläutert

Sollen zu einem späteren Zeitpunkt weitere Subsysteme initialisiert werden, so ist das durch SDL_InitSubSystem möglich:

int SDL_InitSubSystem (Uint32 flags)

Für diese Funktion gilt dasselbe wie für SDL_Init.

SDL_Quit wird verwendet, um alle Subsysteme, die zuvor initialisiert wurden, abzuschalten und gibt alle entsprechenden Ressourcen wieder frei:

void SDL_Quit ()

Um einzelne Subsysteme abzuschalten kann SDL_QuitSubSystem angewandt werden:

void SDL_QuitSubSystem (Uint32 flags)

Mit SDL_WasInit kann verglichen werden, welche Subsysteme initialisiert wurden:

Uint32 SDL_WasInit (Uint32 flags)

Der Rückgabewert ist dabei eine bitweise oder-verknüpfte Kombination der unter SDL_Init aufgeführten Konstanten. 'flags' gibt an, gegen welche Subsysteme geprüft wird.

Intern aufgetretene Fehler der SDL können mit SDL_GetError abgefragt werden:

char SDL_GetError ()*

Es wird ein Zeiger auf einen String zurückgegeben, in dem Informationen über den letzten Fehler gespeichert sind.

B.1.2 Video

Durch SDL_SetVideoMode wird versucht, einen entsprechende Auflösung und Farbtiefe für die Videodarstellung zu wählen.

SDL_Surface SDL_SetVideoMode (int width, int height, int bpp, Uint32 flags)*

Die Parameter 'width', 'height' und 'bpp' geben dabei die Breite und Höhe der gewünschten Display Surface (Framebuffer) an sowie die Farbtiefe in Bits per Pixel. In der vorliegenden Version der SDL werden nur Pixelformate >= 8 Bit unterstützt. 'flags' kann eine beliebige Kombination durch bitweises OR folgender Konstanten sein:

SDL_SWSURFACE	Display Surface im System-Speicher anlegen
SDL_HWSURFACE	Display Surface im Video-Speicher anlegen
SDL_ASYNCBLIT	asynchrone Updates der Display Surface aktivieren
SDL_ANYFORMAT	ist keine Display Surface für das gewünschte Pixelformat verfügbar, so wird bei Angabe dieser Konstante trotzdem anstatt der Shadow Surface die Video Surface verwendet
SDL_HWPALETTE	SDL exklusiven Zugriff auf die Palette erlauben
SDL_DOUBLEBUF	Hardware Double Buffering aktivieren; nur wirksam in Verbindung mit SDL_HWSURFACE
SDL_FULLSCREEN	SDL versucht, den Vollbildmodus zu aktivieren
SDL_OPENGL	es soll ein OpenGL Rendering Context erstellt werden; vorher sollten mit SDL_GL_SetAttribute die OpenGL Video Attribute gesetzt werden
SDL_OPENGLBLIT	dasselbe wie SDL_OPENGL, erlaubt aber zusätzlich Blitting Operationen, wobei die 2D-Oberfläche einen Alphakanal besitzt und mit SDL_UpdateRects aktualisiert werden muss
SDL_RESIZABLE	die Größe des Fensters kann vom Benutzer verändert werden
SDL_NOFRAME	wenn möglich wird ein Fenster ohne Rahmen und Titelleiste erzeugt; im Vollbildmodus immer gesetzt

Gezeichnet wird bei 2D-Oberflächen auf eine verdeckte Display Surface, welche durch Aktualisierung mittels SDL_UpdateRect, SDL_UpdateRects oder SDL_Flip (nur bei Hardware Double Buffering) auf dem Bildschirm dargestellt wird. Da OpenGL für die Game-Engine verwendet wurde, wird hier nicht weiter auf die 2D-Funktionalität der SDL eingegangen.
Die OpenGL Video Attribute können mit SDL_GL_SetAttribute gesetzt werden:

int SDL_GL_SetAttribute (SDL_GLattr attr, int value)

Der Rückgabewert -1 zeigt dabei einen Fehler auf, 0 Erfolg.
'attr' gibt das gewünschte Attribut an, das auf den Wert 'value' gesetzt wird. Folgende Konstanten sind für 'attr' zulässig (Puffergrößen in Bit):

SDL_GL_RED_SIZE	Größe des Framebuffers für Rot
SDL_GL_GREEN_SIZE	Größe des Framebuffers für Grün
SDL_GL_BLUE_SIZE	Größe des Framebuffers für Blau
SDL_GL_ALPHA_SIZE	Größe des Framebuffers für Alpha
SDL_GL_DOUBLEBUFFER	0 oder 1; Double Buffering aus/ein
SDL_GL_BUFFER_SIZE	Größe des Framebuffers
SDL_GL_DEPTH_SIZE	Größe des Depthbuffers
SDL_GL_STENCIL_SIZE	Größe des Stencilbuffers
SDL_GL_ACCUM_RED_SIZE	Größe des Accumulationbuffers für Rot
SDL_GL_ACCUM_GREEN_SIZE	Größe des Accumulationbuffers für Grün
SDL_GL_ACCUM_BLUE_SIZE	Größe des Accumulationbuffers für Blau
SDL_GL_ACCUM_ALPHA_SIZE	Größe des Accumulationbuffers für Alpha

Die Änderungen der OpenGL Attribute werden erst nach SDL_SetVideoMode gültig. Soll danach geprüft werden, ob die Attributänderungen erfolgreich waren, so ist dies mit SDL_GL_GetAttribute möglich:

int SDL_GL_GetAttribute (SDL_GLattr attr, int value)*

Diese Funktion ist ähnlich SDL_GL_SetAttribute, jedoch wird hierbei der Attributwert ausgelesen und in 'value' gespeichert.

Wird OpenGL zur Laufzeit geladen bzw. möchte man bestimmte OpenGL Erweiterungen wie z.B. Multitexturing verwenden, so kann mit SDL_GL_GetProcAddress die Adresse der gewünschten Funktion gefunden werden:

void SDL_GL_GetProcAddress (const char* proc)*

Es wird dabei ein Zeiger auf die OpenGL Funktion 'proc' zurückgeliefert, bzw. NULL wenn diese nicht gefunden wurde.

OpenGL kann zur Laufzeit mit SDL_GL_LoadLibrary geladen werden:

int SDL_GL_LoadLibrary (const char path)*

-1 wird bei einem Fehler zurückgegeben, 0 bei Erfolg. 'path' gibt den Pfad zur OpenGL Library an. Das Laden muss vor dem Aufruf von SDL_SetVideoMode erfolgen, und anschließend muss SDL_GL_GetProcAddress verwendet werden, um über Funktionszeiger auf OpenGL zuzugreifen.

Ist Double Buffering aktiv, so muss anstatt glSwapBuffers SDL_GL_SwapBuffers angewandt werden:

void SDL_GL_SwapBuffers ()

Ebenfalls wichtig bei Verwendung der SDL für OpenGL ist, anstelle der Headerfiles 'gl.h' (OpenGL) und 'glu.h' (OpenGL Utilities) nur 'sdl_opengl.h' zu inkludieren.

Der Mauszeiger kann mit SDL_ShowCursor abgeschaltet werden:

int SDL_ShowCursor (int toggle)

'toggle' kann entweder SDL_ENABLE, SDL_DISABLE oder SDL_QUERY sein. Es wird der aktuelle Status des Mauszeigers zurückgeliefert.

B.1.3 Window Management

Mit SDL_WM_SetCaption kann der Titel des Application-Windows verändert werden:

void SDL_WM_SetCaption (const char title, const char* icon)*

Der Parameter 'title' gibt den Namen des Titels an, 'icon' den Namen des Symbols.

SDL_WM_GetCaption wird verwendet, um die entsprechenden Namen herauszufinden:

*void SDL_WM_GetCaption (char** title, char** icon)*

Es ist auch möglich, ein eigenes Icon für die Applikation mit Hilfe von SDL_WM_SetIcon bereitzustellen:

void SDL_WM_SetIcon (SDL_Surface icon, Uint8* mask)*

'icon' beinhaltet dabei die Informationen für das darzustellende Icon. Mit 'mask' kann die Form des Icons bestimmt werden, d.h. welche Pixel durchsichtig sind und welche nicht.

Für die Game-Engine ist es ebenfalls sinnvoll, alle Input-Messages direkt an die Applikation weiterzuleiten. Dies kann durch SDL_WM_GrabInput erreicht werden:

SDL_GrabMode SDL_WM_GrabInput (SDL_GrabMode mode)

Der aktuelle Grab-Modus wird zurückgeliefert. Für 'mode' können die Konstanten SDL_GRAB_QUERY, SDL_GRAB_ON und SDL_GRAB_OFF verwendet werden.

B.1.4 Timer

Mit SDL_GetTicks kann die seit der Initialisierung der SDL vergangene Zeit in Millisekunden ermittelt werden:

Uint32 SDL_GetTicks ()

Es muss hierbei beachtet werden, dass aufgrund des 32-Bit-Rückgabewertes ein Überlauf nach ca. 49 Tagen Laufzeit auftritt. (2^{32} ms entspricht ca. 49,710 Tagen).

Soll eine bestimmte Zeit gewartet werden, bis das Programm weiter fortfährt, kann SDL_Delay verwendet werden:

void SDL_Delay (Uint32 ms)

Die Applikation wartet mindestens 'ms' Millisekunden. Möglicherweise kann die Wartedauer länger sein, was vom Scheduling des Betriebssystems abhängt.

SDL_AddTimer erstellt einen neuen Timer, der periodisch eine Callback-Funktion aufruft:

SDL_TimerID SDL_AddTimer (Uint32 interval, SDL_NewTimerCallback callback, void param)*

Zurückgegeben wird die ID des neuen Timers oder NULL, falls ein Fehler aufgetreten ist. Die Callback-Funktion 'callback' wird alle 'interval' Millisekunden aufgerufen. Durch den Zeiger 'param' können dieser Funktion Parameter übergeben werden.

Mit SDL_RemoveTimer kann ein Timer wieder gelöscht werden:

SDL_bool SDL_RemoveTimer (SDL_TimerID id)

Der boolean Rückgabewert gibt an, ob der Timer erfolgreich entfernt wurde. Als 'id' muss die ID des zu löschenden Timers angegeben werden.

B.1.5 Event Handling

Dem Event Handling liegt eine Union aus verschiedenen Typen von Events zugrunde:

```
typedef union
{
        Uint8 type;
        SDL_ActiveEvent active;
        SDL_KeyboardEvent key;
        SDL_MouseMotionEvent motion;
        SDL_MouseButtonEvent button;
        SDL_JoyAxisEvent jaxis;
        SDL_JoyBallEvent jball;
        SDL_JoyHatEvent jhat;
        SDL_JoyButtonEvent jbutton;
        SDL_ResizeEvent resize;
        SDL_ExposeEvent expose;
        SDL_QuitEvent quit;
        SDL_UserEvent user;
        SDL_SywWMEvent syswm;
} SDL_Event;
```

Diese Struktur wird verwendet, wenn Ereignisse aus der Queue ausgelesen bzw. neue der Warteschlange hinzugefügt werden.

Die Art des Ereignisses kann durch 'type' bestimmt werden, welche eine der folgenden Konstanten sein kann:

Ereignistyp:	Ereignisstruktur:
SDL_ACTIVEEVENT	SDL_ActiveEvent
SDL_KEYDOWN/UP	SDL_KeyboardEvent
SDL_MOUSEMOTION	SDL_MouseMotionEvent
SDL_MOUSEBUTTONDOWN/UP	SDL_MouseButtonEvent
SDL_JOYAXISMOTION	SDL_JoyAxisEvent
SDL_JOYBALLMOTION	SDL_JoyBallEvent
SDL_JOYHATMOTION	SDL_JoyHatEvent
SDL_JOYBUTTONDOWN/UP	SDL_JoyButtonEvent
SDL_QUIT	SDL_QuitEvent
SDL_SYSWMEVENT	SDL_SysWMEvent
SDL_VIDEORESIZE	SDL_ResizeEvent
SDL_VIDEOEXPOSE	SDL_ExposeEvent
SDL_USEREVENT	SDL_UserEvent

Es sollen hierbei nur die für die Maus- und Tastatureingaben wichtigen Events näher erläutert werden.

SDL_KeyboardEvent wird benutzt, wenn als Typ der Event-Union SDL_KEYDOWN bzw. SDL_KEYUP vorliegt. Die Strukturdefinition sieht wie folgt aus:

```
typedef struct
{
        Uint8 type;
        Uint8 state;
        SDL_keysym keysym;
} SDL_KeyboardEvent;
```

'type' entspricht dem Typ der Event-Union, 'state' enthält die selbe Information wie 'type', verwendet allerdings andere Konstanten, nämlich SDL_PRESSED und SDL_RELEASED. Um zu bestimmen, welche Taste gedrückt oder losgelassen wurde, wird 'keysym' verwendet. Folgendermaßen ist SDL_keysym definiert:

```
typedef struct
{
        Uint8 scancode;
        SDLKey sym;
        SDLMod mod;
        Uint16 unicode;
} SDL_keysym;
```

'scancode' gibt Auskunft über den Hardware-spezifischen Scancode der verwendeten Tastatur. 'sym' beinhaltet einen SDL-definierten Wert für die jeweilige Taste und 'mod' gibt an, ob bestimmte Key-Modifier aktiv sind, wie z.B. Shift, Alt, Ctrl etc. Wurde Unicode-Übersetzung für die SDL aktiviert, so kann der entsprechende Wert aus 'unicode' ausgelesen werden, jedoch sollte die Übersetzung aufgrund des Overheads nur angewandt werden, wenn sie wirklich benötigt wird.

Mausbewegungen werden durch SDL_MouseMotionEvent angezeigt. Die entsprechende Strukturdefinition:

```
typedef struct
{
        Uint8 type;
        Uint8 state;
        Uint16 x, y;
        Sint16 xrel, yrel;
} SDL_MouseMotionEvent;
```

'type' kann hierbei nur SDL_MOUSEMOTION sein. Der aktuelle Status der Maustasten wird in 'state' festgehalten und kann mit dem SDL-Macro SDL_BUTTON interpretiert werden. 'x' und 'y' bzw. 'xrel' und 'yrel' geben dabei die absolute und relative Mauszeigerposition an. Ist Grab-Input aktiv (SDL_WM_GrabInput (SDL_GRAB_ON)) und der Cursor versteckt (SDL_ShowCursor (SDL_DISABLE)), so werden relative Positionen ausgegeben, selbst wenn der Mauszeiger den Rand des Fensters erreicht. Momentan ist dies allerdings nur für Windows und Linux in der SDL implementiert.

Wird eine der Maustasten gedrückt oder losgelassen, so wird ein Event vom Typ SDL_MOUSEBUTTONDOWN bzw. SDL_MOUSEBUTTONUP erzeugt. Die Definition der SDL_MouseButtonEvent-Struktur lautet:

```
typedef struct
{
        Uint8 type;
        Uint8 button;
        Uint8 state;
        Uint16 x, y;
} SDL_MouseButtonEvent;
```

'type' und 'button' geben den Typ des Events sowie die betroffene Taste an, welche entweder SDL_BUTTON_LEFT, SDL_BUTTON_RIGHT oder SDL_BUTTON_MIDDLE sein kann. Wie bei Keyboard-Events gibt hier 'state' das gleiche wie 'type' an, nur mit unterschiedlichen Konstanten (siehe SDL_KeyboardEvent). 'x' und 'y' enthalten die Koordinaten des Mauszeigers zum Zeitpunkt des aufgetretenen Events.

Wichtig ist ebenfalls SDL_QuitEvent. Die Definition dieser Struktur ist trivial:

```
typedef struct
{
        Uint8 type
} SDL_QuitEvent;
```

Diese Struktur dient keinem besonderen Zweck. Vielmehr ist das Event an sich wichtig um festzustellen, ob die Applikation ein Signal zur Terminierung empfangen hat. Wird dieses Ereignis gefiltert oder ignoriert, so ist es für den Benutzer nicht möglich die Applikation zu beenden, außer er beendet direkt den Prozess.

Um nun Ereignisse verarbeiten zu können, müssen mit SDL_PumpEvents alle Geräte auf neu anliegende Nachrichten abgefragt werden:

void SDL_PumpEvents ()

Wird diese Funktion nicht ausgeführt, so bleibt die Event-Queue leer.

Mit SDL_PeepEvents kann nun die Queue auf Events abgefragt werden:

int SDL_PeepEvents (SDL_Event events, int numevents, SDL_eventaction action,*
Uint32 mask)

Tritt ein Fehler auf, wird -1 zurückgegeben, andernfalls die Anzahl Elemente, die in 'events' abgespeichert wurden. Mit 'action' kann das Verhalten der Funktion kontrolliert werden. Wird hier SDL_ADDEVENT angegeben, so können bis zu 'numevents' neue Ereignisse an das Ende der Queue eingefügt werden. Bei Verwendung von SDL_PEEKEVENT werden bis zu 'numevents' Ereignisse gelesen, diese allerdings nicht entfernt, und mit SDL_GETEVENT werden 'numevents' Ereignisse entnommen. Für die beiden letztgenannten Aktionen kann mit 'mask' bestimmt werden, welche Ereignis-Typen eingelesen werden sollen (siehe SDL_Event).

Soll nur eine Message in der Warteschlange abgefragt werden, so ist dies mit SDL_PollEvent möglich:

int SDL_PollEvent (SDL_Event event)*

Zurückgegeben wird 1, wenn Events anliegen, ansonsten 0. Ist 'event' ungleich NULL, so wird das nächste Event aus der Queue entfernt und in dieser Variable gespeichert.

Ähnlich arbeitet die Funktion SDL_WaitEvent:

int SDL_WaitEvent (SDL_Event event)*

Es wird gewartet, bis ein Event auftritt. Eine 0 wird zurückgeliefert, wenn ein Fehler auftritt, ansonsten 1. Für 'event' gilt dasselbe wie bei SDL_PollEvent.

Möchte man ein Ereignis der Queue hinzufügen, kann man statt SDL_PeepEvent auch SDL_PushEvent verwenden:

int SDL_PushEvent (SDL_Event event)*

Ist der Aufruf erfolgreich, wird 0 zurückgegeben, andernfalls -1. Das Ereignis 'event' wird an das Ende der Warteschleife angefügt.

Es gibt noch weitere Funktionen zur Ereignisbehandlung, wie z.B. Filterfunktionen oder Statusfunktionen zum Einlesen aller momentan gedrückten Tasten. Da diese Routinen nicht in der Implementierung verwendet wurden, sollen sie hier nicht weiter erläutert werden.

B.2 SDL_mixer

B.2.1 Initialisierung

Bevor die Funktionen des SDL_mixer zur Verfügung stehen, muss die Mixer-API mit Mix_OpenAudio initialisiert und somit das Gerät zur Soundausgabe geöffnet werden:

int Mix_OpenAudio (int frequency, Uint16 format, int channels, int chunksize)

Der Rückgabewert -1 zeigt dabei einen Fehler an, 0 Erfolg. 'frequency' gibt die gewünschte Wiedergabefrequenz in Hz an.
Das Format der auszugebenden Samples wird mit 'format' bestimmt und kann eine der folgenden Konstanten sein:

AUDIO_U8	Unsigned 8-bit Samples
AUDIO_S8	Signed 8-bit Samples
AUDIO_U16LSB	Unsigned 16-bit Samples, in little-endian byte-Ordnung
AUDIO_S16LSB	Signed 16-bit Samples, in little-endian byte-Ordnung
AUDIO_U16MSB	Unsigned 16-bit Samples, in big-endian byte-Ordnung
AUDIO_S16MSB	Signed 16-bit Samples, in big-endian byte-Ordnung
AUDIO_U16	dasselbe wie AUDIO_U16LSB
AUDIO_S16	dasselbe wie AUDIO_S16LSB
AUDIO_U16SYS	Unsigned 16-bit Samples, in System-byte-Ordnung
AUDIO_S16SYS	Signed 16-bit Samples, in System-byte-Ordnung
MIX_DEFAULT_FORMAT	dasselbe wie AUDIO_S16SYS

'channels' gibt die Anzahl der Kanäle für die Abmischung der Sound-Samples an (Mono: 1, Stereo: 2). Die Größe jedes abgemischten Samples wird durch 'chunksize' angegeben. Ist dieser Wert zu klein, kann es zum "Stottern" des Sounds kommen, wird er zu groß gewählt, wird die zeitliche Versetzung beim Start der Wiedergabe zunehmend wahrnehmbarer.

Durch Mix_CloseAudio wird die Mixer-API geschlossen und verwendete Ressourcen wieder freigegeben:

void Mix_CloseAudio ()

Danach sollen keine weiteren SDL_mixer-Funktionen, außer Mix_OpenAudio, verwendet werden. Wurde das Soundausgabegerät mehrmals geöffnet, so muss genauso oft Mix_CloseAudio aufgerufen werden.

Sind Fehler aufgetreten, so können diese mit Mix_GetError identifiziert werden:

char Mix_GetError ()*

Rückgabewert ist ein Zeiger auf einen nullterminierten String, welcher Informationen über den zuletzt aufgetretenen Fehler enthält.

Um zu überprüfen, ob die mit Mix_OpenAudio angegebenen Werte auch gesetzt wurden, kann Mix_QuerySpec aufgerufen werden:

int Mix_QuerySpec (int frequency, Uint16* format, int* channels)*

Bei Fehlern wird 0 zurückgegeben, ansonsten die Anzahl der Aufrufe von Mix_OpenAudio. Die entsprechenden Daten zur Wiedergabe der Soundsamples werden in 'frequency', 'format' und 'channels' abgelegt.

B.2.2 Sound Wiedergabe

Bevor einzelne Samples wiedergegeben werden können, müssen sie mit Mix_LoadWAV eingelesen werden:

Mix_Chunk Mix_LoadWAV (char* file)*

Zurückgeliefert wird ein Pointer auf das Sample im Speicher bzw. NULL bei Fehlern. 'file' gibt den Dateinamen des Soundfiles an. Unterstützt werden WAVE, AIFF, RIFF, OGG und VOC Dateien.

Soll die Lautstärke des Samples angepasst werden, kann dies mit Mix_VolumeChunk erreicht werden:

int Mix_VolumeChunk (Mix_Chunk chunk, int volume)*

Die ursprüngliche Lautstärke wird zurückgegeben. 'chunk' muss ein Pointer auf ein Sample im Speicher sein, 'volume' die neue Lautstärke, deren Maximum MIX_MAX_VOLUME ist.

Mit Mix_FreeChunk wird ein im Speicher liegendes Soundsample wieder entfernt:

void Mix_FreeChunk (Mix_Chunk chunk)*

'chunk' muss ein Zeiger auf das Sample im Speicher sein.

Die Anzahl der Kanäle, welche zum Abspielen zur Verfügung stehen und zwischen denen die Soundausgabe abgemischt wird, wird durch Mix_AllocateChannels angegeben:

int Mix_AllocateChannels (int numchans)

Die Anzahl der im Speicher allokierten Kanäle bildet den Rückgabewert. Wird als Parameter 0 übergeben, so werden alle Kanäle freigegeben. Ansonsten versucht der SDL_mixer 'numchans' Kanäle anzulegen. Eine maximal mögliche Anzahl bildet dabei der zur Verfügung stehende Speicher.

Für einzelne Kanäle kann jeweils eine bestimmte Lautstärke mit Mix_Volume gewählt werden:

int Mix_Volume (int channel, int volume)

Dabei wird die aktuelle Lautstärke zurückgegeben. Der Wert in 'volume' wird für alle Kanäle gesetzt, wenn 'channel' -1 ist.

Zum Abspielen können nun vier Funktionen verwendet werden:

int Mix_PlayChannel (int channel, Mix_Chunk chunk, int loops)*
int Mix_PlayChannelTimed (int channel, Mix_Chunk chunk, int loops, int ticks)*
int Mix_FadeInChannel (int channel, Mix_Chunk chunk, int loops, int ms)*
int Mix_FadeInChannelTimed (int channel, Mix_Chunk chunk, int loops, int ms, int ticks)*

Alle Funktionen liefern die Nummer des abzuspielenden Kanals zurück bzw. -1, sollten Fehler auftreten. Die Übergabe der ersten drei Parameter, nämlich 'channel' für die Kanal-Nummer, dem Pointer 'chunk' auf das Sample und die Anzahl der Wiederholungen 'loops', ist ebenfalls für alle Funktionen gleich. -1 als 'channel'-Angabe spielt den nächsten, nicht verwendeten Kanal ab. Besitzt 'loops' den Wert -1, wird das Sample in einer Endlosschleife abgespielt.

Mix_PlayChannelTimed spielt einen Kanal höchstens 'ticks' Millisekunden. Durch Mix_FadeInChannel kann ein Fade-In-Effekt erzielt werden, d.h. die Lautstärke wird von 0 beginnend 'ms' Millisekunden lang stetig bis auf das Maximum erhöht. Mix_FadeInChannelTimed verbindet die Eigenschaften der beiden vorgenannten Funktionen miteinander.

Momentan aktive Kanäle können mit Mix_Pause und Mix_Resume pausiert und wieder fortgeführt werden:

void Mix_Pause (int channel)
void Mix_Resume (int channel)

'channel' gibt die Nummer des gewünschten Kanals an. Mit -1 werden alle Kanäle angesprochen.

Folgende Funktionen können Kanäle anhalten:

int Mix_HaltChannel (int channel)
int Mix_ExpireChannel (int channel, int ticks)
int Mix_FadeOutChannel (int channel, int ms)

Der Rückgabewert der letzten beiden Funktionen gibt Auskunft über die betroffenen Kanäle. Mix_HaltChannel liefert immer 0 zurück. 'channel' gibt bei allen den anzuhaltenden Kanal an (-1 für alle Kanäle). Mix_ExpireChannel hält den Kanal nach 'ticks' Millisekunden an, Mix_FadeOutChannel führt einen Fade-Out-Effekt aus, d.h. die Lautstärke wird stetig 'ms' Millisekunden lang erniedrigt, bis sie 0 erreicht.

Informationen über einen bestimmten Kanal können hiermit bestimmt werden:

int Mix_Playing (int channel)
int Mix_Paused (int channel)
Mix_Fading Mix_FadingChannel (int which)
Mix_Chunk Mix_GetChunk (int channel)*

Die ersten beiden Funktionen liefern 1 zurück, wenn der angegebene Kanal momentan aktiv bzw. pausiert ist, 0 wenn dies nicht der Fall ist. Bei Angabe von -1 als Wert für 'channel' wird die Anzahl der Kanäle zurückgegeben, für welche die jeweilige Funktion 1 liefern würde.
-1 als Parameter für die letzten beiden Funktionen führt zu undefiniertem Verhalten. Mix_FadingChannel stellt den Fading-Status des Kanals fest. Dieser kann MIX_NO_FADING, MIX_FADING_OUT oder MIX_FADING_IN sein. Mix_GetChunk gibt den Pointer auf das momentan abgespielte Sample zurück bzw. NULL, sollte der Kanal nicht allokiert sein, oder noch kein Sample abgespielt haben.

B.2.3 Sound Effekte

Soll für einen Kanal eine Entfernung der Geräuschquelle simuliert werden, so ist dies durch Mix_SetDistance möglich:

int Mix_SetDistance (int channel, Uint8 distance)

Ein Rückgabewert ungleich 0 bedeutet hierbei Erfolg. 'channel' bestimmt die Nummer des Kanals. Der Wert kann auch MIX_CHANNEL_POST sein, um den Effekt auf den endgültig abgemischten Outputstream anzuwenden. 'distance' muss zwischen 0 (laut/nah, entfernt Effekt vom Kanal) und 255 (leise/weit entfernt) liegen.

Die Funktion Mix_SetPosition simuliert einen simplen 3D-Audio-Effekt:

int Mix_SetPosition (int channel, Sint16 angle, Uint8 distance)

Für den Rückgabewert, 'channel' und 'distance' gilt dasselbe wie für Mix_SetDistance. 'angle' gibt den Winkel zur Geräuschquelle in Grad an (0 entspricht vorne, fortlaufend im Uhrzeigersinn).

B.2.4 Musik Wiedergabe

Mit Mix_LoadMUS wird ein Musik-File in den Speicher geladen:

Mix_Music Mix_LoadMUS(const char* file)*

Es wird ein Zeiger auf das Musikstück im Speicher zurückgegeben, bzw. NULL bei Fehlern. 'file' gibt den Dateinamen an. Unterstützt werden die Formate WAVE, MOD, MIDI und OGG.

Zum freigeben des Speichers wird Mix_FreeMusic benutzt:

void Mix_FreeMusic (Mix_Music music)*

'music' muss der Pointer auf das Musik-Sample sein.

Die folgenden Funktionen starten die Musikwiedergabe:

int Mix_PlayMusic (Mix_Music music, int loops)*
int Mix_FadeInMusic (Mix_Music music, int loops, int ms)*

Die Verwendung erfolgt analog zur Soundwiedergabe.

Die Musik kann ebenso wie normale Kanäle pausiert und fortgeführt werden:

void Mix_PauseMusic ()
void Mix_ResumeMusic ()

Mit Mix_VolumeMusic wird, analog zu normalen Kanälen, die Lautstärke des Musik-Kanals gewählt:

int Mix_VolumeMusic (int volume)

Gestoppt wird die Musik durch folgende Funktionen, die sich ebenfalls äquivalent zum Anhalten eines Kanals der Soundausgabe verhalten:

int Mix_HaltMusic ()
int Mix_FadeOutMusic (int ms)

Um Informationen über den Musik-Kanal zu erhalten, werden ähnliche Funktionen zur Verfügung gestellt wie für Sound-Kanäle:

int Mix_PlayingMusic ()
int Mix_PausedMusic ()
Mix_Fading Mix_FadingMusic ()

Zusätzlich kann mit Mix_GetMusicType bestimmt werden, welche Art Musik geladen wurde:

Mix_MusicType Mix_GetMusicType (const Mix_Music music)*

Zurückgegeben werden die Konstanten MUS_NONE, MUS_CMD, MUS_WAV, MUS_MOD, MUS_MID, MUS_OGG oder MUS_MP3.

B.3 ODE

B.3.1 World

Ein neues World-Objekt wird durch dWorldCreate erzeugt:

dWorldID dWorldCreate ()

Die zurückgelieferte ID wird unter anderem zur Zerstörung der World verwendet:

void dWorldDestroy (dWorldID world)

Alle Bodies und Joints, die in der World enthalten waren, werden zerstört. Joints, die sich in einer Joint Group befinden (ein Container für Joints), werden deaktiviert, aber nicht gelöscht.

Der Gravitationsvektor kann mit dWorldSetGravity gesetzt werden:

void dWorldSetGravity (dWorldID world, dReal x, dReal y, dReal z)

Die Einheit ist m/s^2. Für Erdgravitation wäre (0, 0, -9.81) der anzugebende Vektor (vorausgesetzt die positive Z-Achse zeigt nach oben). Als Standard wird Nullgravitation verwendet.

Ein neuer Simulationsschritt wird durch Aufruf von dWorldStep durchgeführt:

void dWorldStep (dWorldID world, dReal stepsize)

'stepsize' gibt die Zeit des Simulationsschrittes in Sekunden an. Alle Objekte in der World werden aufgrund ihrer aus den wirksamen Kräften berechneten Geschwindigkeit bewegt.

Wird die ODE nicht mehr benötigt, so sollte dCloseODE verwendet werden, um ggf. allokierten Speicher wieder freizugeben:

void dCloseODE ()

B.3.2 Rigid Body Funktionen

Einzelne Körper können der World mit dBodyCreate hinzugefügt werden:

dBodyID dBodyCreate (dWorldID world)

Der Body wird in die entsprechende World integriert und die ID des neuen Körpers wird zurückgegeben.
dBodyDestroy entfernt einen Körper:

dBodyDestroy (dBodyID body)

Mit folgenden Funktionen können die Startwerte für Position, Rotation, Linear- und Winkelgeschwindigkeit eines Bodies gesetzt werden:

void dBodySetPosition (dBodyID body, dReal x, dReal y, dReal z)
void dBodySetRotation (dBodyID body, const dMatrix3 R)
void dBodySetQuaternion (dBodyID body, const dQuaternion q)
void dBodySetLinearVel (dBodyID body, dReal x, dReal y, dReal z)
void dBodySetAngularVel (dBodyID body, dReal x, dReal y, dReal z)

In Bezug auf die Winkelgeschwindigkeit gibt der Vektor die Drehachse und der Betrag des Vektors die Geschwindigkeit an. Rotationen können entweder als 3x3-Matrix oder als Quaternion übergeben werden.

Um die aktuellen Werte eines Körpers auszulesen dienen diese Funktionen:

const dReal dBodyGetPosition (dBodyID body)*
const dReal dBodyGetRotation (dBodyID body)*
const dReal dBodyGetQuaternion (dBodyID body)*
const dReal dBodyGetLinearVel (dBodyID body)*
const dReal dBodyGetAngularVel (dBodyID body)*

Zurückgegeben wird ein Pointer auf ein Array von float-Werten, das für Position, Linear- und Winkelgeschwindigkeit aus 3 Elementen und für Rotation entweder aus 12 (4x3 Matrix) bzw. 4 bei einem Quaternion besteht.

Einem Körper kann ein Zeiger auf benutzerdefinierte Daten durch dBodySetData gegeben werden:

void dBodySetData (dBodyID body, void data)*

Mittels dBodyGetData kann nun auf diese Daten wieder zugegriffen werden:

void dBodyGetData (dBodyID body)*

Körper können auch durch folgende Funktionen ein- und ausgeschaltet werden:

void dBodyEnable (dBodyID body)
void dBodyDisable (dBodyID body)

Deaktivierte Körper werden beim nächsten Worldstep nicht in die Berechnungen miteinbezogen.

Ob ein Body aktiv ist, kann mit dBodyIsEnabled abgefragt werden:

int dBodyIsEnabled (dBodyID body)

Ist der Körper aktiv, wird 1 zurückgegeben, andernfalls 0.

B.3.3 Funktionen für Masse und Kräfte

Der Masse für einen Körper liegt folgende Struktur zugrunde:

```
typedef struct dMass
{
        dReal mass;
        dVector4 c;
        dMatrix3 I;
} dMass;
```

'mass' gibt die Gesamtmasse des Bodies an, 'c' das Zentrum der Masse bzgl. des Körpers und 'I' beschreibt die Masseverteilung innerhalb des Körpers und wirkt als eine Art Skalierungsfaktor zwischen Drehmoment und Winkelgeschwindigkeit.
Es stehen nun unterschiedliche Funktionen zur Verfügung, um Massen verschiedener Form zu erzeugen. Im Rahmen der Implementierung wurde lediglich dMassSetSphereTotal verwendet:

void dMassSetSphereTotal (dMass mass, dReal total_mass, dReal radius)*

Die Parameter der Masse 'mass' werden durch die Gesamtmasse der Sphäre 'total_mass' und ihren Radius 'radius' entsprechend gesetzt.

Die Zuweisung der Masse an einen Körper erfolgt mit dBodySetMass:

void dBodySetMass (dBodyID body, const dMass mass)*

Mit dBodyGetMass kann diese Masse wieder ausgelesen werden:

void dBodyGetMass (dBodyID body, dMass mass)*

Körper in einer World können nun durch hinzufügen von Kräften manipuliert werden:

void dBodyAddForce (dBodyID body, dReal fx, dReal fy, dReal fz)
void dBodyAddTorque (dBodyID body, dReal fx, dReal fy, dReal fz)
void dBodyAddRelForce (dBodyID body, dReal fx, dReal fy, dReal fz)
void dBodyAddRelTorque (dBodyID body, dReal fx, dReal fy, dReal fz)
void dBodyAddForceAtPos (dBodyID body, dReal fx, dReal fy, dReal fz,
dReal px, dReal py, dReal pz)
void dBodyAddForceAtRelPos (dBodyID body, dReal fx, dReal fy, dReal fz,
dReal px, dReal py, dReal pz)
void dBodyAddRelForceAtPos (dBodyID body, dReal fx, dReal fy, dReal fz,
dReal px, dReal py, dReal pz)
void dBodyAddRelForceAtRelPos (dBodyID body, dReal fx, dReal fy, dReal fz,
dReal px, dReal py, dReal pz)
Die Angaben für die Koordinaten können absolut oder relativ zum Körper sein (dBodyAddRel...). "...AtPos"- und "...AtRelPos"-Funktionen besitzen außerdem noch Parameter für die Angabe eines Punktes, in absoluten oder relativen Koordinaten, an dem die Kraft wirksam wird. Die akkumulierten Kräfte werden nach jedem Worldstep wieder auf 0 gesetzt.

Die aktuell wirksame Kraft wird durch dBodyGetForce zurückgegeben:

const dReal dBodyGetForce (dBodyID body)*

Gleiches gilt für das Drehmoment durch dBodyGetTorque:

const dReal dBodyGetTorque (dBodyID body)*

Mit den folgenden Funktionen können direkt die akkumulierten Kräfte bzw. das Drehmoment gesetzt werden:

void dBodySetForce (dBodyID body, dReal x, dReal y, dReal z)
void dBodySetTorque (dBodyID body, dReal x, dReal y, dReal z)

Dies ist in erster Linie für deaktivierte Körper nützlich, um sicherzustellen, dass auf sie wirksame Kräfte auf 0 gesetzt werden, wenn sie reaktiviert werden.

B.3.4 Geoms

Durch dCreateSphere wird ein neues Sphere-Geom erzeugt:

dGeomID dCreateSphere (dSpaceID space, dReal radius)

Zurückgegeben wird die ID des neuen Geoms. Ist die Space ID 'space' ungleich 0, wird der erzeugte Geom in den angegebenen Space eingefügt.

Der Radius kann mit dGeomSphereSetRadius geändert und mit dGeomSphereGetRadius zurückgegeben werden:

void dGeomSphereSetRadius (dGeomID geom, dReal radius)
dReal dGeomSphereGetRadius (dGeomID geom)

Eine Ebene wird mit dCreatePlane erzeugt:

dGeomID dCreatePlane (dSpaceID space, dReal a, dReal b, dReal c, dReal d)

Die Ebenengleichung ist a*x + b*y + c*z = d, wobei es sich bei (a, b, c) um den Normalenvektor der Ebene handelt, der die Länge 1 haben muss. Ebenen sind non-placeable Geoms und besitzen somit weder eine Position noch eine Rotation. Die übergebenen Parameter erwarten folglich globale Koordinaten.

dGeomDestroy zerstört einen Geom unabhängig von seiner Klasse:

void dGeomDestroy (dGeomID geom)

Benutzerdefinierte Daten können durch folgende Funktionen dem Geom zugewiesen und wieder ausgelesen werden:

void dGeomSetData (dGeomID geom, void)*
void dGeomGetData (dGeomID geom)*

Um nun einem Geom den entsprechenden Körper zuzuweisen, wird dGeomSetBody verwendet:

void dGeomSetBody (dGeomID geom, dBodyID body)

Geom und Body besitzen nun dieselbe Position und Rotation. Werden bei einem von beiden diese Werte verändert, gilt dies ebenfalls für den anderen.

Die ID des assoziierte Body wird durch dGeomGetBody zurückgegeben:

dBodyID dGeomGetBody (dGeomID geom)

Die folgenden Funktionen können zur Manipulation des Geoms angewandt werden, und verhalten sich analog zu den Funktionen von Rigid Bodies:

void dGeomSetPosition (dGeomID geom, dReal x, dReal y, dReal z)
void dGeomSetRotation (dGeomID geom, const dMatrix3 R)
void dGeomSetQuaternion (dGeomID geom, const dQuaternion)
const dReal dGeomGetPosition (dGeomID geom)*
const dReal dGeomGetRotation (dGeomID geom)*
void dGeomGetQuaternion (dGeomID geom, dQuaternion result)

Zu beachten ist, dass non-placeable Geoms nicht mit einem Body verbunden und auch nicht manipuliert werden können. In der Debug-Version der ODE würde dies zu einem Laufzeitfehler führen.

Ebenfalls analog zu Bodies können Geoms aktiviert und deaktiviert werden:

void dGeomEnable (dGeomID geom)
void dGeomDisable (dGeomID geom)
int dGeomIsEnabled (dGeomID geom)

B.3.5 Spaces

Es gibt zwei Arten von Spaces, die folgendermaßen erzeugt werden:

dSpaceID dSimpleSpaceCreate (dSpaceID space)
dSpaceID dHashSpaceCreate (dSpaceID space)

Ist die Space ID ungleich 0 wird der erzeugte Space in den bestehenden eingefügt.

"Simple Spaces" prüfen jedes Objekt mit jedem anderen auf Kollision. Die dabei benötigte Zeit beträgt $O(n^2)$ für n Objekte, weswegen diese Art Space nur für eine geringe Zahl an Geoms geeignet ist.
"Hash Spaces" benutzen einen Octtree, um die Position der Geoms zu überprüfen und schnell geeignete Kollisionskandidaten zu identifizieren. Die verwendeten dreidimensionalen Zellen haben eine Kantenlänge von 2^i, wobei für den Integer i ein Minimum und Maximum vorgegeben werden kann. Für n Objekte beträgt die Zeit, um sie auf Überschneidungen zu prüfen, O(n), solange die Objekte nicht zu nah beieinander liegen.

Das Minimum und Maximum für die Zellen wird mit dHashSpaceSetLevels gesetzt:

void dHashSpaceSetLevels (dSpaceID space, int minlevel, int maxlevel)

Ein Space wird durch dSpaceDestroy zerstört:

void dSpaceDestroy (dSpaceID space)

Für den Space kann ein "Cleanup"-Modus durch folgende Funktionen gewählt und abgefragt werden:

void dSpaceSetCleanup (dSpaceID space, int mode)
int dSpaceGetCleanup (dSpaceID space)

Standardwert für den Modus ist 1, d.h. bei Zerstörung des Space werden sämtliche in ihm enthaltene Geoms ebenfalls gelöscht. Bei Modus 0 ist dies nicht der Fall.

Geometrie Objekte, die bei ihrer Erstellung keinem Space zugeordnet wurden, können mit diesen Funktionen hinzugefügt bzw. entfernt werden:

void dSpaceAdd (dSpaceID space, dGeomID geom)
void dSpaceRemove (dSpaceID space, dGeomID geom)

Die Anzahl der im Space enthaltenen Geoms kann mit dSpaceGetNumGeoms ermittelt werden:

int dSpaceGetNumGeoms (dSpaceID space)

Mit Hilfe von dSpaceQuery kann überprüft werden, ob sich ein Geom in einem bestimmten Space befindet:

int dSpaceQuery (dSpaceID space, dGeomID geom)

Rückgabewert 1 signalisiert, dass sich der Geom im Space befindet, 0 wenn dies nicht zutrifft.

B.3.6 Kontaktpunkte, Contact Joints und Kollisionsabfrage

Contact Joints werden durch dJointCreateContact generiert und durch dJointDestroy wieder zerstört:

dJointID dJointCreateContact (dWorldID world, dJointGroupID group,
const dContact contact)*
void dJointDestroy (dJointID joint)

Der erzeugte Contact Joint wird in 'contact' gespeichert und der Joint Group 'group' in der World 'world' hinzugefügt.

Eine Joint Group ist ein Container für verschiedene Joints und wird mit dJointGroupCreate gebildet und mit dJointGroupDestroy gelöscht:

dJointGroupID dJointGroupCreate (int max_size)
void dJointGroupDestroy (dJointGroupID group)

'max_size' dient der Rückwärtskompatibilität und sollte auf 0 gesetzt werden.

Alle Joints einer Group können mit dJointGroupEmpty wieder entfernt werden:

void dJointGroupEmpty (dJointGroupID group)

Die Struktur eines Contact Joints sieht folgendermaßen aus:

```
struct dContact
{
    dSurfaceParameters surface;
    dContactGeom geom;
    dVector3 fdir1;
};
```

'fdir1' gibt die erste Richtung der Reibungskraft an und wird nur verwendet, wenn das dContactFDir1-Flag in surface.mode gesetzt wurde. In 'geom' werden Informationen über die an der Kollision beteiligten Geoms gespeichert.

Über 'surface' können die Eigenschaften der Kollision vorgegeben werden:

```
struct dSurfaceParameters
{
        int mode;
        dReal mu;
        dReal mu2;
        dReal bounce;
        dReal bounce_vel;
        dReal soft_erp;
        dReal soft_cfm;
        dReal motion1,motion2;
        dReal slip1,slip2;
} dSurfaceParameters;
```

'mode' muss eine durch bitweises OR verknüpfte Kombination folgender Konstanten sein:

dContactMu2	Falls gesetzt wird 'mu' für Reibungsrichtung 1 und 'mu2' für Reibungsrichtung 2 verwendet, ansonsten wird 'mu' für beide Richtungen verwendet.
dContactFDir1	Falls gesetzt wird fdir1 als erste Reibungsrichtung verwendet, andernfalls wird für die Richtung die Senkrechte zur Kontaktnormalen angenommen.
dContactBounce	Falls gesetzt kann die Elastizität der Kollision durch den Parameter 'bounce' angegeben werden. 'bounce_vel' bestimmt dabei die Mindestgeschwindigkeit, um ein Abprallen der Geoms hervorzuführen.

dContactSoftERP	Falls gesetzt kann die Fehlertoleranz für die Kontaktnormale durch 'soft_erp' gesetzt werden. Oberflächen werden dadurch "weicher".
dContactSoftCFM	Falls gesetzt kann mit 'soft_cfm' der Faktor für den "Zusammenhalt" der Geoms am Gelenkpunkt bestimmt werden. Dies kann ebenfalls dazu verwendet werden, um Oberflächen "weicher" zu machen.
dContactMotion1	Falls gesetzt kann mit 'motion1' eine Geschwindigkeit für die Oberfläche der Geoms in Richtung der ersten Reibungsrichtung gesetzt werden.
dContactMotion2	gleiches wie oben, nur für 'motion2' und die zweite Reibungsrichtung
dContactSlip1	Falls gesetzt kann durch Angabe von 'slip1' ein Gleitfaktor für die erste Reibungsrichtung angegeben werden.
dContactSlip2	gleiches wie oben, nur für 'slip2' und die zweite Reibungsrichtung.
dContactApprox1_1	Falls gesetzt soll für die erste Reibungsrichtung der Reibungskegel durch eine Pyramide angenähert werden
dContactApprox1_2	gleiches wie oben, nur für die zweite Reibungsrichtung
dContactApprox1	äquivalent zu dContactApprox1_1 und dContactApprox1_2

Die Struktur von 'geoms' im Contact Joint sieht folgendermaßen aus:

```
struct dContactGeom
{
        dVector3 pos;
        dVector3 normal;
        dReal depth;
        dGeomID g1,g2;
};
```

'pos' gibt die Position des Kontaktpunktes an, 'normal' die Normale auf die Kontaktfläche. 'depth' beschreibt, wie weit die Geoms 'g1' und 'g2' während der Kollision ineinander eingetaucht sind.

Die Kollisionsabfrage selber wird mit dCollide durchgeführt:

int dCollide (dGeomID o1, dGeomID o2, int flags, dContactGeom contact, int skip)*

Zurückgegeben wird die Anzahl der erzeugten Kontaktpunkte. 'o1' und 'o2' sind die Geoms, die auf Kollision geprüft werden sollen. Die niederwertigen 16 Bits von 'flags' geben die Zahl der Kontaktpunkte an, die generiert werden sollen. Die restlichen Bits müssen 0 sein und sind für eine Verwendung in zukünftigen Versionen der ODE reserviert. Ist 'flags' gleich 0, so wird es auf 1 gesetzt. 'contact' muss auf ein Array von dContactGeom-Elementen zeigen, in denen die Kontaktpunkte gespeichert werden. Das Byte-Offset des Arrays kann durch 'skip' angegeben werden und darf nicht kleiner als sizeof (dContactGeom) sein.

Um die Vorteile von Spaces nutzen zu können, muss dSpaceCollide benutzt werden:

void dSpaceCollide (dSpaceID, void data, dNearCallback* callback)*

Diese Funktion prüft alle Geoms im gegebenen Space auf Kollisionen und ruft für jedes gefundene Paar die Callback-Funktion dNearCallback auf. 'data' wird dabei als Parameter an diese Funktion weitergeleitet.
Die Callback-Routine muss wie folgt deklariert werden:

void dNearCallback (void data, dGeomID o1, dGeomID o2)*

'data' wird von dSpaceCollide weitergegeben. 'o1' und 'o2' sind Geoms, welche potentielle Kandidaten für eine Kollision sind. Ob diese beiden Geometrie Objekte wirklich miteinander kollidieren, muss in dNearCallback geprüft und entsprechend reagiert werden (Contact Joints mit dCollide generieren etc.).

C. Glossar

Adventure: Adventures oder Abenteuerspiele bezeichnen rätselbasierte Computerspiele, meist ohne Actionelemente. Die Rätsel sind häufig in eine Handlung eingebettet.

Bump Mapping: Technik, mit der versucht wird, einem Polygon Plastizität zu verleihen, indem sich die Textur dynamisch an Veränderungen des Lichteinfalls anpasst.

Computer Role Playing Game: Auch als CRPG bezeichnet. Versuch, P&P (Pencil and Paper) Rollenspiel-Strukturen in ein Computerspiel zu übernehmen, d.h. der Spieler bewegt sich in einer fantastischen Welt, besitzt bestimmte Attribute und Fähigkeiten und soll sich seiner Charakterklasse entsprechend verhalten.

Demo: Bezeichnung für eine Multimedia-Anwendung, durch die der Autor die Fähigkeiten eines bestimmten Rechners demonstrieren und ausreizen will.

Emulator: Eine virtuelle Maschine, welche versucht eine andere Rechnerkonfiguration möglichst exakt nachzuahmen.

Jump'n'Run: Bezeichnung für ein Plattformspiel am Computer, bei dem der Spieler in erster Linie durch geschicktes Hüpfen Fallen und Gegnern ausweichen muss.

Lens Flares: Reflektionen, die bei grellem Licht durch die Linsen einer Kamera auftreten können. Beim 3D-Rendering wird damit eine Technik bezeichnet, diese Reflektionen realistisch am Computer wiederzugeben.

Level-Editor: Werkzeug, mit dem die Struktur der Spielwelten eines Spiels bestimmt werden kann.

Model-Editor: Werkzeug zum Erstellen von dreidimensionalen Modellen am Computer.

Multitexturing: Mehrere Texturebenen werden einem Polygon zugewiesen, um einen höheren Detailgrad zu erreichen.

Open-Source Software: Software, deren Sourcecode veröffentlicht worden ist.

Rigid Bodies: Bezeichnung für nicht verformbare Körper.

Script-Editor: Werkzeug zum Erstellen von Skripten in einer vorher definierten Skriptsprache.

Shooter: Computeractionspiel mit dem Hauptziel, reaktionsschnell virtuelle Gegner abzuschießen.

Simulationen: In Bezug auf Computerspiele wird hierbei ein Genre beschrieben, dass sich so genau wie möglich an wissenschaftliche Vorgaben der Realität hält.

Spiele-SDK: Sammlung verschiedener Werkzeuge zur Vereinfachung der Spieleentwicklung.

D. Abkürzungsverzeichnis

AI	**A**rtificial **I**ntelligence
AIFF	**A**udio **I**nterchange **F**ile **F**ormat
API	**A**pplication **P**rogramming **I**nterface
ArP	**Ar**tificial **P**lanetoids
ASCII	**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange
CRPG	**C**omputer **R**ole **P**laying **G**ame
eof	**e**nd **o**f **f**ile
FPS	**F**rames **P**er **S**econd
GLUT	Open**GL** **U**tility **T**oolkit
GUI	**G**raphical **U**ser **I**nterface
HUD	**H**ead **U**p **D**isplay
ISR	**I**nterrupt **S**ervice **R**outine
KI	**K**ünstliche **I**ntelligenz
MOD	Noisetracker/Soundtracker/Protracker **Mod**ule Format
ODE	**O**pen **D**ynamics **E**ngine
OGG	**Ogg** Vorbis File Format
OpenAL	**Open** **A**udio **L**ibrary
OpenGL	**Open** **G**raphics **L**ibrary
RIFF	**R**esource **I**nterchange **F**ile **F**ormat
SDK	**S**oftware **D**evelopment **K**it
SDL	**S**imple **D**irectMedia **L**ayer
TGA	**T**ar**ga** Image File Format
VOC	Creative **Voic**e File Format

E. Literaturverzeichnis

Bücher:

DeLoura, Mark A. (edt.): Game Programming Gems, Charles River Media Inc., 2000

Eberly, David H.: 3D Game-Engine Design - A Practical Approach to Real-Time Computer Graphics, Morgan Kaufmann, 2001

Hawkins, Kevin und Dave Astle: OpenGL Game Programming, Prima Tech's Game Development Series, 2001

OpenGL Architecture Review Board: OpenGL Programming Guide Third Edition, Addison-Wesley, 2001

OpenGL Architecture Review Board: OpenGL Reference Manual Third Edition, Addison-Wesley, 2001

Zerbst, Stefan: 3D Spieleprogrammierung mit DirectX Band I+II, Libri Books on Demand, 2000

Internet:

[1] http://www.8bit-museum.de/, 20.03.2004

[2] http://dictionary.com/, 20.03.2004

[3] http://www.gametutorials.com/, 20.03.2004

[4] http://www.javaworld.com/javaworld/jw-08-1998/jw-08-step.html, 20.03.2004

[5] http://www.libsdl.org/, 20.03.2004

[6] http://www.libsdl.org/projects/SDL_mixer/, 20.03.2004

[7] http://www.mvps.org/directx/articles/writing_the_game_loop.htm, 20.03.2004

[8] http://nehe.gamedev.net/, 20.03.2004

[9] http://opende.sourceforge.net/, 20.03.2004

[10] http://www.spatial-effects.com/SE-research1.html, 20.03.2004

[11] http://www.uni-wuerzburg.de/sopaed1/vernooij/spiel/spiel.htm, 20.03.2004

[12] http://www.wotsit.org/, 20.03.2004