Department für Informatik - Abteilung Rechnernetze und Telekommunikation

Masterarbeit

Entwicklung eines Java-basierten Frameworks zur Erstellung von 3D-Point-and-Click-Adventure-Spielen

Bearbeitungszeitraum: 06.06.2007 - 06.12.2007

vorgelegt von:
Frank Bruns
Kirchstr. 58
49757 Werlte

Oldenburg, den 06.12.2007

erster Gutachter:
Prof. Dr. Wolfgang Kowalk

zweiter Gutachter:
Stefan Brunhorn, Dipl.-Inf.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Video-Spiele sind, vor allem bei denjenigen Menschen, die schon in ihrer Kindheit mit ihnen in Berührung gekommen sind, als gebräuchliches Unterhaltungsmedium etabliert. Sie erscheinen uns ebenso alltäglich und selbstverständlich, wie das Angebot an Funk- und Fernsehprogrammen. Insbesondere Spiele mit ausgeprägter Story-Tiefe und identifikationsfähigen Charakteren laden dazu ein, vorübergehend in eine fantastische Erlebniswelt abzutauchen. Oftmals kommen sie sogar deutlich fesselnder daher, als ihre filmischen Pendants.

Auch ich bin schon lange von den erzählerischen Möglichkeiten der Video-Spiele in einen begeisterten Bann gezogen worden. Blicke ich zurück auf die Titel, die mir in meiner langjährigen Laufbahn als Konsument am stärksten in Erinnerung geblieben sind, so fallen mir vorrangig Spiele ein, die sehr viel Wert auf das Erzählen einer spannenden Geschichte gelegt haben. Dabei kam es mir weniger darauf an, ob die Handlung in einer fortschrittlichen Science-Fiction-Welt, einer düsteren Zukunftsutopie oder einfach in einer Gegenwart, die der unsrigen nachempfunden ist, angesiedelt war. Hauptsache die Atmosphäre wurde glaubhaft vermittelt und die Hintergrundgeschichte stimmig präsentiert.

Gerade Adventure-Spiele eignen sich besonders gut, um komplexe Geschichten zu erzählen. Dieses Projekt behandelt daher die Entwicklung eines Frameworks für 3D-Point-and-Click-Adventures in Java. Mehrere Gründe haben mich dazu motiviert, ein solches Unterfangen in Angriff zu nehmen.

Zuallererst ist meine neu entdeckte Leidenschaft für die Spiele-Programmierung zu nennen. Als ich das Buch „Developing Games in Java“ von David Brackeen in die Hände bekam, war ich sofort Feuer und Flamme bei dem Gedanken, irgendwann selbst Spiele in meiner bevorzugten Programmiersprache implementieren zu können. Das Timing war perfekt, als sich Prof. Wolfgang P. Kowalk entschloss, im Sommersemester 2006 erstmalig eine Veranstaltung zu OpenGL mit Java anzubieten. Diese Veranstaltung legte den Wissensgrundstock für die Entwicklung von dreidimensionalen Spielen. Durch eine gewisse Vorliebe für Point-and-Click-Adventures fasste ich den Schluss, ein Framework zu entwickeln, mit dem sich solche Spiele flexibel realisieren lassen. Meine ersten Gehversuche in diesem Zusammenhang waren jedoch klassisch 2D-basiert, führten aber schon zu einem recht annehmbaren Resultat. Als später die Frage auf kam, worüber ich gerne meine Abschlussarbeit schreiben möchte, war mir schnell klar, dass sie sich im Themenspektrum Spiele-Programmierung bewegen sollte. Mir kam der Gedanke, das Adventure-Framework von Grund auf neu zu entwickeln und dabei Fehler der alten Variante zu vermeiden. Dieses Mal sollte es allerdings nicht auf 2D-Grafik zurückgreifen, sondern eine selbst zu entwickelnde 3D-Engine erhalten.

Ich habe im Vorfeld des Projektbeginns recherchiert, ob ein vergleichbares System im Open-Source-Bereich bereits exisitiert. Im 2D-Sektor bin ich zwar vereinzelt fündig geworden, doch blieb die Suche ergebnislos, wenn es darum ging, 3D-Adventures zu erstellen.

Kommerzielle Systeme sind sicherlich vorhanden, wie entsprechende Spiele belegen, jedoch stellen diese das Kapital eines Unternehmens dar, welches als besonders schützenswert eingestuft und deshalb verschlossen gehalten wird. Dies ist ein weiterer motivierender Faktor für mich, eine Eigenentwicklung zu wagen.

In gewisser Weise erfülle ich mir durch dieses Projekt auch einen Wunschtraum. Ich bin durch Video-Spiele an den Computer herangeführt worden und verdanke diesem Interesse vermutlich auch die Entscheidung, ein Informatik-Studium aufzunehmen. Im Hinterkopf behielt ich dabei immer die Vorstellung, von der Perspektive des Konsumenten auf die Seite des Entwicklers vorzustoßen. Mit zunehmendem Interesse an der Materie rückte aber noch ein weiter Aspekt des Ansporns in den Mittelpunkt. Allenthalben stößt man auf Ablehnung und Vorurteile, wenn man Java in einem Atemzug mit Spiele-Programmierung nennt. Ich verstehe dieses Projekt deshalb auch als Medium der Beweisführung für die Tauglichkeit Javas als ernst zunehmende Basis-Technologie in der Spiele-Industrie.

Zu guter letzt treibt mich der Wille an, weitere Erfahrungen in diesem äußerst kreativen Segment der Software-Entwicklung zu sammeln und ein Projekt durchzuführen, mit dem ich mich aus vollstem Herzen identifizieren kann.

1.2 Ziel der Masterarbeit

In dieser Masterarbeit soll ein Framework entwickelt werden, welches die Erstellung so genannter Point-and-Click-Adventure-Spiele erlaubt. Im Gegensatz zu einer Vielzahl von Vertretern dieses Genres, sollen Spiele, die mit dem zu entwerfenden System angefertigt werden, in einer dreidimensionalen Umgebung situiert sein.

Der gesamte Aufgabenbereich der Implementierung kann grob in drei Teile gegliedert werden, die nachfolgend kurz abgerissen werden:

1. **Implementierung einer 3D-Game Engine zur Darstellung und Transformation der Spielwelt.** Basierend auf OpenGL (bzw. JOGL) soll eine 3D Game Engine entwickelt werden, die auf die Bedürfnisse von Adventure-Spielen zugeschnitten ist. Es muss demnach möglich sein, eine dreidimensionale Spielwelt in Echtzeit zu rendern. Solche Spielwelten bestehen im Allgemeinen aus einer Menge von texturierten 3D-Objekten, die überwiegend mittels spezieller Modellierungssoftware erstellt wurden und über die 3D-Engine importiert werden müssen. Die Spielumgebung soll in einzelne Abschnitte (Räume) unterteilt und separat gerendert werden können. Die Beschreibung eines jeden Abschnittes soll in externen Dateien vorgenommen werden, so dass Änderungen ohne Programmierkenntnisse vorgenommen werden können. Ein Spieler kann sich besser mit der Spielumgebung identifizieren, wenn sie optisch ansprechend und glaubhaft wirkt. Deshalb ist es wichtig, eine gewisse grafische Qualität zu erreichen. Echtzeitschattentechniken und Partikeleffekte könnten dazu ihren Beitrag leisten.
   Um den Eindruck einer dynamischen und interaktiven Welt zu vermitteln, ist es nötig, dass die 3D-Objekte nach Belieben transformiert und per Mausklick selektiert werden können. Gegebenenfalls müssen Methoden zur Kollisionserkennung zwischen Objekten implementiert werden. Auf jeden Fall aber sind Wegfindungsroutinen nötig, die die Spielfiguren bei Bewegungen von einem Startpunkt zu einem Zielpunkt an Hindernissen entlang navigieren. Für Spiele mit ausgeprägten Konversationsanteilen, wie Adventures es für gewöhnlich sind, ist es essenziell, dass Dialogtexte sowohl visuell als auch akustisch wiedergegeben werden können. Hierzu muss die Möglichkeit gegeben sein, Texte an definierten Bildschirmpostionen zu rendern und synchron dazu (komprimierte) Sounddateien abzuspielen.

2. **Implementierung eines auf Adventure-Spiele ausgerichteten Programmkerns.** Der Programmkern dient zur Verarbeitung der adventuretypischen Spiellogik. Als Spiellogik bezeichne ich die Interaktion mit der Spielwelt und deren Folgen. Der Spieler soll durch seine Aktionen den gegenwärtigen Zustand der Spielwelt dynamisch verändern können und infolgedessen mit neuen Situationen konfrontiert werden. Die Spiellogik soll, durch die Anbindung einer Skript-Sprache, in externe Dateien ausgelagert werden. Die Skript-Sprache muss dabei auf die Objekte der Spielwelt manipulativ zugreifen können. Dieses Vorgehen ermöglicht es, das jeweilige Spiel um neue Funktionalitäten, bzw. Möglichkeiten zur Interaktion, zu Ergänzen, ohne den Quellcode des Programmkerns anpassen bzw. neu kompilieren zu müssen. Auf diese Weise ist es möglich ein hochgradig generisches Framework zu entwickeln. Es gibt zwei wesentliche Bestandteile des Programmkerns, die besondere Erwähnung

finden sollten. Einerseits muss ein Dialogsystem implementiert werden, dass es ermöglichen soll, bei Gesprächen eine Auswahl an alternativen Frage- bzw. Antwortmöglichkeiten zu bieten. Andererseits wird eine sequenzielle Verarbeitung von Ereignissen im Spiel benötigt, die auf einer FIFO-Schlange basiert. Dadurch lassen sich, beispielsweise als Reaktion auf eine Spieleraktion, eine ganze Reihe von Ereignissen (z.B. Zwischensequenzen) auslösen, die sequenziell auftreten.

3. ***Umsetzung eines Beispiel-Adventures mit dem erstellten Framework.*** Um die Fähigkeiten des entworfenen Systems anschaulich zu demonstrieren, soll auf dessen Basis ein kleineres Point-and-Click-Adventure erstellt werden. Hierzu gehört die Entwicklung einer überschaubaren Storyline mit unterschiedlichen Charakteren und sinnvoll in ansehnliche Umgebungen eingebettete Rätselaufgaben.

1.3 Aufbau der Masterarbeit

Dieser Projektbericht macht es sich zur Aufgabe, einen Überblick über die Dinge zu verschaffen, die mit der Entwicklung des Adventure-Frameworks unmittelbar zusammenhängen.

Die Ausarbeitung beginnt in Kapitel 2 mit einigen theoretischen Grundlagen und Fragestellungen, die den Leser an die Thematik heranführen.

Damit die Methoden der Entwicklung und Implementierung im Verlauf der Arbeit nachvollzogen werden können, muss ein elementares Verständnis der verwendeten Technologien vorausgesetzt werden. Um diese Wissensbasis zu schaffen, erfolgt in Kapitel 3 eine relativ knappe Beschreibung substanzieller Aspekte der zugrunde liegenden Basistechnologien. Im ganzen Projektbericht wird außerdem häufig auf Schaubilder und Beispiele zurückgegriffen, die das Verständnis erleichtern und Gesichtspunkte belegen sollen.

Das nachfolgende Kapitel 4 orientiert sich grob an gängigen Prozessen der Softwareentwicklung, bestehend aus Analyse (mit Anforderungsdefinition), einem fundierten Entwurfs-Konzept und der Implementierungsphase. Dieses Kapitel ist jedoch in zwei große Teile gegliedert. Während der erste Teil sich mit der Entwicklung der 3D-Engine befasst, wird im zweiten Teil entsprechend das Adventure-Framework betrachtet.

Mit der Umsetzung eines Demo-Spiels in Kapitel 5 wird auszugsweise verdeutlicht, wie das entwickelte Framework praktisch anzuwenden ist.

Inhaltlich wird der Projektbericht mit meinen resümierenden Schlussbetrachtungen in Kapitel 6 abgeschlossen. Im Vordergrund steht dort ein knapper Rückblick auf die Ergebnisse des Projekts sowie ein kurzer Ausblick auf die Weiterentwicklung des Systems.

Für Personen, die sich mit dem oder einem anderen Punkt der Arbeit näher beschäftigen wollen, dient das Literaturverzeichnis in Kapitel 7.

Zum schnellen Auffinden gesuchter Beispiele und Schaubilder wurde ein Abbildungsverzeichnis (Kapitel 8) am Ende des Projektberichts eingefügt.

Das Stichwortverzeichnis in Kapitel 9 ist ein Anker zum schnellen Auffinden zentraler Begriffe innerhalb des Dokuments.

2 Einführende Grundlagen

Diesem Grundlagen-Kapitel kommt die Aufgabe zu, dem Leser einführende Hintergrundinformationen zum Verständnis der Arbeit näher zu bringen. Im Vordergrund steht dabei die Vermittlung technischen Basiswissens. Mitunter werden jedoch auch allgemeinere Bereiche angeschnitten.

2.1 Point-and-Click-Adventures

Ich den beiden nachfolgenden Subkapiteln möchte ich erstens versuchen, mit eigenen Worten zu erklären, was eine Point-and-Click-Adventure im Kern ist, und zweitens erörtern, worin der Reiz an solchen Spielen bestehen könnte.

2.1.1 Was ist ein Point-and-Click-Adventure?

Die Bezeichnung *Point-and-Click-Adventure* ist der Sammelbegriff für ein ganzes Genre von Videospielen. Sie hat sich innerhalb der spieleproduzierenden Industrie und der internationalen Fachpresse längst durchgesetzt. Ähnlich wie bei den Genrebezeichnungen *Jump 'n' Run* (dt.: Springen u. Rennen) oder *Beat 'em Up* (dt. etwa: Verprüg'le sie) steht der Ausdruck Point-and-Click (dt.: Draufzeigen und Klicken) für eine Tätigkeit, die das Fundament des Spielprinzips bildet. Der Spieler bewegt seine Spielfigur mausgesteuert, d.h. durch Klicken auf die gewünschte Zielposition, durch diverse Umgebungen in der Spielwelt. Dabei sammelt er, ebenfalls per Mausklick, Gegenstände ein oder beginnt Unterhaltungen mit Nicht-Spielercharakteren (Non-Player-Characters; kurz: NPCs). Die Anwendung der gesammelten Gegenstände auf geeignete Punkte der Umwelt zum einen und gewonnenen Informationen aus den Unterhaltungen zum anderen, lassen den Spieler dann im Spielverlauf voranschreiten. Da solche Spiele in der Regel auf martialische Einlagen jedweder Art verzichten und ein potenzielles Ableben der Spielfigur eher genreuntypisch ist, zeichnen sich Point-and-Click-Adventure durch ein relativ gemächliches Spieltempo aus. Stattdessen nehmen sich die Spieldesigner viel Zeit für die narrativen Elemente der Rahmenhandlung und zur Schaffung einer atmosphärischen Dichte.

2.1.2 Der Reiz an Point-and-Click-Adventures

Gut gemachte Point-and-Click-Adventures verfügen für gewöhnlich über eine fesselnde Hintergrundgeschichte und einen hohen Rätselanteil. Darin findet sich der Ursprung des Wortteils *Adventure* in der Genrebezeichnung. In kommerziellen Spielen dieser Gattung finden sich, insbesondere seit Mitte der Neunziger Jahre, oft sehr durchdachte Handlungsverläufe. Aufbau und Qualität können heutzutage nicht selten mit Drehbüchern der Filmindustrie schritthalten. Immer öfter werden für die Verfassung der Handlung gar professionelle Autoren engagiert.

Der Reiz an solchen Spielen liegt für viele Spieler deshalb im intensiven Erlebnis der Storyline. Da Computerspiele im Vergleich zu Filmen über eine längere Spieldauer verfügen, ist es möglich die Charaktere ausführlicher zu exponieren und sich entwickeln zu lassen.

Zudem kann die Geschichte detaillierter und facettenreicher erzählt werden. Es ergibt sich ein ähnlicher Effekt auf den Spieler, wie beim Konsumieren von TV-Serien, deren Konzept auf Fortsetzungsepisoden beruht. Im Laufe der Zeit etabliert sich eine emotionale Bindung zu den Hauptfiguren, die den Spieler dazu motiviert, den weiteren Handlungsverlauf zu verfolgen. Er verspürt den Drang, im Spielgeschehen voranschreiten zu wollen. Die Identifizierung mit den Charakteren im Allgemeinen und dem kontrolliertem Alter Ego im Besonderen, ist dabei stark ausgeprägt, weil der Spieler interaktiv in die Spielwelt eingreift und so zum Mittelpunkt der Handlung wird. Die kontrollierte Spielfigur ist quasi der verlängerte Arm des Spielers, der Vermittler zwischen Spielwelt und Realität, der die befohlenen Aktionen in die Spielwelt überträgt. In Filmen dagegen, besteht das Publikum nur aus Zuschauern. Ihnen wird eine rein passive Beobachterrolle zugeschrieben, aus der sie nicht ausbrechen können. Dem Erfolg des Unterhaltungsmediums Film tut dies zwar keinen Abbruch, dennoch ist festzuhalten, dass das Computerspiel den Menschen auf weiteren, zusätzlichen Ebenen des Erlebens ansprechen kann.

2.2 Eignung von Java im Kontext der Spieleentwicklung

Es gibt immer noch viele kritische Stimmen, wenn es um den Einsatz von Java in der Spieleentwicklung geht. Diese Stimmen bedienen sich einer Reihe von Argumenten, die die Spieletauglichkeit Javas sehr in Zweifel ziehen. Das folgende Kapitel setzt sich mit den Pros und Contras der Thematik eingehend auseinander.

2.2.1 Kontroverse Betrachtung verschiedener Gesichtspunkte

Es ist eine unbestreitbare Tatsache, dass C und C++ seit mehr als zehn Jahren die dominierenden Programmiersprachen in der Videospieleindustrie sind. Im kommerziellen Sektor existieren bisher nur wenige nennenswerte Beispiele für Entwicklungsstudios, die Java als zentrale Technologie zur Umsetzung ihrer Spielevision verwenden. Ausgenommen davon sind Firmen, die kleine Gelegenheitsspiele (casual games) für Mobiltelefone, so genannte Handy-Games, produzieren. Zunehmender Beliebtheit erfreut sich die Java-Plattform allerdings im Open-Source-Bereich, wo sich Gruppen von Hobby-Entwicklern zusammenfinden, um an Spieleprojekten ohne kommerziellen Hintergrund zu arbeiten. Gelegentlich erwächst aus den gemeinsamen Anstrengungen sogar in ein Produkt, dass den Vergleich mit manch professionellem Spiel nicht zu scheuen braucht.

Dennoch kann Java in der Spieleindustrie bis heute nicht so recht Fuß fassen. Viele Vorurteile aus den Anfangstagen der Plattform (Mitte der Neunziger) nagen bis dato am Image der Sprache. Besonders kritisch wurde die Laufzeitgeschwindigkeit gesehen. Java sei zu langsam für ernsthafte Videospiele, war häufig zu vernehmen. Seinerzeit handelte es sich zwar eher um Wahrheiten denn Vorurteile, jedoch sind eine Menge der geäußerten Kritikpunkte mittlerweile nicht mehr zu halten. Im folgenden werden einige der zentralen Aussagen aufgegriffen und argumentativ entkräftet, soweit möglich:

Java sei zu langsam, um damit Spiele zu entwickeln. Als Java 1996 erschien, konnte man diesem Punkt durchaus zustimmen. Die erste Version der Java-Plattform mit dem JDK 1.0 war deutlich langsamer als C/C++. Die Laufzeitgeschwindigkeit stand der von C/C++ in etwa um den Faktor 20-40 nach. Im Laufe der Jahre investierten die Entwickler von Java enorme Arbeit in die Verbesserung ihrer Technologie. Aus Performance-Sicht ist dies besonders deutlich an der Virtual Machine festzumachen. Die Weiterentwicklung der zugrunde liegenden Compilertechnik mit jeder neuen Java-Version, ließ die Geschwindigkeit, mit der Programmcode auf der Virtual Machine ausgeführt wurde, merklich ansteigen. Einen wesentlichen Beitrag dazu leistet das *Just-In-Time*-Verfahren. Es wird vom Java Hotspot-Compiler verwendet. Häufig benötigte Codeabschnitte, so genannte Hotspots, werden zur Laufzeit in nativen Maschinencode übersetzt und weiter optimiert. Dadurch lässt sich die Ausführungsgeschwindigkeit der Hotspots stark beschleunigen [Krüger2002]. Mit der Version J2SE 5.0 ist der Performance-Abstand zu C/C++ deutlich geschrumpft. Bei effizienter Programmierung ist Java-Code nur noch durchschnittlich 1.1 mal langsamer, verglichen mit Code der in C/C++ geschrieben wurde [Davison2005]. Im Internet existieren (kontrovers diskutierte) Benchmarks, die sogar andeuten, dass Java unter bestimmten Umständen schneller sei als C/C++.

Die Ebene, auf der in Java programmiert werde, sei zu abstrakt. Das Prinzip der Objektorientierung ist in Java tief verwurzelt. Eben dieser objektorientierte Ansatz sorge dafür, dass der Programmcode stark aufgebläht erscheinen könne. Die Kritik in diesem Argument ist nachvollziehbar. Wendet man den objektorientierten Stil konsequent an, dann gelangt man höchstwahrscheinlich zu einer Vielzahl von Klassen und Methoden, die dem Konzept des Information Hiding bzw. der Datenkapselung geschuldet sind. Zum Beispiel wird der Zugriff auf Instanzvariablen einer Klasse normalerweise über `get()`- und `set()`-Methoden geregelt, die extra für diesen Zweck eingeführt werden. Es ließe sich jedoch einrichten, die Instanzvariablen per Direktzugriff anzusprechen, ohne den Umweg über eine Methode gehen zu müssen. Dadurch könnten zwar eine Vielzahl von Methoden eingespart werden, es widerspräche aber dem objektorientierten Ansatz und dem gängigem Java-Programmierstil. Gelegentlich wird ein Performance-Nachteil, durch den indirekten Zugriff proklamiert, doch dürfte der kaum ins Gewicht fallen, da die modernen Java-Compiler im übersetzen Code die indirekte Zugriffsmethode in einen Direktzugriff transformieren. Insgesamt ist es aber schon möglich, dass die Performanz der Anwendung durch den aufgeblähten Code etwas herabgesetzt wird. Dem gegenüber stehen allerdings die ungeheuren Vorteile der objektorientierten Programmierung. Man erhält einen hohen Grad an Wiederverwendbarkeit von Softwaremodulen. Externe und eigene Klassenbibliotheken lassen sich leicht in das Projekt integrieren. Der Code wird deutlich wartbarer und weniger anfällig für Fehler. Nicht zuletzt profitieren speziell große Projekte von OOD-Werkzeugen, wie UML, so dass sie leichter zu verwalten sind. Alle genannten Punkte führen zu einer höheren Robustheit des Produkts, geringeren Entwicklungszeiten und somit niedrigeren Projektkosten. Dies sind maßgebliche Faktoren für den marktwirtschaftlichen Erfolg eines Unternehmens.

Java sei nicht hardwarenah genug für die Spielentwicklung. Es ist wahr, dass einige hardwarenahe Operationen, die beispielsweise aus C und C++ bekannt sind, in Java nicht durchgeführt werden können. Gewisse Techniken zur Steigerung der Performanz bleiben dem Java-Programmierer also vorenthalten. C++ gestattet zum Beispiel die Einbettung von Assembler-Codeabschnitten (inline assembly[1]). Da in Assembler verfasster Quelltext seinem Wesen nach deutlich hardwarefokussierter gestaltet werden kann, als kompilierter hochsprachlicher Programmcode, bietet sich ein enormes Optimierungspotenzial. Gerne wird auch auf die direkte Speicherverwaltung verwiesen, die es dem C/C++-Programmierer unter anderem erlaubt, selbst zu entscheiden, wann nicht mehr benötigte Speicherressourcen wieder freizugegeben sind. Die Java-Plattform nimmt dem Programmierer diese Möglichkeit aus der Hand und setzt stattdessen auf das Konzept der Garbage-Collection. Der Garbage-Collector löscht Objekte, die nicht mehr referenziert werden, automatisch aus dem Hauptspeicher und gibt deren Ressourcen wieder frei. Problematisch an diesem Verfahren ist für die Kritiker der Sprache Java, dass man kaum Einfluss darauf nehmen kann, zu welchem Zeitpunkt der Aufräumprozess startet. Unter ungünstigen Umständen kann es vorkommen, dass ein Spiel sekundenlang einfriert und erst nach Abschluss des Garbage-Collection-Prozesses wieder auftaut.

Anderseits kann man es als Vorteil betrachten, dass Java viele Dinge automatisiert behandelt. Besonders die direkte Speicherverwaltung gilt in C/C++ als klassische

1Inline Assembly - http://www.it-academy.cc/article/889/InlineAssembler(x86)+in+C+C++.html (Stand: 19.08.2007)

Fehlerquelle. Überzeugte Java-Entwickler betrachten die Automatisierungen daher eher als Arbeitserleichterung und weniger als die Entmündigung von Manipulationsinstrumenten.

Betrachtet man den Bereich der relevanten Peripheriegeräte für Videospiele, zeigt sich die geringe Hardwarenähe aber ganz unverhohlen. Java bietet keine unmittelbare Möglichkeit an, Eingabegeräte, wie Joysticks oder Gamepads anzusprechen. Insbesondere für die Entwickler von Flugsimulatoren, Renn- und Sportspielen ist eine Anbindung an diese Geräte jedoch unverzichtbar. Ein Open-Source-Community-Projekt namens JInput[2] verschafft hier mittlerweile Linderung. Es handelt sich dabei um ein vollständig in Java entwickeltes API, das Plugins aus nativem Code für das jeweilige Zielbetriebssystem verwendet und dem Programmierer auf diesem Weg den Gerätezugriff ermöglicht.

Die Installation und Ausführung von Java-Programmen sei zu kompliziert. Erstellt man ein Softwareprojekt in C/C++, dann generiert der Compiler aus den Quelldateien eine ausführbare .exe-Datei (Windows-Executable). Durch einen einfachen Mausklick auf diese Datei im Windows-Betriebssystem, lässt sich die Software dann starten. Viele Programme werden mittlerweile mit komfortablen Installationsroutinen ausgeliefert. Der Benutzter klickt sich dabei durch eine Sequenz von Dialog-Fenstern, in denen er Zielordner und gegebenenfalls weitere Installationsoptionen angeben kann. Diese Form der Installation ist weit verbreitet und besonders in der Windows-Welt sehr etabliert.

Möchte man hingegen eine Java-Applikation ausführen, stellen sich dem Anwender einige Hürden in den Weg, die es zu überwinden gilt. Besonders ungeübte Anwender sehen sich mit fast unlösbaren Aufgaben konfrontiert. Zuallererst muss die Existenz einer passenden Java-Laufzeitumgebung (JRE) auf dem Zielsystem gewährleistet sein, weil Java-Programme ohne Virtual Machine nicht lauffähig sind. Heutzutage wird auf vielen Computern zwar eine JRE vorinstalliert mitgeliefert, dennoch kann nicht unbedingt davon ausgegangen werden, dass eine JRE initial zur Verfügung steht. Ist keine JRE vorhanden, muss sie heruntergeladen und installiert werden. Der Download ist immerhin ca. 15MB groß. Häufig müssen zudem noch Klassenbibliotheken von Drittanbietern eingebunden werden (z.B. JOGL). Der ursprüngliche Weg, eine Java-Anwendung zu starten, bedient sich Konsolenkommandos. Befehle, wie `java GameMain` oder `java -jar <mygame>.jar` müssen für diesen Zweck in die Konsole bzw. Eingabeaufforderung eingegeben und bestätigt werden. Das ist für die meisten Computernutzer zu kompliziert. Viele Anwender wissen ohnehin nicht einmal, was mit der Konsole oder Eingabeaufforderung überhaupt gemeint sein könnte.

Um mit einem Videospiel möglichst viele User erreichen zu können, muss der Installations- und Ausführungsprozess komfortabel sein. Die Aussagen des vorherigen Absatzes lassen Java an dieser Forderung scheinbar scheitern. Aber auch hier gibt es Lösungen, die die Situation erleichtern.

In Äquivalenz zu den ausführbaren .exe-Dateien, stehen in Java ausführbare .jar-Dateien. Anstelle der Konsolenkommandos tritt ein einfacher Doppelklick auf eine .jar-Datei, um die darin enthaltene Applikation zu starten. Da das Jar-Dateiformat zugleich ein Pack- und Komprimierungsformat ist, kann es vorkommen, dass Extraktionsprogramme, wie z.B. WinRar, die Java Runtime Environment als Standardprogramm zum Ausführen von .jar-Dateien verdrängen. Infolgedessen ginge die einfache Doppelklickfunktion verloren. Durch die Bereitstellung von Batch-Dateien (Windows) bzw. Shell-Skripten (Linux, MacOS), die die

2 Java Input API Project (JInput) – https://www.jinput.dev.java.net

nötigen Konsolenbefehle kapseln und sich ebenfalls durch Klicken ausführen lassen, könnte dieser Unzulänglichkeit wirkungsvoll entgegengetreten werden. Aufgrund der Erwartung des Nutzers, eine .exe-Datei vorzufinden, sollten die Batch-Dateien selbsterklärend benannt werden (z.B. start.bat).

Eine weitere Alternative besteht in der Verwendung von Software, die für Java-Anwendungen eine Installationsroutine im etablierten Stil der nativen Windows-Anwendungen generiert. Ein Vertreter aus dem kommerziellen Sektor ist *install4j.*[3] Mit dieser Software ist es möglich, eine JRE beizulegen, so dass die Problematik der potenziell fehlenden Runtime Environment ausgehebelt wird.

Mit der *Webstart*-Technologie[4] hat die Firma Sun eine äußerst komfortable Distributionsmethode für Java-Applikationen geschaffen. Durch Webstart können Programme in Netzwerken und über das Internet zugänglich gemacht und ausgeführt werden, sofern eine JRE zur Verfügung steht. Der Software-Anbieter muss die Dateien seiner Anwendung für die Webstart-Technologie vorbereiten und kann sie, zusammen mit einer speziellen Metadatei, auf einem Webserver ablegen. Ruft ein Anwender die URL auf, die mit der Metadatei verknüpft ist, lädt das Webstart-System alle nötigen Programmressourcen auf den Client-Rechner herunter und führt anschließend die Anwendung aus. Auf diese Weise können Java-Programme durch den unkomplizierten Klick auf einen Link ausgeführt werden. Ein besonderes Feature ist der automatische Update-Mechanismus. Vor der Programmausführung wird stets überprüft, ob aktuellere Programmressourcen bereitgestellt wurden. Trifft dies zu, wird die Applikation auf dem Client-Rechner entsprechend aktualisiert.

Java steht auf Spielkonsolen nicht zur Verfügung. Eine der meist gepriesenen Eigenschaften Javas ist ihre Plattformunabhängigkeit. Die viel zitierte Aussage „Write once, run everywhere.“ bringt dies treffend auf den Punkt. Wurde ein Programm in Java geschrieben, lässt es sich anschließend auf jedem Zielsystem ausführen, für das eine Implementierung der Virtual Machine vorliegt. Dies erweitert die potenzielle Käuferschicht, ohne kostspielige Portierungen für verschiedene Plattformen vornehmen zu müssen. Virtual Machines existieren zum Beispiel für die diversen Windows-Versionen, Unix/Linux-Distributionen und Apple's Mac-Systeme, nicht jedoch für die gängigen Spielkonsolen. Hier befindet sich Javas Achillesferse in Bezug auf die Spieleindustrie. Entpuppten sich viele der vorangegangenen Kritikpunkte häufig als überholte Vorurteile, so muss man Java in diesem Fall offenkundige Unzulänglichkeiten bescheinigen. Der lukrative Konsolenmarkt ist für Spiele auf Java-Basis gegenwärtig verschlossen! Das verheißungsvolle Prinzip der Plattformunabhängigkeit wird ad absurdum geführt. Sollte sich in Zukunft daran nichts ändern, so drohen alle Bemühungen, Java ernsthaft als Sprache in der Spieleindustrie zu etablieren, schon aus marktstrategischen Überlegungen heraus zu scheitern.

Das Videospiele-Geschäft ist zum milliardenschweren Industriezweig herangewachsen. Es existiert ein Markt von mehreren hundert Millionen Viedeospielern. Davon lassen sich aber nur 10-20% dem PC, also dem für Java relevanten Bereich, zuordnen. Der Rest verteilt sich auf das Triumvirat der großen Konsolenhersteller Sony (Playstation 1-3), Microsoft (XBOX, XBOX 360) und Nintendo (Gamecube, Wii) [Davison2005]. Da auf keiner der Konsolen bisher eine Java Virtual Machine verfügbar ist, wird ersichtlich, warum Java unter professionellen

3 Installer Builder *install4j* - http://www.ej-technologies.com/products/install4j/overview.html
4 Java Webstart - http://java.sun.com/products/javawebstart/

Spieleentwicklern auch weiterhin mit schwerwiegenden Akzeptanzproblemen zu kämpfen haben wird.

Neben dem PC macht Java eher durch die Präsenz auf einer anderen Plattform auf sich aufmerksam: Nahezu jedes aktuelle Mobiltelefon ist mit einer Java Virtual Machine ausgestattet, so dass sich auf dem Handymarkt eine marktbeherrschende Stellung ergibt. Mittlerweile ist eine Fülle so genannter Handy-Games erhältlich, die kommerziell angeboten werden. Da es sich hier, bedingt durch die limitierte Hardware, fast ausschließlich um wenig komplexe Gelegenheitsspiele (Casual Games) handelt, die technologisch gesehen wenig mehr bieten, als Spiele, die vor 20 Jahren auf dem C-64 veröffentlicht wurden, trägt dies nicht gerade dazu bei, Javas Image als ernst zu nehmende Sprache für komplexe, technisch moderne Spiele aufzuwerten.

Die technologische Infrastruktur ist weniger ausgereift. Durch die Stellung als de facto-Standard, stehen für C/C++ sehr viele spielerelevante Tools und Bibliotheken für die Entwicklung zur Verfügung. Viele der heute in der Industrie bedeutenden 3D-Engines, wie die Doom-3- oder die Unreal-3-Engine, sind für diese Sprachen entwickelt worden. Daraus folgt eine unbestreitbare Technologieführerschaft im Spielesektor, die die Java-Plattform für die Entwicklergemeinde weniger attraktiv erscheinen lässt. Auf Seiten von C/C++ existieren mit OpenGL und Microsofts DirectX bereits zwei weit verbreitete APIs für das hardwarebeschleunigte Rendern von 2D- und 3D-Welten, die sich in technologischer Sicht kaum nachstehen. Java unterstützt bisher nur das plattformunabhängige OpenGL. Mit JOGL[5] (Java Bindings for OpenGL) und LWJGL[6] (Lightweight Java Game Library) werden mindestens zwei bedeutende Bindings für den Zugriff auf die OpenGL-Schnittstelle angeboten. DirectX kann, mit Blick auf dessen proprietäre und plattformabhängige Natur, nicht verwendet werden.

Da Spiele seit jeher besondere Aufmerksamkeit durch eine opulente Grafik erregen konnten, ist es wichtig, dass moderne Grafikkarten-Features, wie Vertex- und Fragment-Shader, auch für Java-Entwickler zur Verfügung stehen. Weil die zuvor genannten Bindings OpenGL bis zur aktuellen Version 2.0 unterstützen (Stand: Juni 2007) und auch die Shader-Hochsprache GLSL verwendet werden kann, können Java-Spiele ihren Konkurrenten in Sachen Grafik durchaus ebenbürtig sein.

Neben den OpenGL-Bindings, die zu einer konkurrenzfähigen Grafik befähigen, stehen mittlerweile weitere APIs und Bindings zur Verfügung, die dazu beitragen können, Spielerlebnisse intensiver und wirklichkeitsgetreuer werden zu lassen. Das in dieser Arbeit bereits erwähnte JInput-API zur Ansteuerung von Gamepads und Joysticks ist ein Beispiel hierfür. Nahezu ebenso wichtig, wie die visuelle Präsentation, ist eine atmosphärische Klangkulisse. Sie kann die Wahrnehmung des Spielers stark beeinflussen. Eine fröhlich beschwingte Hintergrundmusik oder gruselige Geräusch-Einspielungen prägen den emotionalen Grundton einer Szene. Der Game-Designer kann dadurch auf ein imposantes Werkzeug für die Gestaltung seiner Spielwelt zurückgreifen. Um den Eindruck einer dreidimensionalen Umgebung zu verstärken, liegt es nahe, auch die Geräusche im Spiel räumlich erklingen zu lassen. Zu diesem Zweck hat man ein API namens OpenAL für C/C++ entwickelt, das positionsbezogenen 3D-Sound ermöglicht. Es kann als Audio-Equivalent zum

5 JOGL-Projekt – https://jogl.dev.java.net
6 LWJGL-Projekt – http://lwjgl.org

grafischen OpenGL verstanden werden. Über das Binding JOAL[7] (Java Bindings for OpenAL) ist es nun auch unter Java nutzbar.

Physikalisch korrektes Fallen und Abprallen von Körpern oder Gegenständen unterstützt eindrucksvoll die Glaubwürdigkeit der Spielumgebung. Die realitätsnahe Umsetzung der Effekte von Gravitation und Impuls eröffnen zudem ganz neue Wege im Rätseldesign. Neben kommerziellen Physik-APIs gibt es auch kostenlose Bibliotheken, die zu diesem Zweck entwickelt wurden. Beispielhaft sei auf die quelloffene Physik-Engine ODE (Open Dynamics Engine) verwiesen. Sie dient zur Simulation der Dynamik bewegter Objekte in der Spielumgebung und steht über das Binding ODEJava[8] auch Java-Entwicklern zur Verfügung.

Eingangs wurde ein Manko an konkurrenzfähigen 3D-Engines auf Java-Basis erwähnt. In der Breite ist dies zutreffend, doch umso stärker rücken die wenigen Vorreiter in den Fokus der öffentlichen Wahrnehmung. Die wohl bekannteste 3D-Engine in der Java-Games-Community ist die JMonkey Engine[9]. Da es sich um ein OpenSource-Projekt auf Basis von LWJGL handelt, kann sie kostenfrei für eigene Spiele verwendet werden. Die Firma Agency9 bietet gegen eine Lizenzgebühr ihre AgentFX-Engine[10] an. Für Entwickler ohne kommerzielles Interesse, ist aber auch eine kostenlose Community-Edition erhältlich. Im Gegensatz zur JMonkey Engine setzen die Entwickler von Agency9 aber auf JOGL als OpenGL-Binding. Beide Engines benutzen Scenegraph-Strukturen für das Rendern, bieten moderne Grafik-Features und lassen sich durch Shader-Programmierung beeinflussen.

Haben C/C++-Spiele-Entwickler noch einen deutlichen Vorteil durch die über Jahre gewachsene breite technologische Infrastruktur, so kann Java mit einigen Sprachfeatures überzeugen, die gegenwärtig zunehmende Relevanz erfahren: Java wurde von Haus aus mit ausgeprägten Client/Server- und Netzwerk-Fähigkeiten ausgestattet. Dieser Umstand prädestiniert Java förmlich für Multiplayer- und Online-Games, deren Markt gegenwärtig ein außergewöhnliches Wachstum verzeichnet.

Nicht zuletzt profitiert Java sehr von der einfachen Einbindung von fremden Programm-Bibliotheken, so genannter Third-Party-Librarys, in das eigene Projekt. Diese Module können enorme Arbeitsersparnisse bewirken und folglich den Entwicklungsprozess signifikant produktiver machen.

2.2.2 Persönliche Meinung

Viele der besonders gravierenden Einwände gegen Java als Programmiersprache für Video-Spiele, haben sich mit fortschreitender Entwicklung der Plattform als unhaltbar erwiesen. Die Kritik an der geringen Performanz ist in diesem Kontext besonders hervorzuheben. Jedoch sieht sich die Spieleindustrie bisher nicht genötigt auf die Trendtechnologie Java umzusatteln. Man macht es sich auf einem großen Erfahrungsschatz in der C/C++-Spieleprogrammierung bequem. Schließlich kann man auf einen reichhaltigen Fundus erprobter und bewährter Werkzeuge zurückgreifen. Eine Fülle von verfügbaren Hochleistungs-Game-Engines erschwert die Aufgabe des Status Quo noch zusätzlich, würde eine Neuorientierung zu Java hin doch zeitraubende und kostspielige Reimplementierungen nach sich ziehen.

7 JOAL-Projekt - https://joal.dev.java.net
8 ODEJava-Projekt - http://odejava.org
9 JMonkey Engine – http://www.jmonkeyengine.com
10 AgentFX-Engine – http://www.agency9.se/products/agentfx

Auf lange Sicht könnte sich der Umstieg jedoch lohnen. Hat man sich erstmal ein Portfolio an Entwicklungswerkzeugen geschaffen, schlagen die Vorteile, die Java mit sich bringt (Wartbarkeit, kürzere Entwicklunsgzeit usw.), voll zu Buche. Die große Chance zur Verbreitung und vermutlich oft auch Bekanntmachung Javas liegt im Nachwuchs. In den meisten Universitäten und Fachhochschulen wird Java mittlerweile als primäre Programmiersprache gelehrt. Heutige Hochschulabsolventen beherrschen die Sprache also in der Regel gut und wissen insbesondere um deren Vorteile. Vielleicht schaffen es diese Absolventen in absehbarer Zeit, den Projektleitern und Finanziers die Option Java näher zu bringen und bestehende Vorurteile abzubauen.

Nicht nur aus betriebswirtschaftlichen Erwägungen heraus, sondern auch mit Blick auf den Entwicklungskomfort, wäre die Verwendung von Java insgesamt zu empfehlen. Die gegenwärtige Beschränkung auf PC-Spiele, einhergehend mit dem beschriebenen Ausschluss vom Konsolenmarkt, überschattet die Empfehlung aber etwas. Es ist zu hoffen, dass diese Hürde in Zukunft überwunden werden kann.

2.3 Mathematisches Basiswissen für 3D-Spiele

Mathematik ist der Schlüssel zu interaktiven Spielwelten. Dabei ist es völlig gleichgültig, ob es sich um flache zweidimensionale oder räumliche Umgebungen handelt. Geometrische Kalkulationen hauchen Video-Spielen den Odem des Lebens ein. Sie sind die Grundlage für viele Aspekte der künstlichen Szenerie und finden sowohl in der Darstellung als auch in der Spiellogik Anwendung. Zum Beispiel bei Objektbewegungen, Kollisionserkennungen, Distanzermittlungen, Winkel- sowie Beleuchtungs- und Schattenberechnungen spielen sie die zentrale Rolle.

Da diese Masterarbeit unter anderem die Entwicklung einer 3D-Game-Engine beinhaltet, ist es nahe liegend, einige relevante Facetten der 3D-Geometrie vorzustellen. Im wesentlichen werden Operationen auf den mathematischen Strukturen *Vektor* und *Matrix* behandelt, die im Kontext einer Game-Engine als essenziell zu bezeichnen sind.

2.3.1 Fundamentale Vektor-Operationen im 3D-Raum

Vektoren sind gewissermaßen die fundamentalen Komponenten von 3D-Welten. Ihr Anwendungsgebiet ist vielfältig. So finden sie beispielsweise Anwendung als Eckpunkte (Vertizes) der Dreiecke aus denen 3D-Modelle für gewöhnlich bestehen. Aber auch Positionen (Positionsvektor) und Richtungen (Richtungsvektor) lassen sich mit ihrer Hilfe beschreiben.

Vektoren können also auch Positionen spezifizieren, das heißt, insbesondere Punkte im Raum eines kartesischen Koordinatensystems. Punkte und Vektoren sind zwar konzeptionell verschieden, mathematisch jedoch verhalten sie sich identisch [Dunn2002]. Deshalb werden beide Begriffe fortan synonym verwendet.

Die folgenden Subkapitel beschäftigen sich mit einigen häufig benötigten Vektor-Operationen und deren Interpretation im Rahmen einer dreidimensionalen Spielwelt. Die drei Komponenten eines 3D-Vektors werden fortan mit *x*, *y* und *z* bezeichnet. Bei komplexeren Operationen, wird das Verständnis anschaulich durch Abbildungen erleichtert.

2.3.1.1 Länge eines Vektors

Die Länge $\|\vec{v}\|$ eines Vektors ist ein positiver skalarer Wert. Er gibt die Länge der Wegstrecke ausgehend vom Ursprung des Vektors bis zum spezifizierten Endpunkt an. Die Länge eines Vektors lässt sich mit einer Formel berechnen, die auf das pythagoräische $a^2+b^2=c^2$ zurückzuführen ist. Sie lautet: $\|\vec{v}\| = \sqrt{\sum_{i=1}^{3} v_i^2} = \sqrt{v_1^2+v_2^2+v_3^2} \Rightarrow \sqrt{x^2+y^2+z^2}$

2.3.1.2 Negation eines Vektors

Vektoren können negiert werden. Manchmal wird diese Operation auch als Invertieren bezeichnet. Die Negation liefert einen Vektor, der zwar die gleiche Länge hat wie der Originalvektor, jedoch in die genau entgegengesetzte Richtung zeigt. Diese Operation kann also genutzt werden, um einen Vektor in die Gegenrichtung umzukehren. Mathematisch geschieht dies durch die Multiplikation der drei Vektorkomponenten mit -1, beziehungsweise

einer einfachen Umkehrung der Komponenten-Vorzeichen. Die Negation eines Vektors $\vec{a}$ kann schematisch wie folgt beschrieben werden.

$$\vec{a} \overset{Negation}{\Rightarrow} -\vec{a} = -\begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} = \begin{pmatrix} -a_x \\ -a_y \\ -a_z \end{pmatrix}$$

2.3.1.3 Addition von Vektoren

Durch die Addition von Vektoren entsteht ein neuer Vektor, der sich aus den Einzelvektoren zusammensetzt. Mathematisch werden zwei Vektoren addiert, in dem ihre Komponenten paarweise summiert werden. Folgende Formel verdeutlicht dies:

$$\vec{a} + \vec{b} = \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} + \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} = \begin{pmatrix} a_x + b_x \\ a_y + b_y \\ a_z + b_z \end{pmatrix}$$

2.3.1.4 Subtraktion von Vektoren

Nicht nur die Addition lässt sich auf Vektoren durchführen. Sie lassen sich auch leicht subtrahieren. Diese Operation erfolgt durch paarweise Subtraktion der Vektorkomponenten.

$$\vec{a} - \vec{b} = \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} - \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} = \begin{pmatrix} a_x - b_x \\ a_y - b_y \\ a_z - b_z \end{pmatrix}$$

2.3.1.5 Distanz zwischen zwei Vektoren

Die Distanz zwischen zwei Vektoren ist eine oft herangezogene skalare Größe in Video-Spielen. Da Vektoren und Punkte sich mathematisch äquivalent verhalten, gibt die Distanz zwischen zwei Vektoren, den Abstand eines Punktes zu einem anderen an. Punkte repräsentieren in Spielen Positionen. Es kann also die Distanz von einem Ausgangspunkt zu einem Zielpunkt errechnet werden. Um den Abstand zwischen zwei Vektoren zu kalkulieren, muss eingangs ein Vektor $\vec{d}$ berechnet werden, der die mathematische Verschiebung von Ausgangsvektor $\vec{a}$ zu Zielvektor $\vec{b}$ widerspiegelt. Das geschieht durch Subtraktion des Ausgangsvektors vom Zielvektor. Berechnet man nun die Länge des resultierenden Vektors, wie in Kapitel 2.3.1.1 vorgestellt, erhält man das Skalar, welches der gesuchten Distanz entspricht. Folgende Formel kombiniert beide Schritte, so dass sie direkt das korrekte Ergebnis liefert.

$$Distanz(\vec{a}, \vec{b}) = \|\vec{d}\| = \|\vec{b} - \vec{a}\| = \sqrt{(b_x - a_x)^2 + (b_y - a_y)^2 + (b_z - a_z)^2}$$

Beispiele für konkrete Anwendungsfälle der Distanz-Funktion in Spielwelten sind die Ermittlung des Abstands einer Spielfigur zu einem designierten Wegpunkt, oder die Berechnung der entfernungsbasierten Wiedergabelautstärke von positionellen 3D-Sound.

2.3.1.6 Punktprodukt zweier Vektoren

Das Ergebnis des Punktprodukts zweier Vektoren ist ein Skalar, dessen Interpretation geometrische Aussagen über die Vektoren gestattet. Das Punktprodukt lässt sich durch die Summe der Produkte einer paarweisen Multiplikation der Vektorkomponenten bilden. Nachstehende Formel fasst diesen Satz mathematisch zusammen.

$$\vec{a}\cdot\vec{b}=\sum_{i=1}^{3} a_i b_i = a_1 b_1 + a_2 b_2 + a_3 b_3 \;\Rightarrow a_x b_x + a_y b_y + a_z b_z$$

Aus dem skalaren Wert des Punktprodukts lassen sich Schlussfolgerungen bezüglich des Winkels *θ* zwischen den beteiligten Vektoren anstellen. Dabei können drei Fälle unterschieden werden:

- Wenn $\vec{a}\cdot\vec{b}=0$, dann ist *θ* = 90°. Die beiden Vektoren sind also rechtwinklig zueinander .
- Wenn $\vec{a}\cdot\vec{b}>0$, dann ist *θ* < 90°. Der Winkel zwischen den Vektoren ist also spitz.
- Wenn $\vec{a}\cdot\vec{b}<0$, dann ist *θ* > 90°. Der Winkel zwischen den Vektoren ist also stumpf.

Wie man die Informationen, die aus dem Punktprodukt gewonnen werden können, sinnvoll anwendet, hängt von der Interpretation im jeweiligen Szenario ab. Beispielsweise kann geprüft werden, ob ein Punkt innerhalb des Sichtbereichs der Kamera liegt, wenn die Kamera ein Sichtfeld von 90° besitzt. Dazu müsste der Vektor zwischen der Kamera-Position und dem zu prüfenden Punkt berechnet werden. Anschließend kalkuliert man das Punktprodukt zwischen Kamera-Position und dem gerade ermittelten Vektor. Wenn das Ergebnis kleiner als 0 ist, dann fällt der Sichttest für den Punkt negativ aus [Stahler2004].

2.3.1.7 Kreuzprodukt zweier Vektoren

Das Kreuzprodukt ist nützlich, um einen Vektor zu ermitteln, der im rechten Winkel zu zwei anderen Vektoren steht. Genauer gesagt, ist der Ergebnisvektor rechtwinklig zur Ebene, die von beiden Eingabevektoren gemeinsam aufgespannt wird. Abbildung 1 visualisiert das Ergebnis des Kreuzprodukts zweier Vektoren $\vec{a}$ und $\vec{b}$.

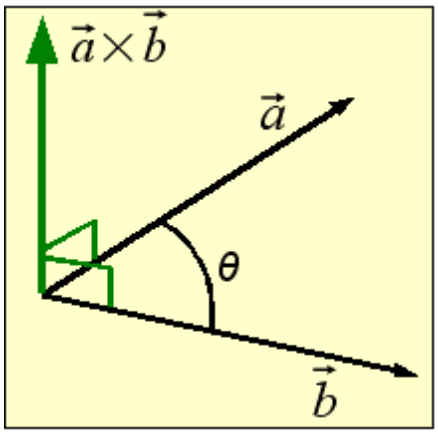


Abbildung 1: Grafische Interpretation des Kreuzprodukts

Das Kreuzprodukt für 3D-Vektoren lässt sich nach folgendem Schema aufstellen:

$$\vec{a}\times\vec{b}=\begin{pmatrix}a_x\\a_y\\a_z\end{pmatrix}\times\begin{pmatrix}b_x\\b_y\\b_z\end{pmatrix}=\begin{pmatrix}a_yb_z-a_zb_y\\a_zb_x-a_xb_z\\a_xb_y-a_yb_x\end{pmatrix}$$

Das Kreuzprodukt ist besonders wichtig für die Berechnung der Normalen eines Dreiecks, wie in Unterkapitel 2.3.1.9 zu lesen ist.

2.3.1.8 Normalisierung eines Vektors

In manchen Situationen kommt es vor, dass nicht die Länge eines Vektors, sondern die Richtung, in die er zeigt bedeutsam ist. In diesen Fällen betrachtet man für gewöhnlich den Einheitsvektor eines Vektors. Einheitsvektoren sind Vektoren der Länge 1. Um einen beliebigen Vektor in einen Einheitsvektor zu transformieren, auch Normalisierung genannt, teilt man jede seiner Komponenten durch die Länge des Vektors. Der normalisierte Vektor weist zwar in die selbe Richtung wie der Originalvektor, hat jedoch die Länge 1. Zu beachten ist, dass der Originalvektor kein Nullvektor sein darf, weil Divisionen durch 0 nicht definiert sind. Mathematisch korrekt, lässt sich die Normalisierung so ausdrücken:

$$\vec{v}_{norm}=\frac{\vec{v}}{\|\vec{v}\|},\ \vec{v}\neq\mathbf{0}$$

Auf diese Weise berechnete Vektoren werden auch als Normalen bezeichnet. Das nächste Unterkapitel befasst sich mit der Berechnung von Normalen für Dreiecksflächen.

2.3.1.9 Berechnung der Normalen eines Dreiecks

Für das Beleuchtungsmodell einer 3D-Szene sind Normalen besonders einflussreiche Faktoren. So können den Polygonen in OpenGL (meist Dreiecke) Normalen zugewiesen werden, die zur Berechnung des Lichteinfalls auf deren Eckpunkte herangezogen werden.

Für korrekte Beleuchtungen von 3D-Modellen werden in der Regel Normalen verwendet, die rechtwinklig zu ihren Dreiecksflächen stehen. Aus Kapitel 2.3.1.7 wissen wir, das zur Berechnung rechtwinkliger Vektoren das Kreuzprodukt genutzt werden kann. Man ermittelt

durch Subtraktion zwei der drei Vektoren, die die Dreiecksebene aufspannen und berechnet das Kreuzprodukt auf ihnen. Die Normalisierung des Ergebnisvektors gemäß Kapitel 2.3.1.8, ergibt den gesuchten Normalenvektor. Die Schritte werden nun mathematisch verdeutlicht:

1. Berechnen von zwei Spannvektoren $\vec{s}_1$ und $\vec{s}_2$ der Dreiecksfläche, die durch die Punkte *a*, *b* und *c* definiert ist: $\vec{s}_1 = b - a \ \wedge \ \vec{s}_2 = c - a$
2. Ermitteln des Kreuzprodukts aus den Spannvektoren, um einen zur Ebene rechtwinkligen Vektor $\vec{n}$ zu erhalten: $\vec{n} = \vec{s}_1 \times \vec{s}_2$
3. Normalisierung des Vektors $\vec{n}$: $\vec{n}_{norm} = \frac{\vec{n}}{\|\vec{n}\|}, \vec{n} \neq \mathbf{0}$

Abbildung 2 veranschaulicht die Situation grafisch.

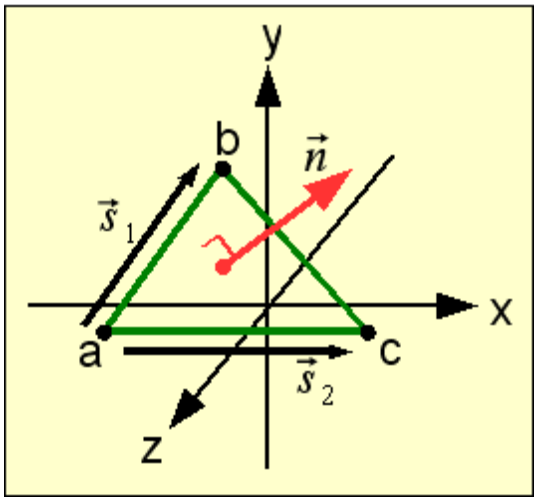


Abbildung 2: Ergebnis der Berechnung des Normalenvektors eines Dreiecks

2.3.2 Matrizen als Grundlage für 3D-Transformationen

Matrizen werden in 3D-Engines verwendet, um Vektoren zu transformieren. Durch die Multiplikation eines Vektors mit einer anwendungsfallbezogenen Matrix, können die Vektorkomponenten entsprechend manipuliert werden. Auch die Grafik-Bibliothek OpenGL verwendet Matrizen, um Transformationen von Vertizes darzustellen.

Die gängigsten Transformations-Operationen sind das Rotieren, Skalieren und Translozieren. Bei der Rotation werden Punkte um die X-, Y- oder Z-Achse gedreht. Skalierungen bewirken Größenveränderungen und Translationen verändern die Position eines Punktes im Raum.

Für jede dieser Transformationen existiert eine geeignete Matrix-Struktur, die bei Multiplikation mit einem Vektor, die gezielte Veränderungen seiner Koordinaten bewirkt. Als Transformations-Matrizen werden in der Regel quadratische 4x4 Matrizen konstruiert. Damit 3D-Vektoren mit 4x4 Matrizen multipliziert werden können, werden Sie um eine vierte Komponente *w* ergänzt, die für die Transfomations-Szenarien jedoch immer den Wert *1* annimmt.

Nachstehende Unterkapitel stellen die einzelnen Transformations-Operationen mitsamt ihren spezifischen Matrizen vor.

2.3.2.1 Translation

Die Translation ist eine Operation, die herangezogen wird, um die Position eines Punktes zu manipulieren. Sie kann linear in allen drei Achsrichtungen des Koordinatensystems erfolgen. Durch die Multiplikation eines Vektors, mit der nun präsentierten Translations-Matrix, wird die Koordinaten-Veränderung umgesetzt.

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & \delta x \\ 0 & 1 & 0 & \delta y \\ 0 & 0 & 1 & \delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Die Translations-Matrix wird mit Hilfe der Werte *δx*, *δy* und *δz,* die die Größe der Positionsveränderung angeben, initialisiert. Die Multiplikation mit dem Eingabevektor resultiert im translozierten Ergebnisvektor.

2.3.2.2 Skalierung

Die Skalierung ist eine Operation, die es erlaubt 3D-Objekte in der Größe (räumliche Ausdehnung) zu variieren. Dazu müssten die einzelnen Vertizes des Objekts mit der folgenden Skalierungs-Matrix multipliziert werden.

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \delta x & 0 & 0 & 0 \\ 0 & \delta y & 0 & 0 \\ 0 & 0 & \delta z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Die Skalierungs-Matrix wird mit Hilfe der Werte *δx*, *δy* und *δz,* die die Größenänderungen auf der X-, Y- und Z-Achse angeben, initialisiert. Die Multiplikation mit dem Eingabevektor resultiert im skalierten Ergebnisvektor.

2.3.2.3 Rotation

3D-Objekte, beziehungsweise deren Vertizes, können mit dem Winkel *θ* um ihre Achsen rotiert werden. Für jede der drei Achsen wird dabei eine unterschiedliche Struktur der Rotations-Matrix verwendet. Im Folgenden werden alle drei Fälle explizit spezifiziert.

Der Winkel *θ* dient als Eingabe zur Konstruktion der jeweiligen Rotations-Matrix. Die Multiplikation mit dem Eingabevektor resultiert im rotierten Ergebnisvektor.

Rotation um die X-Achse

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Rotation um die Y-Achse

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Rotation um die Z-Achse

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Die Ausführungen über die Transformations-Operationen schließen das gesamte Oberkapitel über die projektrelevanten Grundlagen in Sachen 3D-Mathematik ab. Weiterführende Konzepte werden erklärt, sobald sie benötigt werden. Das nächste Oberkapitel stellt wichtige Basistechnologien vor, die für die Umsetzung des Projekts herangezogen werden. Der Fokus liegt jedoch nicht auf einer umfassenden Einführung, sondern es werden vielmehr einzelne Aspekte aus den Technologien hervorgehoben, die einen speziellen Projektbezug aufweisen.

3 Relevante Basistechnologien

Die nachfolgenden Unterkapitel stellen dem Leser in prägnanter Form die wesentlichen Technologien vor, die für die Umsetzung des Projekts herangezogen werden.

3.1 Stand der Technik

Adventure-Spiele haben im Laufe ihrer Geschichte eine aufregende technologische Wandlung durchgemacht. In den Anfangstagen begannen sie als rein textorientierte Spiele, in denen die Handlungsanweisungen des Spielers über Befehlswörter per Tastatur einzugeben waren. Später wurden die Texte auf dem Monitor durch illustrierende Grafiken ergänzt. Einen Umbruch im Steuerungskonzept erlebte das Genre durch die Einführung des Point-and-Click-Konzepts. Nun war es nicht mehr nötig, Text-Kommandos zu tippen, sondern es reichte aus, mit der Maus auf Befehlsbausteine und Hotspots der Bildschirmgrafik zu klicken. Eine neue Generation von Adventures war geboren. Über viele Jahre waren Point-and-Click-Adventures ausschließlich auf zweidimensionale Darstellungen beschränkt. Durch das Aufkommen von 3D-Grafik und den damit möglichen neuen Spielkonzepten, galten sie als technisch überholt und verloren zunehmend an Zuspruch. Schlechte Verkaufszahlen waren die Folge und schreckten Investoren und Entwickler ab, so dass kaum neue Adventures erschienen. Das Genre stürzte in die Krise und wurde des öfteren totgesagt.

Erst in jüngster Zeit kommt es zu einem Wiederaufleben des Adventure-Sektors. Die Anpassung der zugrunde liegenden Technik an die modernen Gegebenheiten wird sicherlich dazu beigetragen haben. Hochauflösende Grafiken, aufwändige Effekte und innovative Konzepte hieven die neueste Generation der Point-and-Click-Adventures langsam zurück in den Massenmarkt.

Im Prinzip lassen sich drei verbreitete Methoden zur Präsentation der Spielgrafik beschreiben, die in heutigen Adventures zur Anwendung kommen und somit den Stand der Technik in diesem Bereich repräsentieren.

1. **Klassisches 2D-System** Alle Grafiken, egal ob Hintergrundbild oder für Spiel-Charaktere, bestehen aus zweidimensionalen Zeichnungen. Durch höhere Auflösungen und Farbtiefen des Bildmaterials, sowie durch aufwändig inszenierte Animationen, entsteht ein Detailreichtum, der den klassischen 2D-Adventures von früher deutlich voraus ist.

2. **2D/3D-Hybrid-System** Eine weit verbreitete Technik unter den modernen Adventures ist die Hybrid-Technik. Sie stellt sich als Kombination aus klassischer 2D-Grafik und den Möglichkeiten der 3D-Technik dar. Bei Spielen, die auf diesem Verfahren beruhen, ist häufig zu beobachten, dass die Hintergrundgrafiken aus vorgerenderten statischen Bildern bestehen, die mit speziellen Modellierungs-Programmen generiert wurden. Vor dieser Kulisse agieren dann animierte 3D-Polygon-Figuren. Auf diese Weise müssen trotz hervorragender grafischer Qualität nur wenige Polygone für eine Szene gezeichnet werden. Die Hardware-Anforderungen für diese Technik fallen deshalb vergleichsweise gering aus.

3. **Vollständige 3D-Grafik** Point-and-Click-Adventures, die dieses Verfahren wählen, vertrauen auf eine vollständige 3D-Umgebung. Als technologisches Fundament können deshalb moderne 3D-Engines, mit all ihren beeindruckenden (Shader-) Effekten, in Anspruch genommen werden. Der spielmechanische Vorteil gegenüber einem Hybrid-System ist die freie Bewegbarkeit der Kamera durch eine dynamische Umgebung. Die gleiche Szene kann mühelos aus verschiedenen Blickwinkeln gezeigt werden. Auf diese Weise lässt sich leicht ein erzählerisch vorteilhafter, filmähnlicher Eindruck herstellen. Selbstredend sind die Anforderungen an die Hardware bei dieser Technik erheblich höher als bei den beiden zuvor genannten Verfahren. Mittlerweile existieren jedoch ein paar (kommerziell erfolgreiche) Spiele, die diese Variante sehr imposant umgesetzt haben.

3.2 OpenGL

Die folgenden Unterkapitel beleuchten einige Aspekte der Basistechnologie OpenGL, die dem Leser einen wissenswerten Eindruck zum besseren Verständnis des Projekts vermitteln sollen.

3.2.1 Was ist OpenGL?

OpenGL ist ein offenes API zur Darstellung von dreidimensionalen Szenen, das auf der Programmiersprache C basiert. Es bietet eine Schnittstelle für den Zugriff auf die Funktionen der Grafikhardware eines Systems an. Ähnlich wie Java, steht OpenGL auf allen bedeutenden Plattformen zur Verfügung. Die Anzahl der Methoden, die den Zugriff auf die Grafikhardware gewährleisten, ist heute mit mehr als 300 zu beziffern [Wright2004].

Ursprünglich wurde OpenGL von der Firma *Silicon Graphics, Inc. (SGI)* entwickelt. Seit 1992 wird die Weiterentwicklung der OpenGL-API jedoch vom *Architecture Review Board (ARB)* gesteuert. Das ARB ist ein Zusammenschluss führender Unternehmen, die sich im Bereich der 3D-Technologien engagieren [Astle2004]. Durch die Kooperation dieser Firmen ist OpenGL zu einem industrieweit anerkannten Standard avanciert. Alle großen Grafikkarten-Hersteller bieten heutzutage eine ausgeprägte Unterstützung der OpenGL-API an.

Die Anwendungsgebiete für OpenGL sind mannigfaltig. Es wird für alle denkbaren Anwendungen eingesetzt, die Komponenten zur dreidimensionalen Darstellung beinhalten. Das sind zum Beispiel CAD-Programme, Prävisualisierungssoftware für Architektur und Baumaßnahmen, medizinische Bildverarbeitung, visuelle Darstellung physikalischer Vorgänge oder insbesondere 3D-Video-Spiele [Wright2004]. Gerade im Spielesektor existiert mit *id Software* (Schöpfer der Doom- und Quake-Serie) ein Unternehmen, das immer wieder neu vor Augen führt, welche grafische Qualität OpenGL-Anwendungen erreichen können.

3.2.2 Die OpenGL-Zustandsmaschine

Der Prozess des Zeichnens einer Szene (auch *Rendern* genannt) wird in OpenGL von einer Zustandsmaschine kontrolliert. Die OpenGL-Zustandsmaschine ist ein Modell aller internen OpenGL-Zustandsvariablen. Die Ausprägung dieser Variablen legt fest, wie ein Objekt beziehungsweise eine Szene letztendlich gezeichnet wird. Hier findet sich der Ansatzpunkt für den Programmierer. Durch das gezielte setzen der Werte dieser Variablen kann der Zustand der Render-Pipeline beeinflusst und in der Folge der Zeichenprozess gesteuert werden.

Das wesentliche Charakteristikum der Zustandsmaschine ist ihr Verhalten in Bezug auf die Varianz von Zustandswerten. Wird eine Zustandsvariable auf einen bestimmten Status gesetzt, so bleibt er erhalten, bis er gegebenenfalls durch einen anderen OpenGL-Befehl verändert wird.

Es existieren verschiedene Arten von Zustandsvariablen. Manche können lediglich die Werte *an* (`enabled`) und *aus* (`disabled`) annehmen. Beispielhaft sei die Transparenz genannt, die über die Kommandos `glEnable(GL_BLEND)` und `glDisable(GL_BLEND)` aktviert beziehungsweise deaktiviert werden kann. Andere hingegen erwarten die Zuweisung

numerischer Parameter. Das setzen der Zeichen-Farbe auf Rot mittels `glColor3f(1.0f, 0.0f, 0.0f)` ist ein Beispiel für einen solchen Fall. Durch die Natur der Zustandsmaschine bliebe die Farbe bis zum Abbruch des Programms auf Rot gesetzt. Es sei denn, ein anderer OpenGL-Befehl manipulierte sie erneut.

Während des Renderns kann es vorkommen, dass es nötig wird, vorübergehend in einen anderen Zustand zu wechseln, um danach zum vorherigen Zustand zurückzukehren. Der augenblickliche Zustand muss also temporär gesichert werden können. Für diesen Fall bietet OpenGL den so genannten Zustands-Stack an. Stacks sind LIFO-Speicherstrukturen. Sie arbeiten nach dem Prinzip, dass das zuletzt im Stack gespeicherte Element auch als erstes wieder extrahiert werden muss. Abbildung 3 verdeutlicht dies in einer schematischen Darstellung.

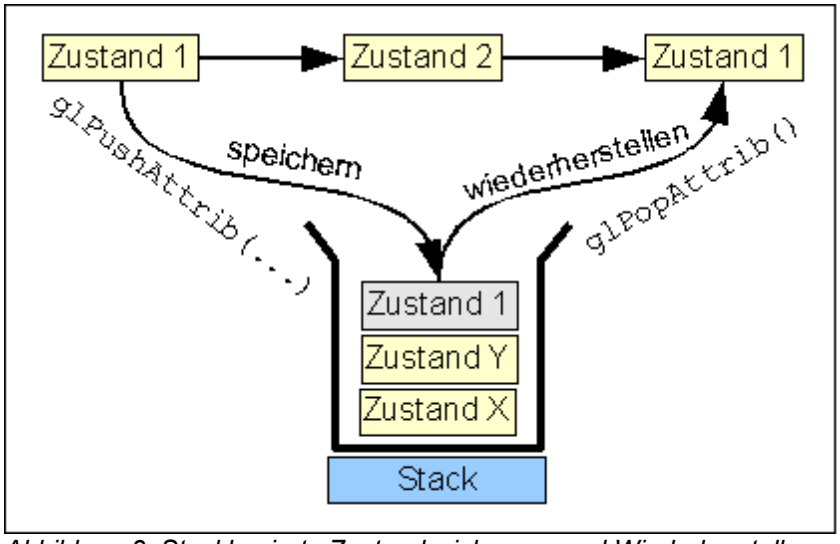


Abbildung 3: Stackbasierte Zustandssicherung und Wiederherstellung

Das Stack-Prinzip findet in OpenGL noch an anderer Stelle Verwendung. OpenGL benutzt spezielle Matrizen, um beispielsweise Transformations-Operationen darzustellen. Die aktuelle Ausprägung dieser Matrizen kann ebenfalls in einem Stack gespeichert und später wiederhergestellt werden.

3.2.3 Namenskonvention der OpenGL-Funktionen

In der Regel folgen die einzelnen Funktionen der OpenGL-Library bestimmten Konventionen der Namensgebung. Anhand dieser kann ein Programmierer direkt einige Eigenschaften der Funktion ablesen. Speziell die Art und die Anzahl der Übergabeparameter wird aus dem Funktionsnamen deutlich. Allgemein folgt die Benennung der OpenGL-Funktionen dem Schema der nachstehenden Grafik (Abbildung 4).

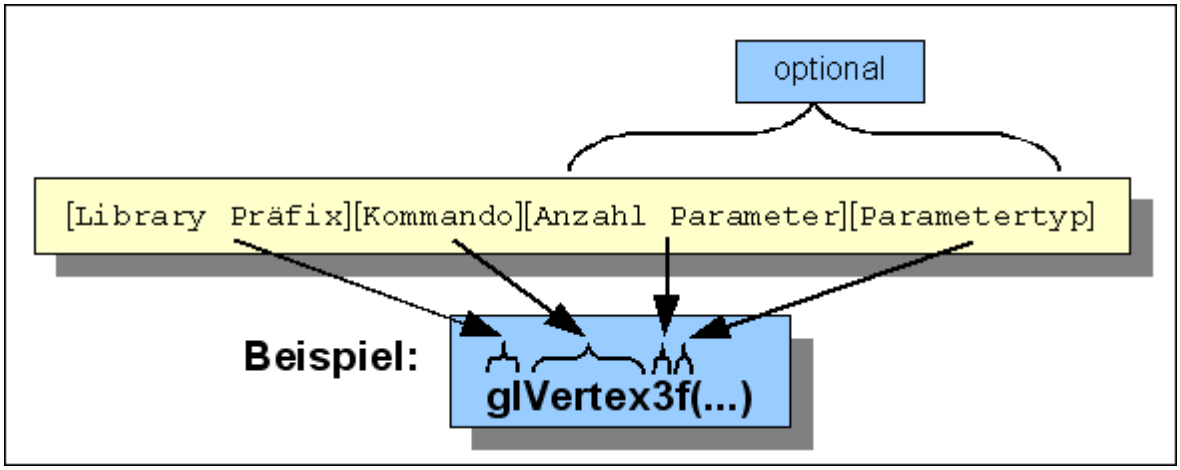


Abbildung 4: Schema zur Namensgebung der Funktionen im OpenGL-API

Von links nach rechts betrachtet, gibt der erste Teil des Funktionsnamens die Library an, aus der diese Funktion stammt. Im Beispiel ist das die *gl* Library. Weiterere mögliche Präfixe sind zum Beispiel `glu` und `glut`, die für die *GLU*- bzw. *GLUT*-Hilfsbibliotheken stehen. Als nächstes folgt der Kern des Funktionsnamens. Im Beispielfall ist dies der Befehl, der einen Punkt, im 3D-Jargon Vertex genannt, auf den Bildschirm zeichnet. Die beiden letzten Teile des Schemas sind optional, weil es in dem API sowohl Funktionen gibt, die diese Angaben aufweisen (bspw. `glColor3f(...)`), als auch solche, die ohne sie auskommen (bspw. `glPushMatrix(...)`). Sie werden genutzt, um die Anzahl und den Typ der Funktionsparameter vorzugeben. Die Beispielfunktion erwartet demnach drei Parameter vom Typ `float` als Übergabewerte.

3.2.4 Zeichnen eines einfachen Polygons

Dreidimensionale Szenen bestehen in der Regel aus einer Menge von Polygonen. Das Wort Polygon stammt aus dem Griechischen (*polys* = viel, *gonos* = Winkel) und wird häufig mit Vieleck übersetzt. Der einfachste Vertreter des Polygons ist das Dreieck.

Dreiecke sind die elementaren Bausteine virtueller Welten. Mit ihrer Hilfe lassen sich geometrische Körper beliebiger Form zusammensetzen. Je mehr Dreiecke für ein Modell verwendet werden, desto facettenreicher und detaillierter kann es erscheinen.

Da OpenGL sich als API zum Rendern von 3D-Szenen versteht, verfügt es selbstverständlich über Funktionen zum Zeichnen von Dreiecken beziehungsweise Polygonen. Um ein Polygon zu zeichnen, müssen die einzelnen Eckpunkte (Vertizes) an die Grafikkarte übermittelt werden. Die verschiedenen Varianten der OpenGL-Funktion `glVertex` dienen zur Spezifizierung und Übertragung der polygonalen Vertizes. Dabei spielt die Reihenfolge, in der die Punkte angegeben werden, eine entscheidende Rolle. Sie bestimmt, ob die Fläche des Polygons eine Front-Fläche (front face) oder eine rückwärtige Fläche (back face) ist. Dies ist wichtig für die grafische Darstellung des Polygons, da beide Varianten unterschiedlich behandelt werden können. Standardmäßig geht OpenGL von einer Spezifizierung der Vertizes entgegengesetzt dem Uhrzeigersinn aus, so dass auf diese Weise erstellte Polygone Front-Flächner sind. Die nächste Abbildung verdeutlicht beide Herangehensweisen anschaulich (Abbildung 5).

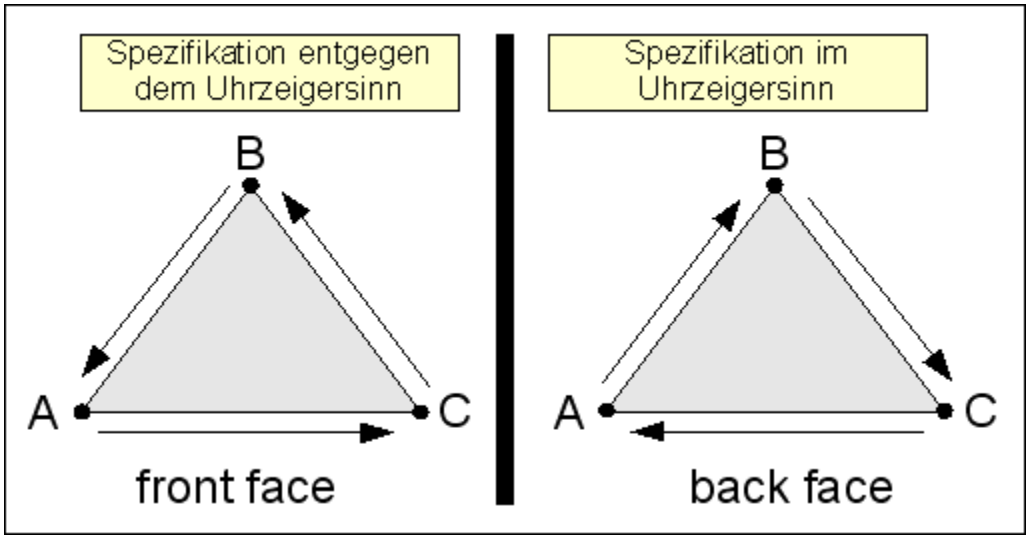


Abbildung 5: Reihenfolge der Vertex-Spezifizierung für front face und back face.

Vernachlässigt man den notwendigen Initialisierungs-Code, um einen OpenGL-Renderkontext zu erstellen, lassen sich Polygone mit wenigen Zeilen Programm-Code zeichnen. Polygone können in OpenGL allerdings auf verschiedene Art und Weise angeben werden. Am häufigsten begegnet man aber den Typen `GL_TRIANGLES` (beziehungsweise dessen Spezialfall `GL_TRINAGLE_STRIP`) für Dreiecke und `GL_QUADS` für Vierecke . Im Grunde reicht es jedoch aus, `GL_TRIANGLES` zu verwenden, da die Polygone, die mit `GL_QUADS` gezeichnet werden können, sich in äquivalenter Form auch mit `GL_TRIANGLES` realisieren lassen. Schließlich kann jedes viereckige Polygon aus zwei Dreiecken zusammengesetzt werden.

Abbildung 6 demonstriert das Rendern zweier einfacher Polygone. Für jeden Eckpunkt wird dabei die gegenwärtige Render-Farbe der OpenGL-Zustandsmaschine verändert. So entsteht der auffällige Farbverlauf auf der Polygon-Fläche. Die fett hervorgehobenen Zeilen sind die essenziellen Befehle zum Zeichnen des jeweiligen Polygons.

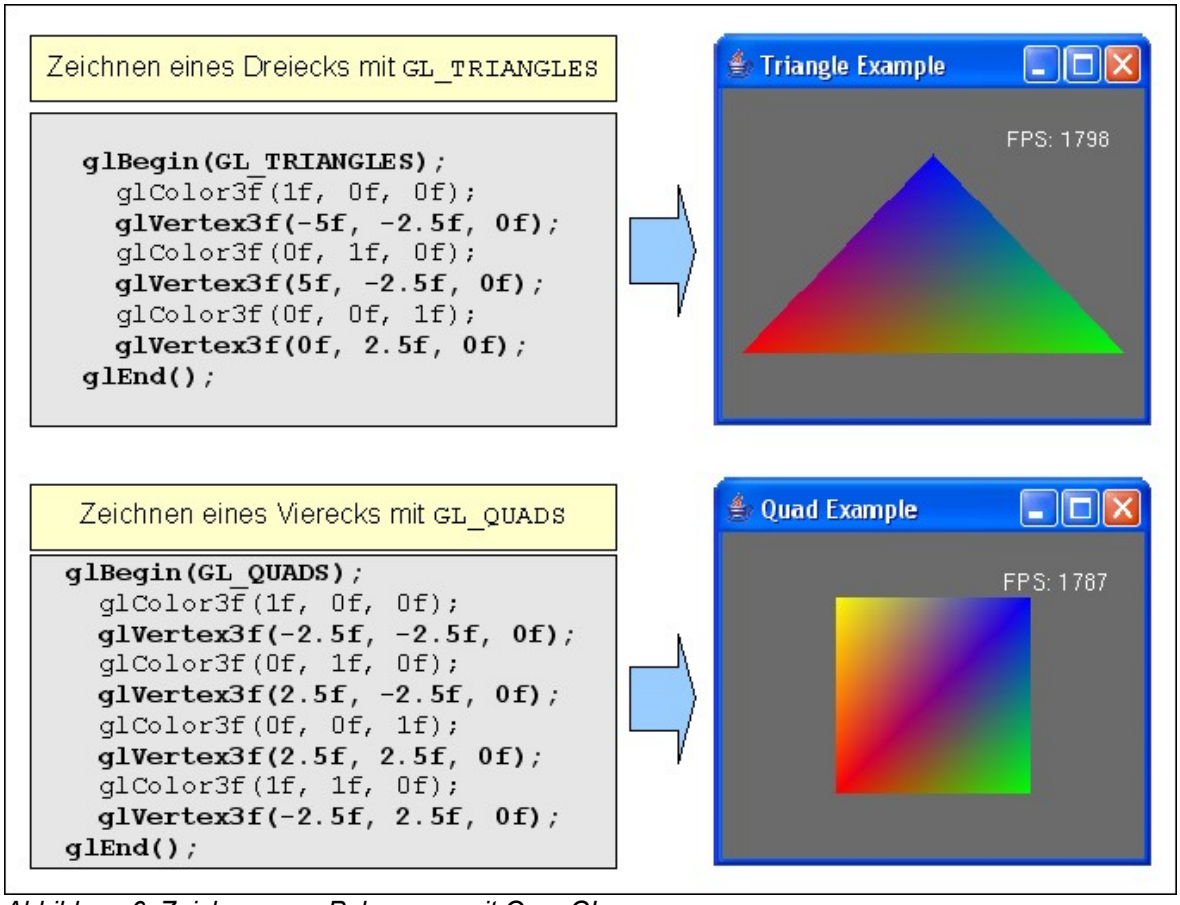


Abbildung 6: Zeichnen von Polygonen mit OpenGL

Der Aufwand zum Zeichnen von Polygonen mit OpenGL hält sich, wie zu erkennen ist, offenbar in Grenzen. Deutlich umfangreicher wird es, wenn fortgeschrittene Techniken in die Szene mit aufgenommen werden sollen. Dazu zählen beispielsweise Lichteinfall, Schattenwurf, Materialeigenschaften und Texturen. Ohne diese Techniken sind realistisch wirkende Szenen nicht denkbar. Allerdings werde ich an dieser Stelle keine tiefer gehende Einführung in OpenGL geben, um den thematischen Rahmen der Masterarbeit nicht zu sprengen.

Da die OpenGL-API ursprünglich für die Programmiersprache C entwickelt wurde, befasst sich das nächste Kapitel mit der Java-Schnittstelle zu OpenGL.

3.3 Das Java Binding für OpenGL (JOGL)

In den folgenden Unterkapiteln wird das Java Binding JOGL erläuternd aufgegriffen. In Verbindung mit OpenGL ist JOGL ein zentraler Stützpfeiler für die Umsetzung des 3D-Teils dieses Projekts.

3.3.1 Was ist JOGL?

Das *Java Binding für OpenGL*, kurz *JOGL* genannt, ist eine Schnittstelle zwischen der Programmiersprache Java und dem OpenGL-API. Es ist zwar nicht das erste Projekt, das versucht eine Brücke zwischen beiden Welten zu schlagen[11], jedoch ist es das einzige Binding, welches offiziell von der Firma Sun Microsystems unterstützt wird. Begonnen wurde JOGLs Entwicklung von Ken Russel und Chris Kline. Heute untersteht das Projekt allerdings der Sun zugehörigen Game Technology Group und wird quelloffen weiterentwickelt [Davis2004].

JOGL wird als Java Specification Request 231 (JSR-231) geführt und schickt sich an, eines Tages offizieller Teil der Java Standard Edition zu werden. Eine Besonderheit des JOGL-Projekts ist die nahtlose Integration in Java-AWT- und Swing-Fenster, so dass 3D-Komponenten in diese Container eingebettet werden können. Außerdem unterstützt es stets die aktuellste OpenGL-Version. Ein Umstand, der durch das eigens entwickelte Tool *GlueGen*[12] ermöglicht wird. Das aktuelle JOGL-Release hat die Versionsnummer JSR-231 1.1.0 (Stand: August 2007) und wird kontinuierlich verbessert.

3.3.2 Voraussetzungen für die JOGL-Programmierung

Um OpenGL-Anwendungen über JOGL implementieren zu können, muss zuvor das softwaretechnische Fundament auf dem Computer der Programmierers hergestellt werden. Für die Entwicklung einer jeden Java-Applikation ist das *Java Software Development Kit* (Java SDK) nötig. Für JOGL muss es dabei mindestens die Version J2SE 1.4 sein. Desweiteren wird eine Version des JOGL-Programmpakets vorausgesetzt. Dieses Paket besteht, je nach Ausprägung, aus einem oder zwei Jar-Files und betriebssystemabhängigen, nativen Binärdateien. Sobald diese Dateien in den Klassenpfad des Projekts aufgenommen wurden, steht der OpenGL-Programmierung in Java nichts mehr im Wege.

3.3.3 Das Interface `GLEventListener`

JOGL-Programme drehen sich im wesentlichen um die Implementierung des `GLEventListener`-Interfaces aus der JOGL-Bibliothek. Dieses Interface definiert vier Methoden, die von einer JOGL-Anwendung implementiert werden müssen. Da es sich um Event-Listener-Methoden handelt, werden sie in bestimmten Situationen (d.h. zu definierten

11 Siehe auch LWJGL (http://www.lwjgl.org) und GL4Java (http://www.jausoft.com/gl4java)

12 *GlueGen* erstellt vollautomatisch alle benötigten Interface-Methoden zur OpenGL-API, so dass auch neue Features zeitnah und ohne zusätzlichen Aufwand über das JOGL-Binding angesprochen werden können.

Events) automatisch aufgerufen. Die nachfolgende Aufzählung beleuchtet die vier Methoden etwas näher:

- **`public void init(GLAutoDrawable gld)`** In dieser Methode erfolgen alle wichtigen Operationen zur Initialisierung des OpenGL-Kontextes. Hier wird demnach der Ausgangszustand der OpenGL-Zustandsmaschine festgelegt. Beispielsweise können Anzahl und Art der initial vorhandenen Lichtquellen spezifiziert oder die Zeichen-Farbe deklariert werden. Da die Methode für den Initialisierungsprozess vorgesehen ist, wird sie nur ein einziges Mal, also zu Beginn des Programmablaufs, aufgerufen.

- **`public void display(GLAutoDrawable gld)`** Die `display()`-Methode stellt den repetitiven Kern der JOGL-Programmierung dar. Eine JOGL-Anwendung findet sich während seiner gesamten Laufzeit fast ausschließlich in dieser Methode wieder. Das kann direkt oder indirekt sein, je nachdem, ob Teile des Codes in eigene Subroutinen unterteilt wurden. In der `display()`-Methode finden alle Render-Vorgänge statt. Damit auch veränderliche Szenen gezeichnet werden können, muss die Zeichnung ständig aktualisiert werden. Aus diesem Grund wird die Methode nach jeder Beendigung eines Durchlaufs erneut aufgerufen. Ist das Zeitintervall zwischen zwei Aufrufen klein genug, ergibt sich das flüssige Bild einer dynamischen Szene.

- **`public void reshape(GLAutoDrawable gld, int x, int y, int width, int height)`** Diese Methode kommt in der Regel eher seltener zur Anwendung. Sie wird immer dann aufgerufen, wenn das Fenster, in dem die JOGL-Applikation ausgeführt wird, sich in der Größe verändert hat. Für gewöhnlich platziert man Code in dieser Methode, der das Seitenverhältnis der gezeichneten Szene an die neuen Fenstermaße anpasst.

- **`public void displayChanged(GLAutoDrawable drawable, boolean modeChanged, boolean deviceChanged)`** Diese Methode wird aufgerufen, wenn die Farbtiefe oder Auflösung des Bildschirms verändert wurde. Jedoch steht diese Funktionalität zur Zeit noch nicht in JOGL zur Verfügung. Deshalb sollte der Methodenrumpf leer bleiben.

3.3.4 OpenGL-Methoden- und Konstantenaufruf mit JOGL

Es treten nur geringe Unterschiede zwischen der herkömmlichen OpenGL-Programmierung und der Benutzung des Java-Bindings zutage. Besonders augenscheinlich sind aber die etwas abweichenden Zugriffe auf die OpenGL-Kommandos und Konstanten. Konstanten werden in OpenGL normalerweise so angegeben: `GL_<Konstantenname>`. In JOGL muss allerdings zusätzlich ein "`GL.`" vorangestellt werden. Dies ist auf JOGLs Natur als Binding zu einer C-API zurückzuführen. Alle OpenGL-Konstanten werden in JOGL als Java-Konstanten der Native-Interface-Klasse `GL` ausgedrückt. Ähnlich verhält es sich bei Methodenaufrufen. OpenGL-Methoden spricht man typischerweise mit `gl<Methodenname>` an. JOGL-Anwendungen jedoch besorgen sich ein Objekt, das auf dem GL-Interface beruht und greifen darüber auf die OpenGL-Methoden zu. Als inoffizielle Konvention hat sich die Bezeichnung `gl` für dieses Objekt durchgesetzt. Methodenaufrufe sehen mit dem Java-Binding deshalb wie folgt aus: `gl.gl<Methodenname>`.

3.3.5 Zeichnen eines einfachen Polygons

In Anlehnung an das Beispiel aus Abbildung 6, zeigt Abbildung 7, wie der JOGL-Code zum Rendern eines Polygons ausschaut. Dieses Mal wird der neue, JOGL-relevante Code fett hervorgehoben. Die nötigen Programmabschnitte zur Initialisierung des OpenGL-Kontextes beziehungsweise der Zustandsmaschine, werden abermals ausgespart.

Zeichnen eines Dreiecks mit `GL_TRIANGLES`

```
public void display(GLAutoDrawable gld)
{
  GL gl = gld.getGL(); //hole GL Objekt

  gl.glBegin(GL.GL_TRIANGLES);
    gl.glColor3f(1f, 0f, 0f);
    gl.glVertex3f(-5f, -2.5f, 0f);
    gl.glColor3f(0f, 1f, 0f);
    gl.glVertex3f(5f, -2.5f, 0f);
    gl.glColor3f(0f, 0f, 1f);
    gl.glVertex3f(0f, 2.5f, 0f);
  gl.glEnd();
}
```

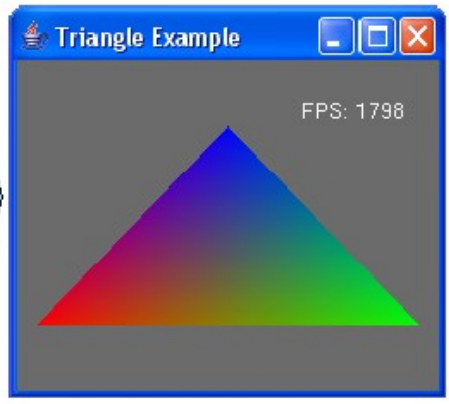


Abbildung 7: Zeichnen von Polygonen mit JOGL, dem Java-Binding für OpenGL

Um den Zusammenhang zur `display()`-Methode herzustellen, habe ich sie in den Quelltext des Schaubilds mit aufgenommen. Das `GL`-Objekt, welches zum Absetzten der OpenGL-Befehle benötigt wird, erhält man aus dem `GLAutoDrawable`-Objekt der Parameterliste.

Neuerdings haben die JOGL-Entwickler auch eine andere Möglichkeit vorgesehen, um jederzeit an eine Instanz des `GL`-Interfaces zu gelangen. Die Hilfsklasse `GLU` enthält die statische Utitlity-Methode `GLU.getCurrentGL()`. Vorausgesetzt der Aufruf erfolgt innerhalb

des JOGL-Render-Threads, liefert diese Methode an jeder beliebigen Stelle des Programms das `GL`-Objekt zurück. Das stetige Weiterreichen des `gl`-Parameters in alle Submethoden, die Zeichenoperationen ausführen wollen, erübrigt sich somit.

3.4 Scripting mit Java

Das Game-Scripting wird ein fundamentaler Bestandteil des Adventure-Frameworks. Aus diesem Grund stellen die nächsten Kapitel einige Aspekte in den Vordergrund, die in diesem Zusammenhang von Bedeutung sind.

3.4.1 Mehrwert von Skriptsprachen für die Spieleentwicklung

Die Verwendung von Skripten in Computerspielen ist weit verbreitet. Aus gutem Grund, denn sie bieten eine flexible Lösung an, um Programmobjekte von außerhalb des eigentlichen Quellcodes zu manipulieren.

Ein Level in einem Computerspiel beinhaltet für gewöhnlich Stellen, die eine gewisse Interaktivität aufweisen. Dies können zum Beispiel Schalter, Türen oder bewegliche Plattformen sein. Solche interaktiven Elemente müssen auf wechselnde Eingaben unterschiedlich reagieren können. Ein Ansatz dies zu Realisieren besteht darin, etwaige Reaktionen im Programmcode des Spiels fest zu kodieren. Wenn ein Spieleframework jedoch einen generischen Ansatz verfolgt, sollte es grundsätzlich vermieden werden, Interaktionsmöglichkeiten im Programmcode festzulegen. Stattdessen ist es ratsam, solche Programmteile in externe Skriptdateien auszulagern. Auf diese Weise können auch Endnutzer ein gewünschtes Verhalten implementieren oder ein ganz neues Spielerlebnis kreieren, ohne auf technische Schwierigkeiten zu stoßen. Durch konsequente Anwendung dieses Prinzips, gewinnt ein Framework bedeutend an Flexibilität.

Das Verhalten einer Spielumgebung kann komplett über Skripte gesteuert werden. Somit können Veränderungen vorgenommen werden, ohne den Quellcode des Frameworks berühren und neu kompilieren zu müssen.

Da Skriptsprachen häufig über eine einfachere Syntax verfügen, als normale Programmiersprachen, können auch Programmierunkundige vergleichsweise leicht Skripte erstellen. Sie müssen keine Kenntnisse über die interne Programmierung des Frameworks haben. Level-Designer und Framework-Entwickler können somit arbeitsteilig tätig werden.

Die Trennung von Spielinhalt und Framework-Code erlaubt unbegrenztes Erstellen neuer Spiele auf gemeinsamer technologischer Basis. Man schafft sich folglich ein Fundament, auf dem immer wieder neu aufgesetzt und variiert werden kann.

Es existieren verschiedene Wege, Skriptfunktionalität in das eigene Framework zu integrieren. Die folgende Auflistung (angelehnt an [Brackeen2004]) zeigt ein paar davon auf:

- **Compilierter Java-Code** Skriptdateien können aus gewöhnlichen Java-Klassen bestehen. Da es Java-Klassen sind, fügen sie sich nahtlos in das Projekt ein. Die Einbindung in kompilierter Form weist außerdem einen Geschwindigkeitsvorteil auf. Diese Methode hat aber den gewichtigen Nachteil, dass man Java beherrschen muss und bei Änderungen verpflichtet ist, den Code neu zu kompilieren (JDK benötigt). Beide Punkte erschweren die Arbeitsteilung zwischen Programmierer und Level-Designer, da der Level-Designer sich dadurch im Tätigkeitsbereich des Programmierer bewegen müsste.

- **Eigener Kommando-Interpreter** Die Implementierung eines eigenen Kommando-Interpreters erlaubt die Ausführung vordefinierter Befehle, wie beispielsweise `grosses_tor.open`. Dieses Verfahren ist vergleichsweise unflexibel und einengend, da nur solche Kommandos zur Verfügung stehen, die zuvor für den Interpreter programmiert wurden. Zusätzlich kann die Implementierung eines Skript-Interpreters sehr zeitaufwändig sein.
- **Verwendung vorgefertigter Skriptsprachen** Skriptsprachen haben oft den großen Vorteil, sehr flexibel und mächtig zu sein. Trotzdem lassen sie sich leicht handhaben und schnell erlernen. Bestimmte Skriptsprachen können Objekte zur Laufzeit beeinflussen und so das Spielgeschehen verändern. Im Vergleich zu kompiliertem Java-Code sind solche Skripte aber langsamer in der Ausführung. In der Regel fällt dies jedoch kaum ins Gewicht, da Skripte tendenziell eher wenige teure Operationen enthalten.

Die wichtigste Fähigkeit diverser Skriptsprachen im Java-Kontext ist der manipulative Zugriff auf Java-Objekte zur Laufzeit. Ist ein Java-Objekt für ein Skript zugänglich gemacht worden, lassen sich alle Methoden, die auf dem Objekt definiert wurden, auch vom Skript aus aufrufen. Der Level-Designer hat dadurch alle Möglichkeiten des Spiele-Frameworks zur Hand, ohne den Ballast der Mächtigkeit Javas mitzuführen. Der folgende Abschnitt führt die kostenlose Skriptsprache BeanShell ein, die komplett in Java und für Java entwickelt wurde. Sie wird im Rahmen dieses Projekts eingesetzt werden.

3.4.2 Die Skriptsprache BeanShell

Pat Niemeyer veröffentlichte die erste Version seines BeanShell-Projekts[13] um 1997. Er entwickelte eine freie Skript-Sprache beziehungsweise einen freien Java-Interpreter für BeanShell-Skripte. Das vollständig in und für Java implementierte BeanShell war zugleich die erste erhältliche Skript-Sprache für diese Plattform[14]. Sie schlägt eine Brücke zwischen herkömmlicher Java-Entwicklung und Skript-Programmierung, so dass beide Teile gemeinsam zu einem Projekt beitragen können.

BeanShell ist eine Skript-Sprache, die syntaktisch beinahe identisch mit gewöhnlichem Java-Code ist. Der größte Unterschied liegt im Typisierungsverhalten. Normaler Java-Code ist streng typisiert. Das heißt, alle Variablen und Rückgabewerte bekommen vom Programmierer initial einen festen Datentyp zugewiesen. Datentypen können in Java primitiv (z.B. `int`, `float`, `boolean`, `char`), oder klassenbasierte Objekte (z.B. `String`, `LinkedList`, `<selbst erstellte Klasse>`) sein. Sieht man von Typecast-Operationen ab, ändern sich diese Typen zur Laufzeit in der Regel nicht mehr. BeanShell-Code hingegen verwendet das Prinzip der losen Typisierung. Dadurch müssen Datentypen nicht mehr im Voraus spezifiziert werden, sondern werden zur Laufzeit dynamisch ermittelt und auch umgewandelt. Diese Verfahrensweise birgt für kleinskalige Programme, wie es Skripte üblicherweise sind, den Vorteil einer flexibleren und schnelleren Entwicklung. Für größer dimensionierte Projekte hingegen, wäre die Vorhersagbarkeit in Bezug auf Programm-Korrektheit stark

13 BeanShell Scripting Language – http://www.beanshell.org
14 Vgl. BeanShell Online-Dokumentation - http://www.beanshell.org/manual/bshmanual.html#History (Stand: 18.07.2007)

kompromittiert[15]. Aus diesem Grund werden BeanShell-Skripte hauptsächlich ergänzend oder für die prototypische Entwicklung eingesetzt.

Der nächste Abschnitt gibt eine Einführung in die Basis-Aspekte zum Umgang mit BeanShell. Das Hauptaugenmerk wird jedoch auf Punkte gelegt, die besonders wesentlich für den späteren Einsatz der Skriptsprache im Adventure-Framework sind. Eine detaillierte Einführung ist auf der offiziellen BeanShell Webseite zu finden.

3.4.3 Einführung in wesentliche Aspekte des BeanShell-Scriptings

Die wichtigste Voraussetzung für das BeanShell-Scripting ist das Einbinden der entsprechenden Programmbibliothek in den Klassenpfad des Projekts. Die Jar-Datei der BeanShell Library kann auf dessen offizieller Webseite heruntergeladen werden. Es stehen verschiedene .jar-Dateien zum Download bereit. Zwingend notwendig ist aber nur die als *bsh-core* bezeichnete Datei. Sie enthält die grundlegende Scripting-Funktionalität. Durch das Hinzufügen weiterer spezieller Jar-Files von der BeanShell-Webseite zum Klassenpfad des Projekts, lässt sich der Funktionsumfang noch erweitern. Dadurch werden beispielsweise bestimmte Kommandos ermöglicht, die den Skript-Code übersichtlicher halten können. Die Methode `print()` sei hier exemplarisch als kürzere Alternative zum gängigen `System.out.println()` genannt.

Sobald die BeanShell-Library installiert wurde, ist es ein Leichtes, Java-Objekte für externe Skript-Dateien zugänglich zu machen. Generell wird wie folgt vorgegangen: Zuerst muss ein Objekt der Klasse `Interpeter` aus dem Package `bsh` erstellt werden. Anschließend müssen alle Programm-Objekte, die in Skripten zugreifbar sein sollen, dem `Interpreter`-Objekt bekannt gemacht werden. Dies wird über unterschiedlich überladene `set()`-Methoden realisiert. Jedem Objekt wird dabei ein eindeutiger Name zugewiesen, der die Rolle eines identifizierenden Referenznamens in den Skripten übernimmt. Nun könnten die Java-Objekte bereits in geskripteten Code-Zeilen verwendet werden. Die `eval()`-Methode des `Interpreter`-Objekts erwartet zu diesem Zweck einen String mit Skript-Code, der bei ihrem Aufruf sofort ausgewertet wird. Da das Adventure-Framework hauptsächlich Skripts ausführen wird, die als separate Methoden in verschiedenen Skript-Dateien vorliegen, ist es angezeigt, sämtlichen Code einer Skript-Datei dem `Interpreter`-Objekt bekannt zu machen. Die Methode `source()` lädt den Inhalt einer BeanShell-Datei und führt ihn gegebenenfalls Zeile für Zeile aus. Das folgende Schaubild (Abbildung 8) verdeutlicht den Sachverhalt, indem es den gesamten Vorgang in vier Phasen zerlegt. Zu jeder Phase werden Code-Ausschnitte mit fiktiven Daten präsentiert. Außerdem wird eine Methode einer beispielhaften BeanShell-Skriptdatei angedeutet.

15 Vgl. BeanShell Online-Dokumentation - http://www.beanshell.org/manual/bshmanual.html#History (Stand: 18.07.2007)

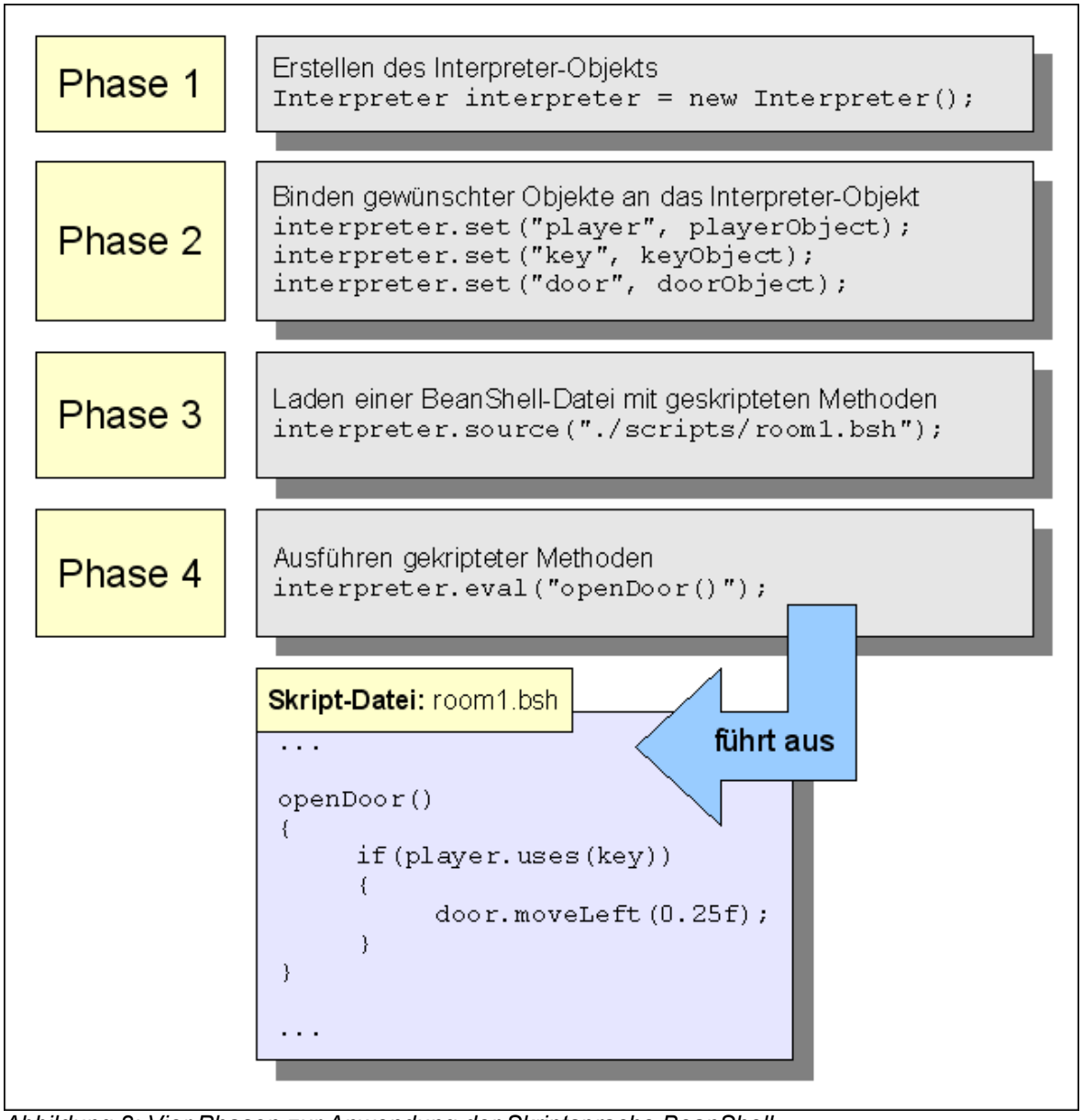


Abbildung 8: Vier Phasen zur Anwendung der Skriptsprache BeanShell

Der Grafik ist zu entnehmen, dass nur geringer Aufwand nötig ist, um Java-Programme mit Scripting-Funktionalitäten auszustatten.

Weiter oben wurde bereits angeschnitten, dass die Syntax der BeanShell-Skripte nahezu identisch ist mit herkömmlichem Java-Code. Die Unterschiede werden primär als syntaktische Erleichterungen deutlich. Beispielsweise muss BeanShell-Code nicht in Klassen und Methoden organisiert sein, sondern kann aus einer einfachen Folge von Befehlen bestehen, die sequenziell abgearbeitet werden (Abbildung 9).

```
import java.util.Date;

Date datum = new Date();
print(datum);
```

Abbildung 9: Vollständiges BeanShell-Skript mit sequenzieller Befehlsfolge

Eine `main()`-Methode ist demnach ebenfalls überflüssig. Allerdings ist es durchaus ratsam, BeanShell-Code in selbst definierte Methoden zu kapseln. Diese Methoden können dann über das zuvor besprochene `Interpreter`-Objekt gezielt aufgerufen werden. Abbildung 10 zeigt eine vollständige Skript-Datei mit zwei Beispiel-Methoden. Es wird auch verdeutlicht, dass geskriptete Methoden sich untereinander aufrufen können.

```
import java.util.Date;

printDatum(){
      Date datum = new Date();
      print(datum);
}

printDatumZweiMal(){
      int nr1 = 1;
      print("Nummer "+nr1);
      printDatum();

      int nr2 = 2;
      print("Nummer "+nr2);
      printDatum();
}
```

Abbildung 10: Strukturierung von BeanShell-Skriptcode in Methoden

Da die Variablenbehandlung in BeanShell dem Prinzip der losen Typisierung folgt, werden Variablentypen dynamisch zur Laufzeit ermittelt. Konkret bedeutet dies, dass Variablentypen bei der Initialisierung nicht explizit angegeben werden müssen. Der nächste Quelltext (Abbildung 11) basiert auf Abbildung 10, nutzt jedoch die lose Typisierung aus. Die veränderten Zeilen werden durch Fettdruck hervorgehoben.

```
import java.util.Date;

printDatum(){
      datum = new Date();
      print(datum);
}

printDatumZweiMal(){
      nr1 = 1;
      print("Nummer "+nr1);
      printDatum();

      nr2 = 2;
      print("Nummer "+nr2);
      printDatum();
}
```

Abbildung 11: Lose Typisierung in BeanShell-Skripten

Dies schließt die relativ knappe Einführung in die wichtigsten Aspekte für die Verwendung von BeanShell-Skripten im Rahmen des Adventure-Systems ab.

3.5 Die eXtensible Markup Language (XML)

XML wird innerhalb des Projekts die Rolle eines Dateiformats zum persistenten Speichern wichtiger Grunddaten spielen. Die nachfolgenden Kapitel geben dem Leser deshalb eine kleine Einführung in die Struktur von XML-Dokumenten.

3.5.1 Was ist die eXtensible Markup Language (XML)?

1996 begann das W3-Konsortium (W3C)[16] unter der Projektleitung von Tim Bray mit der Arbeit an XML. Es dauerte bis Anfang 1998, bis XML dann zum offiziellen W3C-Standard erhoben wurde [Seeboerger2004].

XML[17] ist im Grunde nur ein simples, jedoch wohldefiniertes Format für Textdateien. Es stellt eine vereinfachte Untermenge der Meta-Auszeichnungssprache SGML dar und ist inzwischen industrieweit anerkannt. Syntaktisch fällt die Verwandtschaft zu HTML, einer Auszeichnungssprache für Webseiten, auf. Diese liegt darin begründet, dass HTML ebenfalls mit SGML in Verbindung steht, beziehungsweise eine Anwendung dessen ist.

Das besondere an XML als Auszeichnungssprache ist die Möglichkeit, selbst definierte Tags und Attribute zu verwenden. Diese Eigenschaft kann dazu benutzt werden, um Daten jeglicher Ausprägung wohl strukturiert in einer XML-Textdatei zu speichern. Programme können die Datei einlesen und die Datensätze daraus extrahieren. Natürlich ist auch der umgekehrte Weg, in dem Programme Datensätze als XML-Datei speichern, gangbar. Aus diesem Grund wird XML oft als Datenaustauschformat in Betracht gezogen.

Im Gegensatz zu anderen, häufig binären Datenformaten, hat XML den Vorteil, dass es noch vom Menschen lesbar ist. Das erhöht zwar den Speicherplatzbedarf, erlaubt aber das manuelle Erstellen der Datensätze mit einfachen Texteditoren und das Auffinden von Dokumenten-Strukturfehlern durch Lesen des Klartextes.

3.5.2 Der Aufbau eines einfachen XML-Dokuments

Der Aufbau von XML-Dokumenten zeichnet sich insbesondere durch seine hierarchische Baumstruktur aus. Es gibt also ein Wurzel-Element (oberster Vater-Knoten), das so genannte Root-Tag, und Blatt-Elemente (Kind-Knoten). Kind-Knoten können ihrerseits wieder untergeordnete Knoten enthalten.

Begonnen werden XML-Dokumente mit dem *Prolog*. Er spezifiziert unter anderem die verwendete XML-Version und das Format der Zeichenkodierung. Die Angabe des Prologs ist eigentlich optional, trotz dessen sollte er stets in das Dokument mit aufgenommen werden. Es gehört erstens zum guten Stil und kann zweitens mitunter notwendig sein, damit XML verarbeitende Software das Dokument korrekt interpretiert. Nach dem Prolog folgt das Root-Tag mitsamt etwaigen Unter-Tags.

Anders als bei HTML, müssen die Tags in XML-Dateien immer abgeschlossen werden, damit man eine korrekte XML-Datei erhält. Diese Bedingung kann entweder durch ein

16 Website des W3-Konsortiums - http://www.w3.org/
17 XML auf der Website des W3-Konsortiums - http://www.w3.org/XML/

passendes Paar aus Start- und End-Tag erfüllt werden, oder durch ein leeres Tag, das jedoch Attribute enthalten kann. In der folgenden Abbildung (Abbildung 12) wird ein einfaches XML-Dokument vorgestellt, dass die Studenten einer Universität speichert.

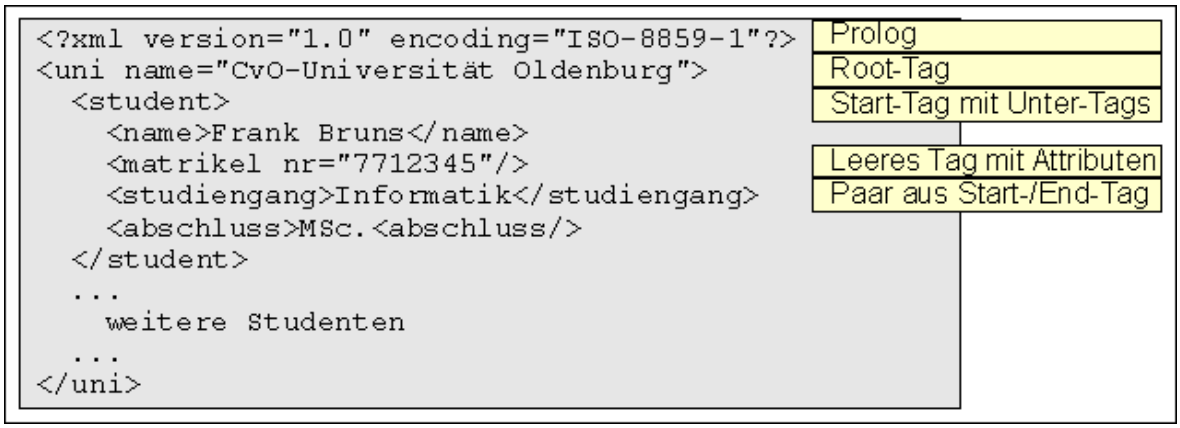


Abbildung 12: XML-Dokument mit Kennzeichnung ausgewählter Elemente

An der obigen XML-Datei lässt sich schön die strukturierte Datenhaltung in der hierarchischen Baum-Form erkennen. Durch Einrücken der Tags und Unter-Tags tritt sie deutlich hervor. Das Tag `<uni>` ist das Wurzel-Element. Über das Attribut `name` wird die Bezeichnung der Universität angegeben. Zur Universität gehören verschiedene Studenten, die über das `<student>`-Tag spezifiziert werden. Studenten verfügen wiederum über Unterelemente, die weitere Informationen über ihn festhalten. Auf diese Weise ließe sich theoretisch eine unbegrenzte Anzahl von Studenten speichern.

Würden nun Programme geschrieben, die genau diese Struktur eines XML-Dokumentes einlesen und verarbeiten könnten, dann spräche man von einem XML-Dateiaustauschformat für verschiedene Instanzen dieser Programme.

Im Anschluss an dieses Kapitel befasse ich mich mit einem API, das speziell auf das Einlesen und Schreiben von XML-Dokumenten ausgerichtet ist. Auf Basis eines solchen APIs, kann XML verarbeitende Software problemlos entwickelt werden.

3.6 Das XML-Framework *dom4j*

Im Rahmen dieses Projekts wird es nötig sein, Informationen aus XML-Dateien zu extrahieren und auch wieder als XML zu speichern. Anhand der *dom4j*-Bibliothek verdeutlichen die folgenden Unterkapitel wie die Verarbeitung von XML-Dokumenten technisch vor sich geht.

3.6.1 Was ist dom4j?

Die Verwendung von XML-Dateien als Speicher- oder Austauschformat für Software-Programme macht das maschinelle Verarbeiten der Dokumente nötig. Mit der Zeit sind verschiedene kostenlose APIs entstanden, die die Implementierung dieser Tätigkeit signifikant erleichtern. Ein solches API ist auch das XML-Framework *dom4j*[18]. Es zeichnet sich durch eine gute Performanz, hohe Flexibilität und besondere Zugänglichkeit aus.

Das *dom4j*-API wurde in Java geschrieben und stützt sich sehr stark auf Javas Collection-Framework. Im Grunde bietet *dom4j* eine einfache Möglichkeit, XML-Dokumente zu parsen, auf deren Inhalt zuzugreifen, ihn zu transformieren und letztendlich wieder als XML zu speichern. Die Struktur eines gefüllten XML-Dokumentenbaums kann aber auch völlig neu erstellt und als Datei abgelegt werden.

Bekanntermaßen existieren zwei grundlegende Ansätze für den Zugriff auf XML-Dokumente. Zum einen das ereignisgesteuerte Parsen mit *SAX*[19], dem *Simple API for XML Access,* und zum anderen der *Document Object Model (DOM)*[20] Ansatz.

Der SAX-Parser liest ein Dokument Zeile für Zeile ein. Stößt er auf ein Element (z.B. Prolog, Tags, Zeichenketten), werden entsprechende Methoden der *SAX*-API aufgerufen, die dessen Inhalt zurück liefern. Nur während dieses einen Methodenaufrufs besteht die Möglichkeit, den Rückgabewert auszugeben oder zu speichern. Danach schreitet der *SAX*-Parser zum nächsten Element voran. Ein nachträglicher Zugriff auf zuvor ermittelte Elemente ist ohne erneutes Parsen des Dokuments nicht möglich. Dieses Vorgehen ist zwar sehr schnell und speicherschonend, eignet sich aber nur zum Einlesen von XML-Dokumenten und nicht zum Bearbeiten [Seeboerger2002].

Der *DOM*-Ansatz wählt eine andere Herangehensweise. Eine XML-Datei wird eingelesen und geparst. Dabei wird das gesamte Dokument intern in einem Dokumentenbaum, eben dem *Document Object Model*, abgebildet. Über vorgesehene Methoden kann auch nach dem Abschluss des Pars-Vorgangs auf die einzelnen Elemente der Baumstruktur zugegriffen werden. Auf diese Weise wird nicht nur das Zugreifen, sondern auch das Transformieren und Zurückspeichern ermöglicht. Die große Zugänglichkeit wird aber mit geringeren Geschwindigkeiten beim Einlesen und einem erhöhten Speicherbedarf für den Dokumentenbaum vergolten [Seeboerger2002].

Dom4j vereinigt in sich sozusagen das beste aus beiden Welten. Es enthält einen *SAX*-Parser zum schnellen Einlesen einer XML-Datei, der jedoch einen *DOM*-Dokumentenbaum zurück liefern kann. Dadurch wird es möglich, trotzdem die einzelnen Elemente des Baums

18 Homepage des XML-APIs *dom4j* – http://www.dom4j.org
19 Offizielle Seite des *SAX*-APIs - http://sax.sourceforge.net/
20 *DOM*-Seite beim W3C - http://www.w3.org/DOM/

gezielt anzusprechen [Seeboerger2002]. Eine weitere schöne Eigenschaft des APIs ist die integrierte Unterstützung für *XPath*[21], die beispielsweise in der verbreiteten Konkurrenz-API *JDOM*[22] nicht ohne weiteres enthalten ist. *XPath* dient zur bequemen Navigation durch die XML-Baumstruktur anhand gezielter Pfadangaben.

Die beiden nächsten Abschnitte stellen kurz vor, wie man mit *dom4j* ein XML-Dokument erstellt, speichert und wieder einliest.

3.6.2 Kreieren und Speichern eines XML-Baums mit dom4j

Das erstellen eines XML-Dokumentenbaums geht mit *dom4j* recht leicht von der Hand. Insgesamt werden aus dessen Programmbibliothek nur wenige Klassen benötigt, um dieser Aufgabe gerecht zu werden. Ausgangspunkt ist die Klasse `Document`. Man erhält ein Objekt dieser Klasse, in dem man die Factory-Methode `DocumentHelper.createDocument()` aufruft. Weiterhin ist noch die Klasse `Element` bedeutsam. Objekte dieser Klasse repräsentieren die einzelnen Tags eines zu schreibenden XML-Dokuments. Sie können demnach auch mit Attributen versehen werden. Um die typisch baumartige Verschachtelung zu erreichen, können `Element`-Objekten wiederum Unterelemente zugewiesen werden. Letztendlich wird durch das Hinzufügen der `Element`-Objekte zum `Document`-Objekt der Baum konkretisiert. Die Klasse `XMLWriter` wird eingesetzt, um den Baum physikalisch als XML-Datei auf die Festplatte zu schreiben. Abbildung 13 zeigt den ganzen Vorgang, in dem ein kleiner XML-Baum erzeugt und abgespeichert wird. Der Inhalt orientiert sich am vorangegangenem Beispiel-Dokument aus Abbildung 12.

21 W3C-Seite zu *XPath* - http://www.w3.org/Style/XSL/
22 Website von JDOM – http://www.jdom.org

```
// Document-Objekt erstellen
Document document = DocumentHelper.createDocument();

// Tags des Dokuments erstellen, befüllen und zuweisen
Element uni = document.addElement("uni");
uni.addAttribute("name","CvO-Universität Oldenburg");

Element student = uni.addElement("student");

Element name = student.addElement("name");
name.setText("Frank Bruns");

Element matrikel = student.addElement("matrikel");
matrikel.addAttribute("nr","7712345");

Element studgang = student.addElement("studiengang");
studgang.setText("Informatik");

Element abschluss = student.addElement("abschluss");
abschluss.setText("MSc.");

// Schreiben des XML-Baums in die Datei "studenten.xml"
OutputFormat format = OutputFormat.createPrettyPrint();
format.setEncoding("ISO-8859-1");
XMLWriter writer = new XMLWriter(
                new FileWriter("studenten.xml"),format);
writer.write(document);
writer.close();
```

wird überführt in folgendes XML

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<uni name="CvO-Universität Oldenburg">
  <student>
    <name>Frank Bruns</name>
    <matrikel nr="7712345"/>
    <studiengang>Informatik</studiengang>
    <abschluss>MSc.<abschluss/>
  </student>
</uni>
```

Abbildung 13: Erstellen eines XML-Dokuments mit dom4j

Korrekterweise hätte der Prozess des Schreibens mit dem `XMLWriter`-Objekt in einen `try-catch`-Block eingebettet werden müssen, der eine `IOException` abfängt. Aus Gründen der besseren Lesbarkeit, habe ich diese Zeilen allerdings ausgelassen.

3.6.3 Einlesen eines XML-Dokuments mit dom4j

Nachdem das vorherige Kapitel anschaulich demonstrierte, wie XML-Dokumente mit *dom4j* erstellt und geschrieben werden, konzentriert sich dieses Kapitel auf den umgekehrten Weg. Wiederum anhand von Beispiel-Code führt es an, wie das Einlesen einer XML-Datei mit dom4j bewerkstelligt wird.

Am Prozess des Einlesens sind erneut nur wenige *dom4j*-Klassen beteiligt. Die Klasse `Document` ist noch aus dem vorangegangenen Kapitel bekannt. Neu sind jedoch die Klassen `SAXReader` und `Node`. Objekte der Klasse `SAXReader` dienen dazu, das XML-Dokument zu parsen. Deren `read()`-Methode liefert ein `Document`-Objekt zurück. Über das `Document`-Objekt können Knoten, das heißt Tags und Attribute, mittels spezieller Methoden, in Kombination mit *XPath*-Ausdrücken, gezielt angesteuert werden. So ein Knoten wird in *dom4j* durch die Klasse `Node` repräsentiert. Zur Extraktion des eigentlichen Inhalts des Knotens, steht eine `getText()`-Methode zur Verfügung.

Vor der Präsentation eines Beispiel-Codes, sei an dieser Stelle kurz auf die Syntax von *XPath*-Ausdrücken in *dom4j* eingegangen. Prinzipiell erfolgt die Navigation in einem XML-Dokumentenbaum entlang seiner Hierarchiestruktur. Pfadangaben in gewöhnlichen Dateisystemen lassen sich als geeignetes Analogon heranziehen. Jede Ebene wird dabei durch einen Schrägstrich (Slash) von der übergeordneten Stufe getrennt. Möchte man eine Pfadangabe ausgehend vom Wurzelknoten des Dokuments starten, so setzt man einen doppelten Schrägstrich zu Beginn des *XPath*-Ausdrucks. Es ist aber auch möglich Pfadangaben relativ zu einem Objekt der *dom4j*-Klasse `Node` anzugeben. Dem @-Zeichen kommt bei *XPath* die Funktion zu, Attribute eines Tags anzusprechen.

Die nächste Abbildung (Abbildung 14) enthält den Code, um das im letzten Kapitel erstellte Dokument ordnungsgemäß einzulesen. Die extrahierten Daten werden in `String`-Variablen abgelegt, mit denen weiterführend gearbeitet werden kann.

Abbildung 14: Einlesen von XML mit dom4j

Dieses Beispiel behandelte die Situation, dass nur ein einziger Student in der XML-Datei spezifiziert ist. Es ist aber durchaus denkbar mehrere bis viele Studenten in der Datei hinterlegt zu haben. In diesem Fall käme nicht die `selectSingleNode()`-Methode zum Einsatz, sondern stattdessen `selectNodes()`. Sie liefert eine Liste von `Node`-Objekten zurück, die mit den gängigen Werkzeugen des Java-Collections-Framework ausgelesen werden kann. Die Inhalte der Nodes können jedoch wie beschrieben angesprochen werden.

4 Analyse, Entwurf und Umsetzung

Der erste Oberabschnitt dieses Kapitels befasst sich mit der Anforderungsanalyse, dem Entwurf und letztendlich der Umsetzung einer geeigneten 3D-Game-Engine als Fundament für das Adventure-Framework. Der zweite große Abschnitt des Kapitels (Teil 2) beschreibt dann die Anforderungen, den Entwurf und die Implementierung des spielspezifischen Adventure-Frameworks selbst.

4.1 Teil 1 – Eine eigenständige 3D-Game-Engine

Da das zu erstellende Adventure-Framework nicht im klassischen 2D-Bereich angesiedelt werden soll, sondern auf Dreidimensionalität auszurichten ist, wird die Verwendung einer 3D-Game-Engine notwendig. Ein gewichtiger Teilaspekt dieser Masterarbeit ist es deshalb, ein System zu implementieren, dass für die Visualisierung der gesamten Spielwelt zuständig ist.

4.1.1 Analyse und Definition der Anforderungen

Die zu entwickelnde Game-Engine muss eine Reihe von Anforderungen erfüllen. Dieses Kapitel beschäftigt sich daher mit der Analyse bzw. Festlegung der notwendigen Soll-Eigenschaften der Engine. Nachfolgend werden einige Subkapitel präsentiert, die die besonders wesentlichen Anforderungen definieren. Jedes der Kapitel erläutert, weshalb die assoziierte Anforderung für das Gelingen der Engine von Bedeutung ist.

4.1.1.1 Eigenständigkeit

Die Engine soll unabhängig vom restlichen Adventure-Framework existieren können, damit sie gegebenenfalls in verschiedenen Projekten zum Einsatz kommen kann. Die Voraussetzung dafür ist eine interne Abgeschlossenheit der Engine. Sie dürfte nicht auf den umgebenden spielspezifischen Rahmen zugreifen.

4.1.1.2 Portabilität

Die Engine soll portabel sein. Portabilität ist ein überzeugender Vorteil gegenüber klassischen 3D-Engines, die häufig für ein spezielles Betriebssystem konzipiert werden. Dadurch wird der plattformübergreifende Einsatz möglich, ohne nennenswerten Mehraufwand betreiben zu müssen. Besonders für den kommerziellen Sektor ist dies ein attraktiver Faktor. Denn der Kreis potenzieller Nutzer wird durch plattformunabhängige Technik mühelos erweitert.

4.1.1.3 Objektorientiertheit

Die gesamte Engine soll einen objektorientierten Ansatz verfolgen. Dazu gehört auch die Kapselung der imperativen OpenGL-Kommandos in Objekt-Strukturen. Durch eine solche Vorgehensweise kann das System von allen Vorteilen profitieren, die gemeinhin mit Objektorientierung in Verbindung gebracht werden. Insbesondere Modularität ist eine

erstrebenswerte Eigenschaft in diesem Zusammenhang, da sie die Wiederverwendbarkeit von Programm-Modulen erhöht.

4.1.1.4 Performanz

Computerspiele im Allgemeinen sind Anwendungen, die versuchen, dynamische Umgebungen in Echtzeit darzustellen. Damit für den Spieler der Eindruck einer glaubwürdigen Szene entsteht, sollte sie mit möglichst hohen Bildwiederholungsraten gerendert werden. Das Auge nimmt abgebildete Bewegungen erst als flüssig wahr, wenn sie mit etwa 24 bis 25 Bildern pro Sekunde aktualisiert werden. Unterhalb des kritischen Werts, erscheinen Animationen nicht mehr kontinuierlich, sondern wirken abgehackt und stockend. Je komplexer eine Szene aufgebaut ist, desto höher wird die Belastung an die Render-Hardware.

Komplexe Szenen sind häufig gleichzusetzen mit hohen Polygonzahlen, die von der Grafikkarte gezeichnet werden müssen. Aus diesem Grund sind Vorkehrungen zu treffen, die die Belastungen der Hardware, wenn möglich, minimieren.

4.1.1.5 Import von externen 3D-Modellen

Dreidimensionale Umgebungen wirken umso interessanter für einen Betrachter, je mehr Details es zu entdecken gibt. Das trifft nicht nur auf die reale Welt, sondern auch auf computergenerierte Schauplätze zu. Durch den reichhaltigen Einbau von 3D-Objekten in die künstliche Welt, nimmt deren Facettenreichtum enorm zu. Zugleich wird die emotionale Bindung eines Spielers zur erlebten Szene gefördert, weil er Parallelen zur Vielfältigkeit der realen Welt erkennt.

Da 3D-Objekte, auch 3D-Modelle oder Geometrien genannt, in der Regel strukturell zu komplex sind, als dass sich die Eckpunkte ihrer Polygone manuell spezifizieren ließen, werden sie mit Hilfe spezialisierter Software-Werkzeuge angefertigt. So modellierte Objekte, werden in Form von Dateien weitergegeben.

Die Game-Engine soll die Fähigkeit besitzen, solche Dateien einzulesen und die darin enthaltenen Polygon-Daten als 3D-Modelle zu visualisieren.

4.1.1.6 Statische und animierte Szene-Komponenten

Spielerisch attraktive Computerwelten bestehen aus einer Fülle von Objekten. Zur Steigerung der Glaubwürdigkeit ist es von Vorteil, wenn die Spielwelt der Realität nachempfunden wird. So soll die Game-Engine zweierlei Arten von Szene-Komponenten präsentieren können: Statische und animierte. Als statische Komponenten werden alle Objekte bezeichnet, die fest in ihrer Ausgangsform verharren. Besonders die Kulissen der Spielumgebung (Gebäude, Möbel, Bäume usw.) bestehen meist aus statischen Objekten. Im Gegensatz dazu, verändern animierte Komponenten ihre Erscheinungsform im laufe der Zeit. Insbesondere Lebewesen (Tiere, Menschen, humanoide Kreaturen usw.) treten als animierte Objekte auf, wenn sie zum Beispiel gehen oder rennen.

Die Game-Engine soll zwischen beiden Klassen unterscheiden können und sie spezifisch unterstützen.

4.1.1.7 Kamera-System

Der große Vorteil dreidimensionaler grafischer Umgebungen ist die inhärente Eigenschaft, ein und die selbe Szene aus verschiedenen Blickwinkeln präsentieren zu können. Während bei 2D-Spielen für wechselnde Ansichten jeweils spezielle Grafiken entworfen werden müssten, bleibt eine 3D-Szene stets aus einem Guss. Sie wird einmal erstellt und kann dann aus beliebigen Perspektiven betrachtet werden. Die Bewegung des Betrachters durch die Welt, weist eine ausgeprägte Analogie zur Fahrt einer Kamera durch Filmkulissen auf.

In der Tat wird in der 3D-Terminologie die Position, von der aus eine Szene betrachtet wird, auch Kamera-Position genannt. Die Engine soll die Kamera-Analogie aufnehmen, so dass der Blickwinkel auf eine Umgebung frei definiert und dynamisch verändert werden kann.

4.1.1.8 Kollisionserkennung

Es entspricht unserer alltäglichen Erfahrung, dass feste Objekte, seien es Lebewesen oder Gegenstände, sich nicht ohne weiteres durchdringen können. Kann dieses Verhalten auch auf eine Spielwelt übertragen werden, so wird sie vom Verstand des Spielers schneller akzeptiert. Ein Wiedererkennungseffekt stellt sich ein. Die Fähigkeit einer Engine, feststellen zu können, ob zwei Objekte miteinander kollidieren, trägt daher signifikant zur Glaubwürdigkeit der Szene bei.

Viele Spielprinzipien sind auf korrekte Kollisionserkennung beweglicher und statischer Objekte angewiesen. Rennspiele zum Beispiel würden konzeptionell sicherlich versagen, wenn der Spieler mit seinem Gefährt den Straßenhindernissen oder konkurrierenden Fahrzeugen nicht ausweichen müsste. Kollisionserkennung ist in diesem Fall ein Mittel, um die fahrerische Herausforderung und den Spielspaß zu steigern.

Aus den genannten Gründen sollen Verfahren implementiert werden, die Kollisionen unterschiedlicher Objekte im dreidimensionalen Raum erkennen können.

4.1.1.9 Wegfindungssystem

Unter *Wegfindungssystemen* versteht man algorithmische Verfahren, mit dem sich Pfade von Start- zu Zielpunkten berechnen lassen. Im Allgemeinen müssen kalkulierte Wege nicht immer direkte Pfade beschreiben, sondern können durch besondere Gegebenheiten der Level-Geometrie beeinflusst werden. Beispielsweise kann der direkte Weg zum Ziel durch ein undurchdringliches Hindernis in der Spielwelt, vielleicht einem Mauerabschnitt, blockiert sein. In der Verantwortung des Wegfindungssystems liegt in einem solchen Fall die Ermittlung eines praktikablen Pfades, der um das Hindernis herum führt.

Viele Spielprinzipien benötigen Methoden zur Wegfindung. Häufig werden sie eingesetzt, um KI-gesteuerte Spielfiguren plausibel in der Umwelt navigieren zu lassen. Aber auch vom Spieler kontrollierte Figuren, können sich Wegfindungssysteme zu Nutze machen. Falls die Steuerungsoptionen des Spielers nur darin bestehen, Zielpunkte auszuwählen, dann dient die Wegfindung zum Berechnen der Zwischenstationen.

Da die computergestützte Wegfindung ein elementares Werkzeug, insbesondere für Navigationshandlungen der künstlichen Intelligenz ist, soll die zu entwickelnde Game-Engine ein angemessenes Verfahren bereitstellen.

4.1.1.10 Licht- und Schattendarstellung

Die Beleuchtung einer 3D-Szene kann ausschlaggebend sein für ihre atmosphärische Dichte und Glaubwürdigkeit. So kann die symbolische Aussagekraft einer Beleuchtungsfarbe Emotionen in uns wecken. Viele Menschen werden sich zum Beispiel an das Bild rot beleuchteter U-Boot-Brücken in diversen Kinofilmen erinnern. Da dieses Licht in der Regel im Alarmfall geschaltet wird, verbindet das Publikum Gefahrensituationen mit ihm.

Beleuchtungen können zudem den Wiedererkennungseffekt im Menschen auslösen. Ein Zimmer, in dem Fotos entwickelt werden (Dunkelkammer), wird sofort als solches erkannt, wenn das charakteristische Rotlicht in ihm brennt. Ein weiteres Beispiel sind Tageszeiten. Während viel helles Licht den Betrachter glauben macht, die Spielwelt werde vom Tagesgestirn erleuchtet, vermittelt eine spärliche Illumination den Eindruck von Nachtzeiten.

Die Game-Engine soll den Einsatz verschiedener Beleuchtungsmechanismen zur Verfügung stellen, so dass die Lichtverhältnisse der Szene individuell gestaltet werden können.

Der Einsatz von Licht wirkt nur dann ausreichend realistisch, wenn er sich einigermaßen mit unseren Alltagserfahrungen deckt. Wo Licht ist, da ist bekanntlich auch Schatten. Schatten lassen eine virtuelle Welt wirklichkeitsgetreuer aussehen. Besonders dann, wenn deren Ausrichtung dynamisch in Bezug zur Position einer Lichtquelle steht.

Nicht zuletzt nützen Schatten auch dem räumlichen Sehen, um besser erkennen zu können, an welcher Stelle sich ein Objekt relativ zur Boden-Ebene befindet. Wird eine Szene ohne Schattendarstellung ihrer Modelle gezeichnet, entsteht bei ungünstigen Blickwinkeln schnell der Eindruck, manche Objekte würden sich nicht auf dem Boden befinden, sondern in geringer Höhe über ihn hinweg schweben.

Die Unterstützung dynamischer Schattenwurf-Verfahren ist demnach eine sinnvolle Anforderung an die Game-Engine zur Verfeinerung des visuellen Gesamtbildes.

Schattendarstellungen können gleichwohl sehr anspruchsvoll sein, wenn es um den Verbrauch von Systemressourcen geht. Aus diesem Grund müssten Schatten durch die Engine selektiv gerendert werden. Das bedeutet, es sollte möglich sein, präzise auszuwählen, welche Objekte Schatten werfen und welche nicht.

4.1.1.11 Optische und akustische Spezialeffekte

Neben der Schattendarstellung, soll die Game-Engine einige weitere Spezialeffekte beherrschen, die dazu beitragen können, die Spielwelt zu beleben. Dazu zählen sowohl Reize optischer Natur, als auch akustische Anregungen. Besonders durch die Tontechnik kann die Grundstimmung einer Szene entscheidend mitgeprägt werden.

Detaillierte Klangkulissen aus Geräuschen, Dialogen und Liedern, bedeuten nicht selten ein erhöhtes Aufkommen an Audio-Dateien. Damit die Datenmenge eines Spiels durch die Fülle an Tonmaterial dennoch nicht unnötig aufgebläht wird, sollten komprimierte Audio-Formate verwendet werden können.

4.1.1.12 Orthographische Projektion von Texten und Bild-Grafiken

Die Ausgabe von Text auf dem Bildschirm ist ein fundamentales Konzept zur Vermittlung von Informationen der Spiele an die Spieler. Bevor die Wiedergabe von digitalisierten Sprachaufnahmen in Video-Spielen Einzug hielt, waren Textnachrichten der zentrale Kommunikationskanal zum Spieler. Vom simplen *„Game Over“* bis zur Erzählung epischer Geschichten reichte die Bandbreite deren Einsatzes. Heutzutage werden Texte oftmals nur noch ergänzend (Untertitel) oder zum Anzeigen längerfristig relevanter Daten (Statusinformationen, Missionsziele) eingesetzt.

Trotzdem ist es für eine Game-Engine unabdingbar, Texte visualisieren zu können. Im Zusammenspiel mit Bild-Grafiken, werden sie benutzt, um so genannte Head-Up Displays (HUDs) zu generieren.

HUDs sind eine Methode zur Darstellung von Benutzerinterfaces in Spielen. Eine Komposition aus Text und (für gewöhnlich leicht transparenten) Bildern wird als 2D-Overlay über die 3D-Szene gelegt und versorgt den Spieler mit Informationen oder stellt interaktive Spielmenüs bereit.

Die Game-Engine soll Texte und Bild-Grafiken als orthographisch projizierten Overlay rendern können, so dass ein Verfahren zur einfachen Ausgabe von Texten und zur Erstellung von Head-Up Displays gegeben ist.

4.1.1.13 Unterstützung peripherer Eingabegeräte

Interaktive Videospiele sind konzeptionell auf Steuerungsbefehle ihrer Anwender angewiesen. Umgekehrt wird ein Spiel erst dann interaktiv, wenn es sich von außen beeinflussen lässt. Als Schnittstelle zwischen Mensch und Maschine kann spezielle Hardware herangezogen werden, die Kommandoeingaben der Spieler an die Spiele übermittelt.

Die Hardware-Industrie vertreibt ein riesiges Angebot an Eingabegeräten, die sich peripher an Computer oder Konsolen anschließen lassen. Zu den klassischen Geräten, wie Tastatur und Maus, gesellen sich mittlerweile Kontrollsysteme, wie Steuerknüppel (Joysticks) und Spiele-Lenkräder, die mit ausgefeilten Funktionen aufwarten.

Damit Spiele, die auf der zu entwickelnden Game-Engine basieren, interaktiv auf Benutzer-Eingaben reagieren können, sollen Komponenten implementiert werden, die auf Impulse von Eingabegeräten ansprechen.

4.1.2 Entwurf

Dieses Kapitel beschreibt den Entwurf der 3D-Game-Engine auf Basis der in den Vorkapiteln festgehaltenen Anforderungen. Ich werde dabei die einzelnen Anforderungen der Reihe nach in den Entwurf der Engine einfließen lassen. In Folge dessen ergibt sich ein ähnlich strukturierter Kapitelaufbau wie bei der Anforderungsdefinition. An Stellen, an denen es sich jedoch als sinnvoll erweist, können einzelne Entwurfsaspekte mehrere Anforderungen unter einem einheitlichen Oberbegriff behandeln.

4.1.2.1 Eigenständigkeit

Die zu entwickelnde Game-Engine soll eigenständig sein. Die gewünschte Unabhängigkeit lässt sich dadurch erreichen, dass das System implementiert wird, ohne speziellen Bezug zu einem Spiel oder Spieltyp aufzubauen, der später eventuell auf Basis der Engine realisiert werden könnte. Die Architektur muss demnach generisch und, so weit möglich, losgelöst von zukünftigen Anwendungen ausgelegt werden. Die Implementierung der Engine vor dem Entwurf des eigentlichen Adventure-Frameworks, gewährleistet die Trennung beider Welten.

4.1.2.2 Einsatz von Java als objektorientierte Programmiersprache

Als wichtige Anforderungen an die Engine wurden unter anderen Portabilität und Objektorientiertheit angeführt. Die Programmiersprache Java erfüllt beide Anforderungen recht gut. Einerseits gehört sie zu den Sprachen, die das Paradigma der objektorientierten Programmierung sehr dezidiert umsetzen. Andererseits sticht Java in der informationsverarbeitenden Industrie durch sein Konzept der virtuellen Laufzeitumgebung hervor, welches eine weitreichende Plattformunabhängigkeit verspricht.

Einhergehend mit der Objektorientiertheit ist es in Java möglich, die typisch imperativen OpenGL-Befehle in objektorientierte Methoden und Klassen zusammenzufassen (Kapselung). Erstellt ein Programmierer Objekte dieser Klassen, lassen sich die Methoden gezielt aufrufen. Der Vorteil der Kapselung von OpenGL-Kommandos in Java-Code liegt im potenziell vereinfachten Umgang mit ihnen. Der Aufruf einer einzelnen Methode kann eine Vielzahl gekapselter OpenGL-Statements an die Grafikkarte senden, ohne dass der Programmierer Kenntnis über die eventuell komplizierten Interna des Codes haben müsste.

Diesen Umstand machen sich objektorientierte 3D-Game-Engines exzessiv zu Nutze. Auch die zu entwickelnde Engine setzt hier an. Durch die Kapselung häufig benötigter OpenGL-Funktionalitäten in themenspezifische Klassen, kann eine hohe Anwendungsfreundlichkeit, insbesondere für Unkundige der OpenGL-Grafik-Programmierung, erreicht werden. Auf diese Weise ist es beispielsweise denkbar, den Einsatz des OpenGL-Beleuchtungssystems oder das Zeichnen komplexer 3D-Modelle auf wenige Zeilen Java-Code zu reduzieren.

Abbildung 15 veranschaulicht die Kapselung von OpenGL-Befehlen praktisch. Es wird ausschnittsweise demonstriert, wie die OpenGL-interne Nebelfunktionalität (Fog) in eine Java-Klasse gekapselt werden kann. In der Abbildung selbst wird jedoch nicht die vollständige Klasse präsentiert, sondern nur deren `draw()`-Methode, die den relevanten OpenGL-Code enthält.

```
public void draw(GL gl)
{
  if(this.enabled)
  {
    gl.glEnable(GL.GL_FOG);

    // activate correct fog mode
    if(this.linearMode) gl.glFogi(GL.GL_FOG_MODE, GL.GL_LINEAR);
    if(this.expMode) gl.glFogi(GL.GL_FOG_MODE, GL.GL_EXP);
    if(this.exp2Mode) gl.glFogi(GL.GL_FOG_MODE, GL.GL_EXP2);

    // adjust density
    gl.glFogf(GL.GL_FOG_DENSITY,this.density);

    // set fog color
    gl.glFogfv(GL.GL_FOG_COLOR,this.color,0);

    // set fog start and end
    gl.glFogf(GL.GL_FOG_START,this.start);
    gl.glFogf(GL.GL_FOG_END,this.end);
  }
  else gl.glDisable(GL.GL_FOG);
}
```

Abbildung 15: Kapselung von OpenGL-Befehlen in eine Java-Methode

Eine Klasse, die diese `draw()`-Methode enthielte, würde wahrscheinlich über diverse Attribute verfügen, um die Nebelfunktion aktiv zu beeinflussen. Angedeutet wird dies durch die mit `this` referenzierten Membervariablen.

Möchte ein Programmierer OpenGL-Nebel in seiner 3D-Applikation verwenden, so muss er lediglich ein Objekt der Nebel-Klasse erstellen und innerhalb von JOGLs `display()`-Methode dessen `draw()`-Methode aufrufen.

4.1.2.3 Verwendung vorgefertigter 3D-Modelle

Wie im Anforderungskapitel bereits erwähnt, sind 3D-Welten umso interessanter, je vielfältiger sie sind. Die Platzierung einer Fülle von geeigneten 3D-Objekten lässt die Szene realistischer erscheinen. Nun ist es so, dass moderne Spiel-Figuren und Objekte aus Hunderten bis Tausenden Polygonen bestehen. Jedes Polygon besteht wiederum aus einzelnen Koordinaten für seine Eckpunkte. Schnell wird klar, dass eine solche Menge an Koordinaten nicht mehr händisch vom Menschen spezifiziert werden kann. Daher greift man auf spezielle Software-Tools zurück, die es erlauben, 3D-Modelle grafisch zu gestalten und in Form von Dateien abzuspeichern. Unter Umständen können solche Modelle auch Animationen enthalten, die ebenfalls in den Modell-Dateien kodiert werden.

Das Angebot an Software-Werkzeugen zur Modell-Gestaltung ist reichhaltig. Oftmals führen deren Hersteller ein proprietäres Dateiformat für die Modell-Dateien ein, die mit ihrer Software erstellt werden. Einige dieser Formate haben über Tool-Grenzen hinweg Verbreitung gefunden.

Dieses Projekt wird drei sehr verbreitete Modell-Formate zumindest rudimentär unterstützen. Zum einen das *3ds*-Format von *Autodesks 3ds Max*[23], desweiteren das vergleichsweise simple *md2*-Format, welches in *id Softwares Quake2*[24]-Spielen zum Einsatz kam und letztlich das *MilkShape-3D-ASCII-Format*, des beliebten Low-Price-Modellers *MilkShape 3D*[25].

Im Grunde enthalten alle genannten Dateiformate hauptsächlich die 3D-Koordinaten der Polygon-Vertizes aus denen ihre Modelle bestehen und etwaige Textur-Koordinaten. Gegebenenfalls sind auch Animationen enthalten, die entweder in Form zusätzlicher Vertex-Koordinaten (md2), oder als Transformations-Matrizen vorliegen, die die Polygon-Vertizes animationsgemäß transformieren (MilkShape 3D).

Damit die 3D-Modelle aus den Dateien in die laufende Applikation eingefügt werden können, müssen Import-Klassen vorhanden sein, die die Modelldaten auslesen. Da das Rad nicht ständig neu erfunden werden sollte, können fertige Importierer aus anderen Projekten diese Standard-Aufgabe übernehmen. Sofern sie frei verfügbar sind, werden sie in angepasster Form in die Engine integriert.

Die Komponenten einer 3D-Welt können grob in zwei Lager klassifiziert werden: Statische und animierte Objekte. Die Engine wird beide Fälle unterscheiden können und spezifische Operationen auf ihnen erlauben. Animierte Objekte können als Erweiterung zu statischen Objekten verstanden werden. Das heißt, animierte Modelle haben im Grunde die gleichen Eigenschaften, wie statische. Allerdings ergänzen sie diese um spezielle Fähigkeiten. Beiden Typen gemeinsam sind grundlegende Mechanismen zur Positionierung, Rotation, Skalierung und Texturierung der Modelle. Animierte Modelle erhalten zusätzliche Operationen zur gezielten Einstellung und Verarbeitung der gewünschten Animationsphasen.

4.1.2.4 Ein flexibles Kamera-System

Interessante Kamera-Einstellungen können helfen, die 3D-Welt eindrucksvoll zu inszenieren. Die Engine wird ein Kamera-System erhalten, welches sowohl für unbewegte Perspektiven als auch für freie Kamera-Bewegungen geeignet ist.

OpenGL verfügt über Einflussmöglichkeiten, die die 3D-Projektion maßgeblich bestimmen. Dazu gehören beispielsweise die Angabe der nahen und fernen Clipping-Ebene und das Setzen des Projektions-Winkels des Sichtkegels (Field of View). Die Engine wird den Prozess der Spezifikation der 3D-Projektion in einer Kamera-Klasse kapseln, so dass er möglichst einfach durchzuführen ist.

Das Bild, welches von der Kamera eingefangen wird, definiert sich nicht nur durch die Projektions-Einstellungen, sondern insbesondere durch Position und Ausrichtung des Sichtkegels. Dem Relativitätsprinzip entsprechend, ist es theoretisch nicht zu unterscheiden, ob ein verändertes Kamera-Bild durch Bewegung der Kamera oder etwa Verschiebung der Welt selbst entsteht. Deshalb ist es durchaus zulässig, die Orientierung der Kamera intern als Transformation der Welt abzubilden. Nach außen soll gleichwohl der Eindruck erweckt werden, die Kamera sei der mobile Teil der Szene, weil dies den Alltagserfahrungen des

23 Autodesk 3ds Max - http://www.autodesk.de/adsk/servlet/index?siteID=403786&id=9899861 (Stand: 28.08.2007)
24 Quake2 von id Software - http://www.idsoftware.com/games/quake/quake2 (Stand: 28.08.2007)
25 MilkShape 3D - http://chumbalum.swissquake.ch (Stand: 28.08.2007)

Menschen deutlich näher kommt. Die oben erwähnte Kamera-Klasse stellt dazu Methoden bereit, die explizit die Positionierung und Ausrichtung der Kamera verändern können. Im Hinterkopf zu behalten ist aber, dass es sich tatsächlich um gekapselte Translations- und Rotationsbefehle auf OpenGL-Ebene handelt, die die gesamte Welt, invers zur vermeintlichen Kamera-Bewegung, transformieren.

4.1.2.5 Maßnahmen zur Leistungsoptimierung des Render-Prozesses

Die Darstellung von dynamischen, in Echtzeit generierten virtuellen Welten, ist seit je her eine anspruchsvolle Herausforderung für die zugrunde liegende Computer-Hardware. Zwar bemüht sich die Industrie, durch stete Weiterentwicklung der Computer-Technik, ihre Produkte leistungsfähiger zu machen, doch entstehen im gleichen Atemzug immer modernere Anwendungen, die die Hardware erneut auszureizen suchen. Mit anderen Worten: Hardware wird ob der Anwendungen und steigenden Ansprüche an sie, wohl niemals *ausreichend* Rechenkraft besitzen. Anwendungsentwickler müssen sich infolgedessen Gedanken über praktikable Wege zur Effizienzsteigerung machen.

Es existieren verschiedene Verfahren, die der Ausführungsgeschwindigkeit des 3D-Render-Prozesses zugute kommen können. Die folgenden Seiten beschreiben drei weit verbreitete Mechanismen, die in den Entwurf der Engine eingehen sollen.

Beschleunigtes Rendern durch Hardware-Unterstützung

Unter Hardwarebeschleunigung versteht man im Gebiet der 3D-Grafik die Entlastung der CPU durch spezialisierte Hardwarekomponenten. Heutige Grafikkarten sind üblicherweise dafür ausgelegt, rechenintensive Grafikoperationen aus der Verantwortung der CPU zu nehmen und sie mit ihrem Grafik-Prozessor (GPU) zu verarbeiten. Die Grafikkarten-Schnittstelle OpenGL setzt exakt an diesem Punkt an, da sie als Schlüssel zu den Ressourcen der Grafik-Hardware zu verstehen ist.

Mit zunehmender Komplexität der 3D-Szenen, steigt die Anzahl der Polygone an, die es zu verarbeiten gilt. Hohe Polygonzahlen stellen enorme Ansprüche an die 3D-Hardware. Damit die Grafikkarte nicht zum Performance-Flaschenhals wird, weil sie die Menge an Polygonen nicht in akzeptabler Zeit bewältigen kann, ist dafür Sorge zu tragen, dass nur die notwendigsten Geometrien gezeichnet werden müssen.

An dieser Stelle kommen die Culling-Verfahren ins Spiel (von engl. *to cull*: *aussondern*). Das Prinzip des Cullings bezeichnet das selektive Filtern von Polygonen, die während der aktuellen Render-Schleife vom Betrachter nicht visuell wahrzunehmen sind. Wenn die selektierten Polygone nicht zum Zeichnen an die Grafikkarte transferiert werden, vermindert sich die Gesamtbelastung der Hardware gegebenenfalls merklich.

Für die Umsetzung in der Engine ziehe ich zwei Verfahren besonders in Betracht, weil sie sowohl effektiv, als auch relativ zugänglich sind. Sie werden nachfolgend veranschaulicht.

Frustum-Culling

Das *Frustum-Culling* filtert die Elemente einer virtuellen 3D-Welt unter Zuhilfenahme des OpenGL-Kamera-Sichtkegels, dem so genannten *Frustum*. Polygone werden dabei auf Sichtbarkeit im Frustum getestet und gegebenenfalls vom Render-Prozess ausgeschlossen.

Der Kamera-Frustum definiert den Sichtbereich in die 3D-Welt. Er hat die Form eines Pyramidenstumpfes, der durch sechs Ebenen aufgespannt wird. Die einzelnen Ebenen werden als *near plane*, *far plane*, *left plane*, *right plane*, *top plane* und *bottom plane* bezeichnet. Zusammengenommen nennt man sie auch *Clipping Planes*. Objekte, die nicht innerhalb aller sechs Ebenen des Pyramidenstumpfes Position beziehen, können optisch nicht wahrgenommen werden. Die folgende Zeichnung (Abbildung 16) demonstriert Lage und Benennung der einzelnen Clipping Planes des Sichtkegels.

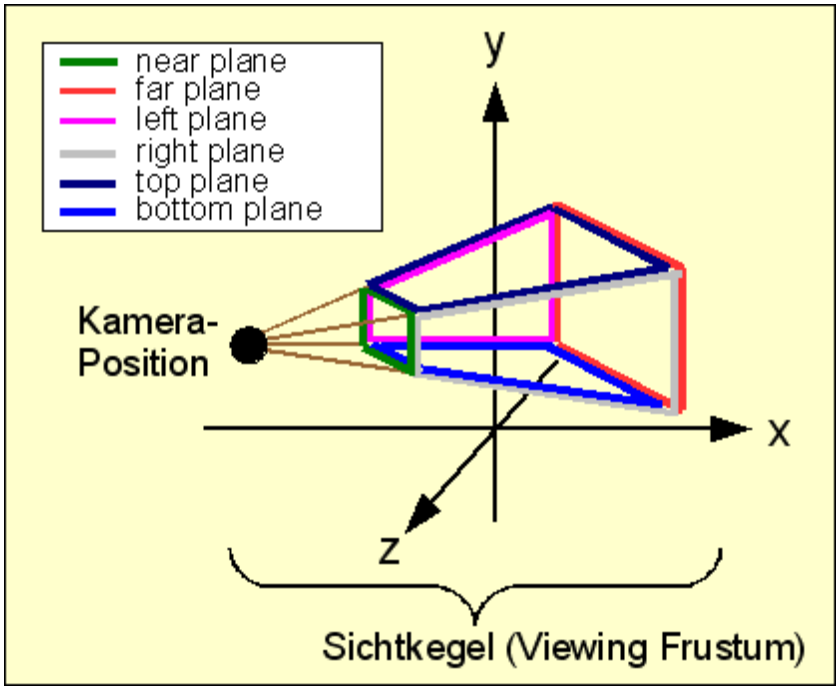


Abbildung 16: Kamera-Sichtkegel (Viewing Frustum) mit Clipping Planes

OpenGL verfügt über einen integrierten Mechanismus, der Faces, die außerhalb des Kamera-Frustums liegen, beim Zeichnen automatisch ignoriert beziehungsweise nur die Teile rendert, die noch partiell sichtbar sind (Clipping). Allerdings ist dies wenig effektiv, da immerzu alle Polygone der Szene an die Grafikkarte weitergeleitet werden müssen, ehe der interne Mechanismus greifen kann. Der Transferprozess allein verbraucht bereits einen signifikanten Teil der verfügbaren Systemressourcen, so dass das Clipping höchstens einen Tropfen auf dem heißen Stein darstellt.

Durch mathematisch geometrische Berechnungen, können Polygone und Polygon-Gruppen, die sich nicht innerhalb des Sichtkegels befinden, allerdings identifiziert werden, bevor der Transfer an die Grafikkarte erfolgt. Wenn diese Polygone während des Zeichenvorgangs ausgespart werden, ist ein relevanter Leistungszuwachs möglich.

Realitätsnahe 3D-Modelle bestehen aus relativ vielen Polygonen. In Video-Spielen kann deren Zahl pro Objekt durchaus im Bereich von Hunderten bis mehreren Tausend angesiedelt sein. Um exakt zu testen, ob ein Objekt innerhalb des Sichtkegels positioniert ist, müsste jedes einzelne dieser Polygone mathematisch darauf untersucht werden, ob es sich im Frustum befindet oder nicht. Es ist leicht einzusehen, dass die notwendigen Kalkulationen

einen Overhead erzeugen würden, der den Performance-Gewinn durch das Frustum-Culling nihiliert oder gar ins Negative verkehrt.

Der Lösungsweg zum wirkungsvollen Einsatz der Frustum-Culling-Technik liegt offensichtlich in der Reduktion der Sichtkegel-Tests. Der Performance-Flaschenhals kann überwunden werden, in dem nicht mehr jedes einzelne Polygon auf den Frustum getestet wird, sondern statt dessen ein unsichtbares geometrisches Hüll-Objekt, welches das 3D-Modell vollständig umfasst. Nach Möglichkeit sollte das Hüll-Objekt, oder besser Hüll-Volumen genannt, annäherungsweise der Form des einzuschließenden Objekts ähneln. Dadurch lässt sich der Sichtbarkeitstest präziser durchführen. Als Hüll-Volumina werden in der Regel Kugeln, Quader oder Cylinder bemüht.

Jedes Modell, dessen Sichtbarkeits-Filterung durch Frustum-Culling effizient bewerkstelligt werden soll, benötigt ein Hüll-Volumen, das es eigens zu berechnen gilt. Ist eine solche Hülle gefunden, kann durch volumenspezifische Schnittberechnungen mit dem Kamera-Frustum entschieden werden, ob sich ein Objekt im Sichtkegel befindet. Fällt der Test negativ aus, können im Render-Prozess die Polygone des Modells bewusst vernachlässigt werden. Die Abbildung 17 erläutert den Sachverhalt grafisch. Der braune Kegel stellt den Kamera-Frustum in einer vereinfachten Draufsicht dar. Die Grafik zeigt das potenzielle Verhalten beim Frustum-Culling. Die Szene besteht aus 3D-Objekten (hier piktografisch Häuser und Menschen), die von einem sie umgebenden Hüll-Volumen eingeschlossen werden. Objekte mit rot dargestelltem Hüll-Volumen liegen außerhalb des Sichtkegels und spielen im Render-Prozess folglich keine Rolle. Die Objekte mit schwarzem Hüll-Volumen liegen innerhalb des Sichtkegels und sollten deshalb gezeichnet werden.

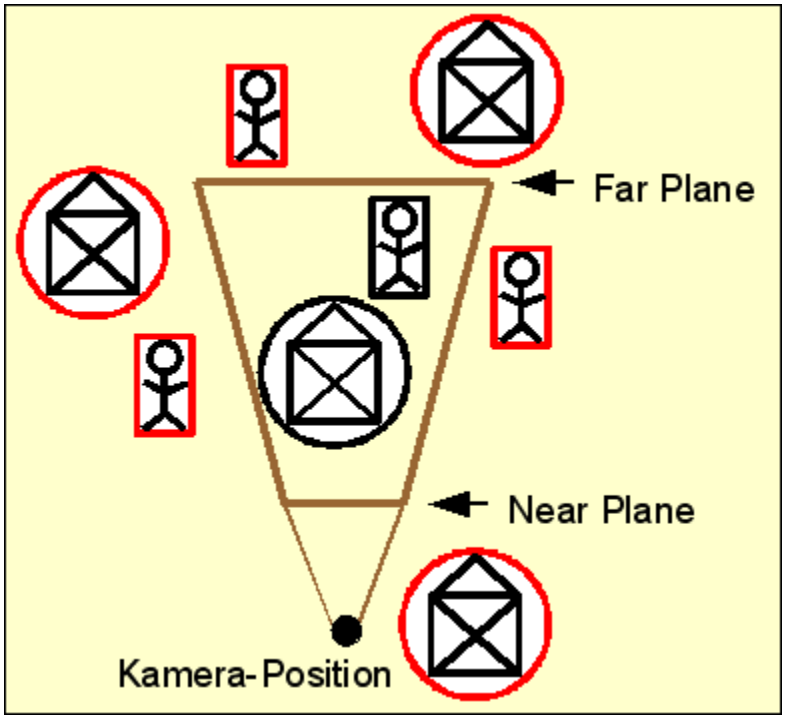


Abbildung 17: Sichtbarkeits-Filterung durch Frustum-Culling

Die Szene aus der obigen Abbildung enthält acht Objekte, zwei davon liegen innerhalb des Kamera-Sichtbereichs. Angenommen jedes dieser Objekte bestünde aus 1000 Polygonen.

Insgesamt enthielte die Szene also 8000 Polygone. Wollte man nun das Frustum-Culling auf Polygonbasis durchführen, müssten 8000 einzelne Polygone auf ihre Position zum Sichtkegels hin untersucht werden. Und das für jeden einzelnen Durchlauf der Zeichenschleife. Der rechnerische Aufwand wäre als äußerst kontraproduktiv für die Laufzeitgeschwindigkeit des Programms einzustufen. Durch die Einführung der Hüll-Volumina ließe sich die Anzahl der notwendigen Tests auf Acht herab senken. Der Aufwand könnte in diesem Fall somit um den Faktor 1000 verringert werden.

Die Bilanz der Frustum-Culling-Technik wird durch leichte Test-Ungenauigkeiten, verursacht durch gegebenenfalls nicht völlig passgerechter Volumina-Formen, nur unwesentlich geschmälert.

Zur weiterführenden Performance-Steigerung wird gerne eine zusätzliche Technik in Kombination mit dem Frustum-Culling angewendet. Sie wird Occlusion-Culling genannt und im Folgenden kurz behandelt.

Occlusion-Culling

Das Ziel von Culling-Techniken ist stets das Aussortieren von Polygongruppen, die sich nicht im Blick der Kamera befinden. Das zuvor betrachtete Frustum-Culling leistet hierzu gute Dienste. Auf den zweiten Blick fällt jedoch auf, dass nicht unbedingt alle unsichtbaren Objekte ausgesiebt werden. Durch das Frustum-Culling werden nur Objekte gezeichnet, die sich innerhalb des Kamera-Sichtkegels befinden. Es ist aber denkbar und durchaus nicht selten, dass ein Modell im Sichtkegel liegt, aber dennoch optisch nicht wahrnehmbar ist, weil es von einem anderen Objekt verdeckt wird. Die Frustum-Culling-Technik ist in diesem Fall unbrauchbar. Es würden Polygone auf dem Bildschirm dargestellt werden, deren Zeichenvorgang im Grunde als überflüssig anzusehen ist.

Mit der Occlusion-Culling-Technik (von engl. *to occlude*: *verdecken*) existiert jedoch ein Verfahren, dass die Unzulänglichkeit des Frustum-Cullings kompensiert. Beim Occlusion-Culling werden die Polygone ausgesondert, die unsichtbar sind, weil sie von anderen Polygonen verdeckt werden. Befindet sich beispielsweise eine Spiel-Figur hinter einem undurchsichtigen Gebäude oder einer hohen Mauer, die allerdings im Kamera-Frustum positioniert ist, so lässt sich mittels Verdeckungstests feststellen, dass die Figur nicht zu sehen ist. Daher ist es überflüssig, Ressourcen in ihren Renderprozess zu investieren. Abbildung 18 zeigt das Ergebnis von Occlusion-Culling. Die Figur mit dem rot dargestellten Hüll-Volumen wird aussortiert und nicht gezeichnet, weil sie von der Mauer verdeckt wird. Die Figur mit der schwarzen Umrandung steht vor der Mauer und ist deshalb sichtbar. Sie muss gezeichnet werden.

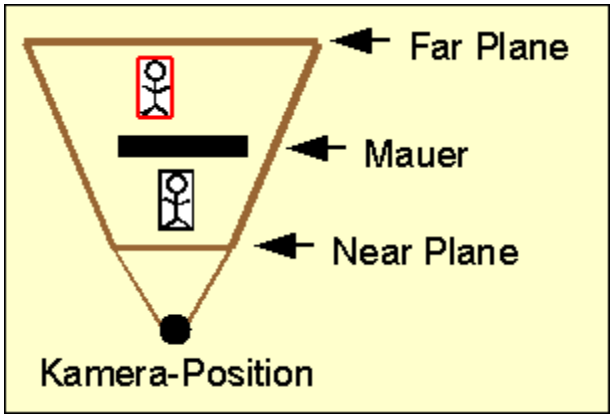


Abbildung 18: Sichtbarkeits-Filterung durch Occlusion-Culling

Das Occlusion-Culling wird in der Regel in Kombination mit dem Frustum-Culling eingesetzt. Beide Verfahren ergänzen sich schließlich gut. In einem ersten Schritt werden alle 3D-Modelle herausgefiltert, die sich außerhalb des Kamera-Frustums befinden. Anschließend werden die verbliebenen Modelle auf Verdeckung hin untersucht und, falls nötig, ebenfalls aussortiert.

Occlusion-Culling kann auf mehrere Weisen in einer Engine umgesetzt werden. Zum einen gibt es das mathematisch manuelle Verfahren, dass jedoch recht kompliziert ist. Zum anderen bietet OpenGL seit der Version 1.5 offiziell eine integrierte Erweiterung an, die Verdeckungstests relativ komfortabel gestattet. Letztere wird in der 3D-Engine wegen ihrer Effektivität und Einfachheit eingesetzt werden. Es ist aber zuvor zu überprüfen, ob die Erweiterung von der verwendeten Grafik-Hardware unterstützt wird. Ist dies nicht der Fall, muss auf die Leistungsoptimierung durch Occlusion Culling verzichtet werden. Die Implementierung des Occlusion-Cullings in die Engine wird in diesem Dokument nicht beschrieben werden, da sie lediglich auf der Anwendung einiger einfacher OpenGL-Operation beruht. Implementierungsdetails und Hintergrundinformationen können in [Wright2004] nachgelesen werden.

4.1.2.6 Effiziente Kollisionserkennung

In den Anforderungen wurde erwähnt, dass Techniken zur Kollisionserkennung unvermeidlich sind, möchte man realistisch wirkende Spielwelten schaffen. Für eine präzise Erkennung von Objektzusammenstößen ist es nötig, die Polygone der beteiligten Objekte auf Kollision zu untersuchen. Bei einfachen Modellen mit wenigen Polygonen mag dies ein gangbarer Weg sein. Nimmt die Polygonzahl der verwendeten Objekte allerdings zu, stellen sich massive Leistungseinbrüche ein, denn die Überprüfung aller Polygone auf Zusammenstöße kann sehr aufwändig werden. Dies kann darin gipfeln, dass eine angenehme Bildwiederholungsrate nicht mehr gewährleistet werden kann und ein flüssiger Spielablauf in unspielbares Stocken überführt wird.

Damit Kollisionserkennungen dennoch in Echtzeit durchführbar sind, muss zu einem gewissen Grad auf Präzision verzichtet werden. Anstelle der eigentlichen 3D-Modelle lassen sich auch ihre Hüll-Volunina auf Kollision testen. Dies ist durchaus zu vergleichen mit den Sichtbarkeitstests im Zusammenhang mit dem Kamera-Frustum (Kapitel 4.1.2.5). Einerseits

weisen modellumspannende Hüll-Volumina einfache Geometrien auf, die sich hochperformant auf Überschneidungen testen lassen. Andererseits könnten aufgrund ihrer Ausdehnung, Kollisionen zwischen Hüll-Volumina festgestellt werden, obwohl die zugehörigen Objekte sich eigentlich noch nicht schneiden. Abbildungen 19 verdeutlicht die unpräzise Erkennung von Kollisionen bei unpassend gewählten Hüll-Volumina (links) und die höhere Genauigkeit bei besser gewählten Passformen (rechts).

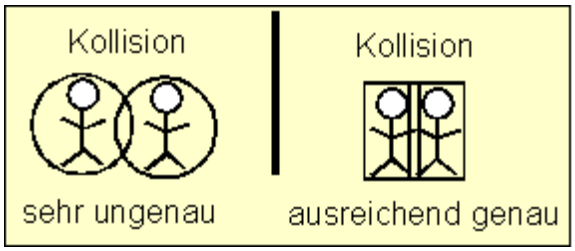


Abbildung 19: Kollisionsgenauigkeit verschiedenartiger Hüll-Volumina

Damit die Kollisionserkennung ausreichend genau bleibt, sollten demnach möglichst eng anliegende Hüll-Volumina verwendet werden. Die Engine wird mindestens die Hüllformen Kugel und Quader unterstützen.

4.1.2.7 Wegfindungssystem

Es existieren verschiedene Ansätze zum Lösen von Wegfindungsproblemen. Das Design der Engine sieht vor, einen der mächtigsten und beliebtesten Algorithmen zu benutzen, um 3D-Objekte von Punkt A zu Punkt B zu führen, ohne dabei Hindernisse außer acht zu lassen: Den heuristikbasierten A*-Algorithmus. A* wird seit 1968 für die Lösung verschiedenster Probleme im Bereich der Wegfindung angewandt [DeLoura2002] und gilt als de facto Standard in der Videospiele-Industrie. Er ist also vermutlich der meist eingesetzte Wegfindungs-Algorithmus in modernen Computer-Spielen. Er ist so attraktiv, weil er garantiert den kürzesten Weg zwischen zwei beliebigen Positionen finden kann, falls er denn existiert. Außerdem ist er im Verhältnis zu seiner Mächtigkeit noch relativ ressourcenschonend [Bourg2004].

Voraussetzungen zum Verständnis von A*

Das besondere am A*-Algorithmus ist seine Eigenschaft, die Kosten bestimmter Wegpunkte in die Suche mit einzubeziehen. Manche Positionen in der Spielwelt (im Umfeld des Algorithmus als Knoten bezeichnet) können mit höheren Passierungskosten verbunden werden, als andere. Zum Beispiel wird das Fortbewegen durch eine matschige Sumpflandschaft deutlich beschwerlicher sein, als das Laufen auf einer befestigten Straße. Je höher die Passierungskosten für einen Knoten ausfallen, desto eher sollte der Suchalgorithmus es vermeiden, ihn in den resultieren Pfad aufzunehmen.

Damit A* funktionieren kann, muss die Spielwelt in eine Menge von Knoten aufgeteilt werden, in der der Algorithmus bei Bedarf nach möglichen Wegen suchen kann. Die Knoten könnten manuell in der Spielwelt platziert werden. Jedoch wäre dies ein sehr unkomfortabler Ansatz, da es für einen Leveldesigner, je nachdem, wie feinmaschig die Knoten gesetzt werden müssen, viel Arbeit bedeuten kann. Stattdessen wird die Engine ein Gitternetz aus kachelförmigen Knoten zur Verfügung stellen, dessen Dimension und Granularität für ein

Level definiert werden kann. Dieses Gitter wird unsichtbar mit der Boden-Ebene assoziiert. Das Grundgerüst an Suchknoten wird somit automatisch generiert. Die Aufgabe des Leveldesigners besteht dann nur noch darin, die Kosten spezieller Knoten festzulegen.

Für eine große Menge von Spielen ist diese Aufteilung der Bodenfläche durchaus praktikabel, deshalb wird sie als Standard-Lösung in die Engine integriert werden.

Zur Berechnung des kürzesten Weges zwischen zwei Punkten verwendet der A*-Algorithmus eine so genannte *Scoring*-Formel. Sie versieht einen Knoten mit einer Punktzahl, die ausschlaggebend dafür ist, ob der Knoten Teil des Resultats wird. Sie lautet:

$$Score(Knoten) = CostFromStart(Knoten) + CostToGoal(Knoten)$$

Die Funktion *CostFromStart()* liefert die geringsten Kosten vom Startknoten bis zum angegebenen Knoten. *CostToGoal()* stellt einen heuristischen Schätzwert dar, der die Kosten vom angegebenen Knoten zum Zielknoten reflektiert. Es hängt von der Qualität dieser Heuristik ab, ob der Algorithmus tatsächlich den kürzesten Weg zum Ziel berechnet oder gegebenenfalls Umwege in Kauf nimmt [Davison2005].

Funktionsweise von A*

A* verwendet zwei Listen, die für untersuchte (Liste *open*) und nicht untersuchte Knoten (Liste *closed*) gedacht sind. Initial ist die *closed* Liste leer, während die *open* Liste den Startknoten enthält. Der Algorithmus verfällt nun in eine Schleife. Er entfernt in jedem Schleifendurchlauf den Knoten, mit der als am höchsten geschätzten Punktzahl aus der *open* Liste und prüft, ob es sich dabei um den Zielknoten handelt. Fällt der Test negativ aus, werden die Nachbarknoten gemäß ihrer Punktzahl sortiert. Handelt es sich um bisher unerforschte Knoten, werden sie zur Liste *open* hinzugefügt. Befanden sie sich bereits in *open*, wird ihre Punktzahl gegebenenfalls aktualisiert. Andernfalls werden sie ignoriert, da sie bereits der *closed* Liste zugeteilt sein müssen. Es werden nun iterativ alle Nachbarknoten untersucht, bis der Zielknoten gefunden wurde. Sofern es geschehen sollte, dass die *open* Liste leer wird, ohne dass der Zielknoten ermittelt werden konnte, bedeutet dies, dass kein Weg vom Start- zum Zielknoten existiert [DeLoura2002]. Der nachstehende Pseudo-Code beschreibt den A*-Algorithmus genauer (Abbildung 20). Er ist [Davison2005] entnommen worden, der inhaltlich wiederum Bezug auf [DeLoura2002] nimmt.

```
add the start node to open;
create an empty closed list;

while(open isn't empty)
{
  get the highest scoring node x from open;
  if(x is the goal node)
    return a path to x; // path found!
  else
  {
    for(each adjacent node y to x)
    {
      calculate the CostFromStart() value for y;
      if((y is already in open or closed) and
        (value is no improvement))
        continue; // ignore y
      else
      {
        delete old y from open or closed (if present);
        calculate costToGoal() and total score for y;
        store y in open;
      }
    }
  }
  put x into closed; // there's no further use for it
}
report no path found;
```

Abbildung 20: Funktionsweise von A als Pseudo-Code. [Davison2005]*

In der Regel wird der gefundene Pfad vom Start zum Ziel über das Rückschritt-Verfahren (Backstepping) rekonstruiert.

4.1.2.8 Licht- und Schattendarstellung

Dieses Unterkapitel stellt sowohl den Entwurf des für die Engine vorgesehenen Beleuchtungssystems vor, als auch das zur Verfügung zu stellende Schattenverfahren.

Entwurf des Beleuchtungssystems

Das Beleuchtungssystem einer Game-Engine sorgt dafür, dass die Oberflächen der Objekte einer Spielwelt den Lichtverhältnissen entsprechend illuminiert werden. Da die zu entwickelnde 3D-Engine OpenGL als Renderbasis benutzen wird, liegt es nahe, dessen integriertes Lichtmodell zu verwenden.

Die Oberflächenillumination in OpenGL hängt von zwei zentralen Faktoren ab. Zum einem von der Farbe des Lichtes selbst und zum anderen von der Farbe der zu beleuchtenden Oberfläche (Materialfarbe). Aus der Mischung beider Komponenten ergibt sich im Endeffekt die Farbe der beleuchteten Oberflächen.

OpenGL verwendet das RGBA-Farbmodell. RGBA steht akronymisch für die vier Farbbestandteile *red, green, blue, alpha*. Die Intensität jeder Farbkomponente kann in

OpenGL einzeln spezifiziert werden. Der Wertebereich liegt jeweils zwischen 0.0f und 1.0f bei Verwendung von Fließkommazahlen respektive zwischen 0 und 255 im Ganzzahlbereich. Es gilt, je größer der angegebene Wert, desto stärker nimmt die Farbkomponente Einfluss auf das Mischungsverhältnis.

OpenGL erlaubt das Erstellen drei verschiedenartiger Lichtquellentypen: *Directional Lights*, *Positional Lights* und *Spot Lights*. Unter *Directional Lights* versteht man Lichtquellen, die unendlich weit weg scheinen und paralleles Licht in eine einzige Richtung aussenden (vgl. Sonnenlicht). Da man den *Directional Lights* keine Position in der Welt zuordnen kann, bleibt die Intensität des Lichts stets konstant [Lengyel2004]. Bei den beiden anderen Lichttypen ist dies nicht der Fall.

Positional Lights strahlen ihr Licht von einer spezifizierten Position gleichförmig in alle Richtungen aus. Sie haben die Eigenschaft, dass die Intensität ihrer Strahlen über zunehmende Distanzen gedämpft wird. Die Stärke der Abschwächung kann in OpenGL über verschiedene Dämpfungskonstanten geregelt werden [Lengyel2004].

Spot Lights können als Spezialisierungen von *Positional Lights* betrachtet werden. Sie verfügen ebenfalls über eine Position und ihr Dämpfungsverhalten ist identisch. Der hauptsächliche Unterschied besteht in der Strahlungsrichtung des Lichts. Das Licht eines *Spot Lights* kann auf eine bestimmte Richtung fokussiert werden, so dass sich der bekannte Scheinwerfer-Effekt einstellt, den man mit Bühnenvorstellungen in Verbindung bringt [Lengyel2004]. Abbildung 21 stellt die drei genannten Typen von Lichtquellen bildlich gegenüber.

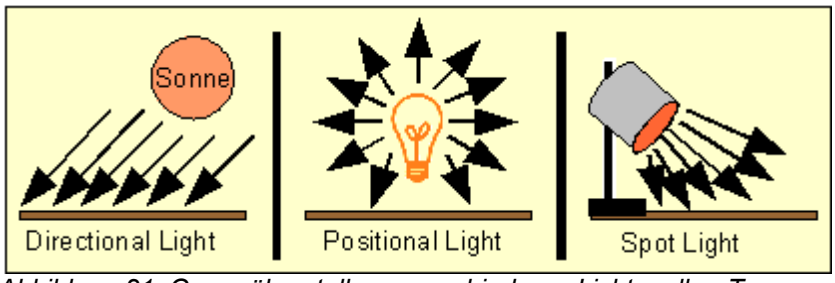


Abbildung 21: Gegenüberstellung verschiedener Lichtquellen-Typen

Berücksichtigt man das Verhältnis der Lichtquellen-Typen zueinander, lässt sich ein strukturelles Gliederungsschema erstellen, das in gleicher Form in der Engine umgesetzt werden wird (Abbildung 22).

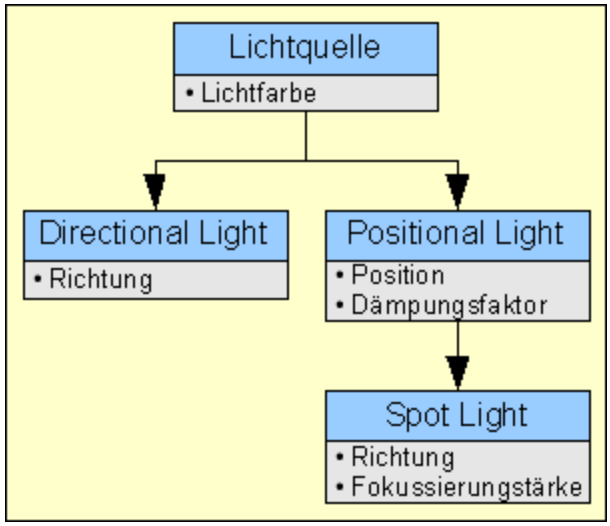


Abbildung 22: Vererbungshierarchie der Lichtquellen-Typen

Die jeweils tiefere Hierarchieebene erbt die Eigenschaften der vorangegangenen Ebene. Dieses Lichtquellen-Schema ist prädestiniert für die Kapselung in einzelne Java-Klassen. Die objektorientierten Vererbungsmechanismen Javas können hier voll zum Tragen kommen.

Echtzeit-Schattenverfahren

Die Computervisualistik hat im Laufe der Zeit einige interessante Verfahren hervorgebracht, um 3D-Szenen mit künstlichen Schattenwürfen in Echtzeit zu bereichern. Drei der bekanntesten sind das *Shadow-Matrix*-Verfahren, die *Shadow-Mapping*-Technik und die so genannten *Shadow-Volumes*. Jede dieser Methoden hat sowohl Vor- als auch Nachteile, die sich für gewöhnlich auf Geschwindigkeit und optischen Realismus beziehen. Die Tabelle in Abbildung 23 stellt sie vergleichend gegenüber.

Shadow-Matrix	Shadow-Mapping	Shadow-Volumes
Vorteile ● sehr schnell ● scharfer Schatten *Nachteile* ● keine Selbstschattierung ● kein Schattenwurf auf andere 3D-Objekte	*Vorteile* ● sehr schnell ● Selbstschattierung ● Schattenwurf auf andere 3D-Objekte *Nachteil* ● grober Schatten → scharfer Schatten i.d.R. auf Kosten der Leistung	*Vorteile* ● scharfer Schatten ● Selbstschattierung ● Schattenwurf auf andere 3D-Objekte *Nachteil* ● sehr langsam

Abbildung 23: Tabellarische Gegenüberstellung von Schattentechniken[26]

Das *Shadow-Volume*-Verfahren liefert optisch die besten Ergebnisse, da es erstens sehr scharfe Schattenkonturen bietet und zweitens seinen Schatten recht realistisch auf die

26 Artikel von Kris Garrein über Schattentechniken - http://www.devmaster.net/articles/shadows/ (Stand: 04.09.2007)

Umgebung wirft. Das bedeutet, er wird eindrucksvoll auf Wände oder andere Objekte der 3D-Szene projiziert. Leider erfordern die notwendigen Berechnungen viel Hardware-Einsatz, so dass dieses Verfahren tendenziell als langsam zu bezeichnen ist. Besonders für dynamische Objekte ist seine Anwendung sehr aufwändig. Nähere Informationen und Implementierungsdetails zu *Shadow-Volumes* sind in [Watt2001] sowie [Lengyel2004] nachzulesen.

Geringere Ansprüche an die Geschwindigkeit der Hardware stellt die *Shadow-Mapping*-Technik auf Basis der Tiefentextur-Projektion. Sie projiziert den Schatten ebenfalls auf andere Objekte der Szene. Allerdings sieht der Schatten nur dann scharf aus, wenn er in eine möglichst große Tiefentextur gerendert wird. Die Verwendung großer Texturen geht jedoch zu Lasten der Systemleistung, so dass die Ausführungsgeschwindigkeit herabgesetzt wird. Eine kleine Tiefentextur ermöglicht eine bessere Geschwindigkeit, führt aber zu artefaktartiger Blockbildung am Rand der Schattenprojektion. Weiterhin werden OpenGL-Extensions benötigt, die erst ab der Version 1.5 verfügbar sind, um die Erzeugung der Tiefentextur hardwarebeschleunigt durchführen zu können. [Wright2004] erklärt die Shadow-Mapping-Technik und demonstriert ihre Anwendung.

Das *Shadow-Matrix*-Verfahren kann in gewisser Weise als Kompromiss zwischen den zuvor genannten Verfahren angesehen werden. Es ist schnell und bietet trotzdem scharfe Schattenkonturen. Der große Nachtteil an dieser Technik ist der geringe visuelle Realismus. *Shadow-Matrix*-Schatten werden ausschließlich auf eine Ebene projiziert. Im Normalfall ist das die Bodenebene. Die optisch reizvolle, da den menschlichen Erfahrungen entsprechende Schattierung anderer Objekte der Umgebung, ist leider nicht gegeben. Zum Beispiel werden Schatten an Wänden einfach abgeschnitten. Berücksichtigt man aber, dass die zu entwickelnde Engine nicht den Ansprüchen einer kommerziellen Lösung gerecht werden muss, stellt das Verfahren einen guten Kompromiss aus Performanz und Ansehnlichkeit dar. Zudem ist es vergleichsweise einfach zu implementieren und soll für die Zwecke des Projekts als ausreichend zu betrachten sein. Prof. Dr. Wolfgang P. Kowalk legt Herleitung, Funktionsweise und Anwendung des *Shadow-Matrix*-Verfahrens in seinem Tutorial[27] zu OpenGL mit Java ausführlich dar.

4.1.2.9 Spezialeffekte optischer und akustischer Natur

Schattendarstellungen können zwar auch zu den grafischen Spezialeffekten gezählt werden, dieses Kapitel befasst sich jedoch mit weiteren Effekten, die nicht direkt von der Position der Szene-Beleuchtung abhängen. Sie werden nachfolgend kurz beschrieben und gehen allesamt in den Entwurf der Engine ein.

Transparenz durch Blending

Transparenz ist ein Effekt, der stark zum Realismus der Szenerie beitragen kann. Glasflächen, wie sie zum Beispiel für Fenster charakteristisch sind, werden erst dann möglich, wenn die Engine in der Lage ist, blickdurchlässige Flächen zu zeichnen. Ebenso verhält es sich mit Wasseroberflächen, die als halb transparente Ebenen dargestellt werden können.

27 JOGL-Tutorial von Prof. Dr. Wolfgang P. Kowalk an der Universität Oldenburg - http://einstein.informatik.uni-oldenburg.de/forschung/opengl/doku/Jogln-Tutorial.pdf (Stand: 05.09.2007)

Das Schlüsselwort in Zusammenhang mit Transparenz in der OpenGL-Terminologie lautet *Blending*. Unter *Blending* versteht man das bereichsweise Verrechnen der Farben von Szene-Objekten, die sich entweder ganz oder teilweise verdecken beziehungsweise verdeckt werden. Angenommen, ein als halb transparent deklarierter Körper verdeckt ein anderes Objekt, vom Blickwinkel der Kamera aus gesehen. Damit das verdeckte Objekt wahrgenommen werden kann und der verdeckende Körper somit transparent erscheint, wird die Farbe des transparenten Körpers in den Bereichen, in denen das andere Objekt durchscheinen soll, mit der Farbe des verdeckten Objekts kombiniert. Der sich ergebene Farbunterschied führt zur Illusion der Transparenz. OpenGL erlaubt es, die Vorschrift, nach der die Farbverrechnung erfolgen soll, manuell festzulegen, so dass unterschiedliche Ergebnisse erzielt werden können.

Damit die Illusion durch ungünstige Verdeckungsreihenfolgen nicht getrübt wird, ist es nötig, durchsichtige und undurchsichtige Objekte getrennt voneinander zu zeichnen. Korrektes Transparenzverhalten wird gewährleistet, wenn eingangs die undurchsichtigen Körper und erst daraufhin die transparenten Szene-Komponenten gerendert werden.

Nebel

Der Einsatz von Nebel-Effekten kann in Video-Spielen vielerlei Gründe haben. In der Regel stechen jedoch zwei Anwendungsfälle hervor. Zum einen kann Nebel als Stilmittel eingesetzt werden und dadurch entscheidend zur Spielatmosphäre beitragen. Zum anderen wird er gerne aus leistungsstrategischen Erwägungen heraus implementiert, weil er auf natürliche Weise die Sichtweite des Spielers begrenzt und Objekte, die im Nebel liegen, nicht gezeichnet werden müssen.

OpenGL bietet eine integrierte Nebelfunktionalität, die leicht anzuwenden ist und sich intern ebenfalls des Blendings bedient. Verschiedene Einflussfaktoren, wie Farbe, Dichte, Anfang und Ende des Nebels erlauben eine relativ individuelle Gestaltung. Die Kapselung in eine objektorientierte Java-Klasse ist auf unkomplizierte Weise möglich.

Kameralinsen-Effekt (Lens Flare)

Als Kameralinsen-oder Lens-Flare-Effekt bezeichnet man ein Phänomen, dass ursprünglich aus der Fotografie bekannt ist. Der Linsen-Effekt wird durch die Streuung und Reflexion einfallenden Lichts im Linsensystem einer Kamera verursacht. Er äußert sich als verschieden förmige Lichtmuster im Kamera-Bild (Abbildung 24).

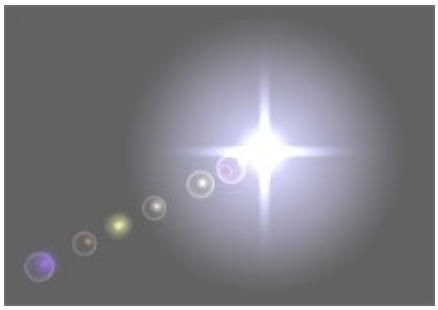


Abbildung 24: Darstellung eines Lens-Flare-Effekts

Die Verwendung von Lens Flares in einer 3D-Szene kann unter Umständen dazu beitragen, den Eindruck der Kamera-Analogie zu verstärken und die Illusion einfallenden Sonnenlichts zu unterstützen.

Technisch kann der Linsen-Effekt in Video-Spielen durch das Zeichnen mehrerer quadratischer Flächen simuliert werden, welche mit Texturen zu versehen sind, die die Bestandteile der Lichtmuster repräsentieren. Auch hier kommt Blending zum Einsatz, da Teile der texturierten Flächen vollkommen durchsichtig gemacht werden müssen.

Environment-Mapping

Unter *Environment-Mapping* versteht man eine Texturierungstechnik, die für die Erzeugung von Pseudo-Spiegelungen auf einem 3D-Modell verwendet wird. In Video-Spielen findet sie häufig auf Objekten Anwendung, die ein gewisses metallisches Aussehen haben sollen. Insbesondere Rennspiele nutzen diese Technik, um ihren Fahrzeugen einen ansehnlichen Lackeffekt zu verleihen.

Zwei von OpenGL konkret unterstütze Varianten des *Environment-Mappings* sind das *Sphere*- und und das *Cube-Mapping*. Beim *Sphere-Mapping* wird die zu reflektierende Umgebung in Form einer einzigen Textur auf ein 3D-Modell projiziert, während das *Cube-Mapping* die Umgebung gar durch sechs Texturen spiegelt. Beide Verfahren setzten dabei auf die automatische Berechnung von Texturkoordinaten.

Durch die Kapselung des Effekts in spezialisierte Textur-Klassen, ist es möglich, diese Technik nahtlos in den herkömmlichen Texturierungsprozess zu integrieren.

Partikel-System

Partikel-Systeme kommen für gewöhnlich dann zum Einsatz, wenn es um die Darstellung von Funken, Flammen, Explosionen, Dampf, Rauch oder Regen geht. Grundstein eines jeden Partikel-Systems sind seine Partikel. Sie können je nach Komplexität und Mächtigkeit des Systems eine Fülle von Eigenschaften besitzen, über die Einfluss auf ihr Verhalten genommen werden kann. Dazu zählen insbesondere Position, Bewegungsrichtung und Geschwindigkeit der Teilchen. Komplexere Systeme simulieren unter anderem auch die Auswirkungen von Schwer- und Anziehungskräften, die in der Welt wirken. Partikel können auf unterschiedlichsten Strukturen beruhen. Oftmals werden sie als texturierte, farbige und halb transparente Vierecke realisiert. Die Verwendung von Punkten und Linien als Grundform ist aber ebenfalls sehr verbreitet.

Das Partikel-System für die zu entwerfende Engine wird aus drei wesentlichen Komponenten bestehen: Dem Partikel-Effekt, einem Partikel-Container und dem Partikel selbst. Der Anwender der Engine wird für jede Art von Partikel-Effekten genau festlegen können, wie sich das Teilchen während seiner aktiven Phase zu verhalten hat. Der Partikel-Container hat die Aufgabe, die enthaltenen Partikel zu verwalten. Er löscht Partikel, deren festgelegte Lebensspanne abgelaufen ist und aktualisiert den Zustand (z.B. Position, Rotation, Alter, Farbe) der noch aktiven Partikel. In diesem Partikel-System werden die Partikel hauptsächlich auf der Methode der texturierten Vierecke basieren. Gegebenenfalls können noch alternative Techniken implementiert werden. Ein entsprechend abstraktes Klassen-Design vorausgesetzt, sollten sich weitere Verfahren nahtlos eingliedern lassen. Jedes

Partikel wird eine Methode zum Rendern enthalten, die von der zentralen Partikel-Effekt-Einheit benutzt wird, um den Effekt zu visualisieren. Die folgende Abbildung 25 illustriert den Entwurfsaufbau des Partikel-Systems in schematischer Form.

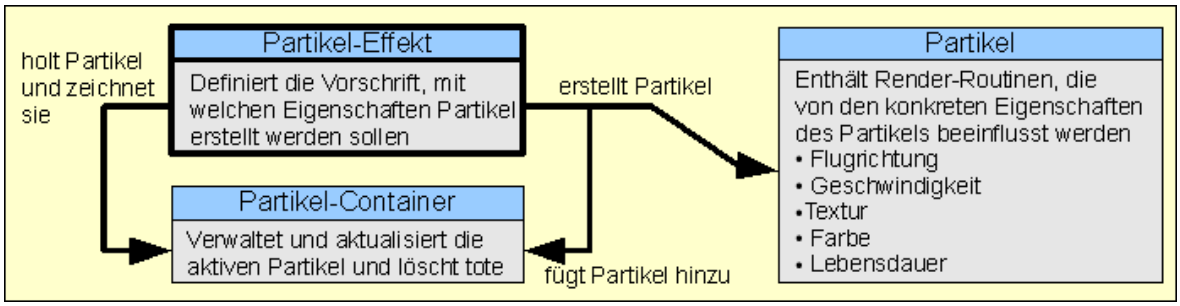


Abbildung 25: Schematisches Design des Partikel-Systems

Audio-Funktionalität

Für eine Video-Spiele-Engine ist es äußerst wichtig, Audio-Dateien abspielen zu können. Die Spielerfahrung kann dadurch um eine zusätzliche multimediale Dimension der Wahrnehmung erweitert werden. Ganz gleich, ob diese Fähigkeit zum Abspielen einer begleitenden Hintergrundmusik, atmosphärischer Geräuschkulissen oder für Sprachausgaben genutzt wird; das individuelle Spielerlebnis wird mit Sicherheit davon profitieren.

Audio-Dateien können in unterschiedlichen Formaten vorliegen. Da das Ton-Material eines Spiels zu enormen Datenmengen heranwachsen kann, ist es geboten, komprimierte Audio-Dateien einzusetzen. Das MIDI- und Mp3-Format erfreuen sich diesbezüglich großer Beliebtheit. Die Engine wird diese beiden Formate in eigene Klassen kapseln, jedoch auch unkomprimierte Audio-Samples, wie WAV-Files, abspielen können. Der Umgang mit den Dateien verschiedenen Typs wird über die Formate hinweg identisch sein, so dass das Sound-System wie aus einem Guss erscheint und nicht unnötig verkompliziert wird. Erreicht werden kann dies durch namentlich und funktional identische Methoden in den verschiedenen Kapselungsklassen.

4.1.2.10 Orthographische Text- und Grafik-Projektion

Kaum ein modernes Videospiel kommt ohne visuelle Benutzerschnittstelle aus. Diese Schnittstelle zwischen Spiel und Spieler ist in der Regel eine Komposition aus grafischen und textuellen Bestandteilen. Sie reicht von einfachen Textfeldern, über Auswahlmenüs bis hin zu komplexen grafischen Strukturen. Allen gemein ist dabei, dass sie durch die Technik der orthographischen 2D-Projektion quasi über das 3D-Bild der Szene gelegt werden (Text- und Bild-Overlay). Die Engine wird eine Reihe von grundlegenden Klassen zur Verfügung stellen, mit denen sich durchaus zweckdienliche Oberflächen in Baustein-Manier komponieren lassen. Ein abstrakter Klassen-Entwurf begünstigt die spätere Ergänzung weiterer Komponenten zum Fundus. Abbildung 26 stellt die Elemente vor, die das System zum Erstellen von Benutzerschnittstellen, wie zum Beispiels HUDs (siehe Kapitel 4.1.1.12), bereit halten wird.

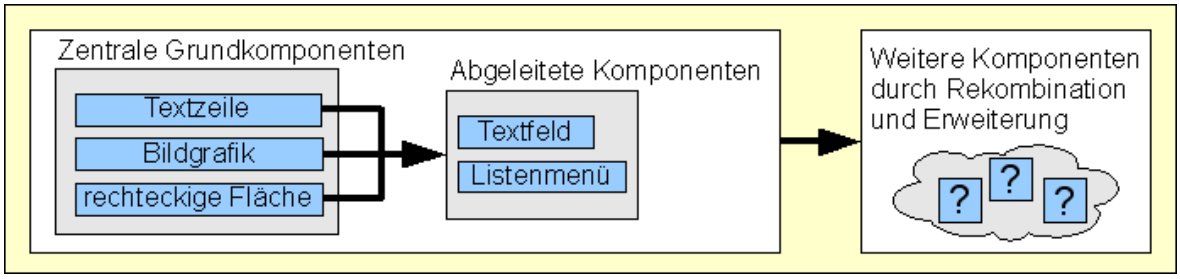


Abbildung 26: Baustein-Modell zum Entwurf von Benutzerschnittstellen

Die fünf vorgefertigten Elemente im linken weißen Kasten können dazu benutzt werden, weitere Overlay-Objekte zu erstellen. Dies ist durch Rekombination der Teile, aber auch durch Erweiterung bestehender Komponenten denkbar.

4.1.2.11 Ein System zur Registrierung von Benutzereingaben

Spiele werden interaktiv, wenn ein Spieler aktiv ins Spielgeschehen eingreifen kann. In aller Regel geschieht dies durch Kommandos, die über periphere Eingabegeräte an das Spiel übermittelt werden. Die Game Engine wird ein solches Verhalten unter Zuhilfenahme eines Input-Systems unterstützen. Dieses Input-System dient zur Spezifikation und Verarbeitung von Benutzereingaben. Für den Rahmen dieses Projekts wird es ausreichend sein, wenn ausschließlich Tastatur- und Mauseingaben vom Input-System behandelt werden können. Ein breites Spektrum von Genres wird auf dem PC gemeinhin mit diesen Eingabegeräten in Verbindung gebracht. Selbst Rennspiel-Simulationen, die gerne auf angeschlossene Lenkradsysteme zurückgreifen, lassen sich notfalls über die Tastatur steuern.

Die Behandlung von Tastatur- und Mauseingaben erfolgt in Java bekanntermaßen mit Hilfe des AWT-Event-Modells und der Implementierung verschiedener Listener-Interfaces (`KeyListener`, `MouseListener`, `MouseMotionListener` und `MouseWheelListener`). Das Input-System wird die notwendigen Operationen kapseln und weitere Funktionalitäten bereitstellen. Dazu gehört unter anderem die Eigenschaft, gegebenenfalls einen Mouse-Look-Modus, wie er insbesondere von First-Person-Games praktiziert wird, zu aktivieren. Zwei Teile wird das System umfassen: Eine Komponente zur Kapselung der spezifischen Eigenschaften einer Aktionsmöglichkeit (beispielsweise Name der Aktion, zugewiesene Keyboard- oder Maus-Taste) und eine Manager-Komponente, die den Ereignisbehandlungscode der Listener-Interfaces kapselt und in Zusammenhang damit, die Aktionsmöglichkeiten verwaltet.

4.1.3 Implementierung

Das Kapitel der Implementierung befasst sich mit der konkreten Umsetzung der 3D-Game-Engine auf Basis der Programmiersprache Java, unterstützt durch die JOGL-API. Da die Implementierung der Engine durch insgesamt etwa 150 Java-Klassen erfolgte, ist es verständlich, wenn nicht auf jeden Aspekt und jede Funktionalität eingegangen werden kann. Stattdessen beleuchtet dieses Kapitel - nicht ausschließlich, aber in erster Linie - die Segmente, die eng mit den Anforderungen und dem Entwurf korrelieren.

4.1.3.1 Elementare Datenstrukturen

OpenGL-Funktionen erwarten häufig Eingabewerte, wie Koordinaten zur Spezifizierung einer Position im 3D-Raum oder eines Polygon-Vertexes. Aber auch Farbwerte sind in verschiedenen Situationen anzugeben. In der Regel sind solche Parameter als separate Ganz- oder Fließkommazahlen sowie Arrays zu übermitteln. Die Kapselung der Werte in Java-Objekte vereinfacht den Umgang mit ihnen, weil spezielle Methoden zur Verfügung gestellt werden können, die einen anwendungsspezifischen Zugriff gestatten. Des weiteren wird die Übersichtlichkeit gesteigert, da ein umständliches hantieren mit unstrukturierten Variablen und Arrays entfällt. Die drei folgenden Unterkapitel beschreiben kurz den Aufbau der Klassen, die elementare Datenstrukturen für den parametrisierten Aufruf grundlegender OpenGL-Funktionen bereithalten.

Repräsentation eines Punktes/Vektors im Raum

Punkte beziehungsweise Raumvektoren sind die substanzielle Grundkomponente dreidimensionaler Grafik-Programmierung. Ihr Anwendungsgebiet reicht von einfachen Positionsangaben über vektorbasierte Operationen bis hin zur Repräsentation von Polygon-Vertizes.

Für die Engine habe ich eine Klasse implementiert, die über die einfache Kapselung dreier Raumkoordinaten weit hinaus geht. Die Klasse `Point3D` verfügt über eine Menge wichtiger Methoden, die im Bereich der 3D-Vektor-Geometrie anzusiedeln sind (siehe Kapitel 2.3). Sie kommen in der Engine an vielen Stellen zum Einsatz. Da die Klasse `Point3D` insbesondere für die Vertizes der 3D-Modelle benötigt wird, speichert sie außerdem die Daten einer Normalen sowie vertexspezifische Textur-Koordinaten.

Die `Triangle`-Klasse: Grundlage aller polygonalen Körper

Die Engine setzt vollkommen auf die Verwendung von Dreiecken als Basispolygon für 3D-Objekte. Da Dreiecke durch drei einzelne Punkte aufgespannt werden, lag es nahe, die Klasse `Triangle` im Prinzip als Container für drei `Point3D`-Objekte zu konzipieren, die zusammen ein Polygon bilden sollen. Eine Reihe von `get()`- und `set()`-Methoden ermöglichen den nachträglichen Zugriff auf die Vertex-Informationen. Wenn gewünscht, berechnet die Klasse automatisch die Normalen der Eckpunkte bei der Erstellung eines Dreieck-Objekts.

Eine Klasse zur Kapselung von Farbwerten

Die Klasse `Color4f` kapselt die vier Farbwerte des RGBA-Systems (rot, grün, blau, alpha). Als reine Kapselungs-Klasse stellt sie nur `get()`- und `set()`-Methoden für den Zugriff auf das Farbobjekt bereit. Verwendet werden ihre Objekte allerdings an den verschiedensten Stellen der Engine. Das Beleuchtungssystem ist zum Beispiel auf Farbwerte angewiesen, ebenso wie die Materialeigenschaften der 3D-Modelle, aber auch die orthographische Projektion von Text und Grafik bedient sich dieser Klasse.

4.1.3.2 Integration von 3D-Modellen in die Game-Engine

Hauptaufgabe von 3D-Anwendungen ist im Normalfall das Zeichnen von Polygonen auf dem Bildschirm. Werden Polygone zu einem Objekt zusammengefasst, spricht man von einem 3D-Modell oder *Mesh*. Die Engine unterscheidet konkret zwischen statischen und animierten Meshes, die durch verschiedene Klassen repräsentiert werden. Da sie aber viele Gemeinsamkeiten aufweisen, habe ich eine abstrakte Oberklasse (`AbstractMesh`) eingeführt, welche alle Attribute und Methoden enthält, die beide Ausprägungen verbindet. Dazu gehören Operationen zur räumlichen Transformation eines Modells, Manipulation der Materialeigenschaften oder auch die Zuweisung von Texturen für die erste und zweite Textur-Einheit.

Das hierarchische Design der Mesh-Klassen zahlt sich schnell aus, wenn man Java-Klassen für spezialisiertere Formen statischer beziehungsweise animierter Objekte einführen möchte. Exemplarisch seien die Klassen zur Darstellung von Bodenebenen (`GroundPlane`) und heightmap-basiertem Gelände (`Terrain`) herausgestellt, die zwar spezialisierten Code enthalten, aber zusätzlich über alle Fähigkeiten ihrer Vater-Klassen verfügen. Es ist also zum Beispiel nicht mehr nötig, Code zur Transformation oder Texturierung eines `Terrain`-Objekts zu schreiben, weil er in der Oberklasse `AbstractMesh` bereits existiert. Dies ist eine besonders elegante Form des Code-Reuse und vereinfacht zudem die Erweiterung des Systems um neue Funktionalitäten.

Die Gabelung der 3D-Objekte, ausgehend von der Klasse `AbstractMesh`, erfolgt in einem ersten Schritt über die Klassen `MeshDL` und `MeshVA`. `MeshDL` wird ausschließlich für statische Modelle verwendet. Sie benutzt OpenGLs *Display-List*-Technik für das performante Rendern polygonaler Körper. Animierte Objekte werden auf Basis der Klasse `MeshVA` gebildet. Sie speichert die verschiedenen Frames der Animationen in `Triangle[]`-Arrays, so dass sie durch OpenGLs *Vertex-Array*-Technik gezeichnet werden können.

Die Engine integriert eine Reihe von Model-Loadern, die überwiegend freien externen Quellen entnommen und für dieses Projekt aufbereitet wurden. Der Loader für 3ds-Modelle, basierend auf dem *joglutils*-Projekt[28], leitet von `MeshDL` ab, da nur statische Modelle mit ihm geladen werden können. Die beiden anderen verfügbaren Model-Loader erweitern die `MeshVA`-Klasse, weil sie des Verarbeitens animierter 3D-Modelle mächtig sind. Zum einen handelt es sich um einen Loader, der Model-Dateien einlesen kann, die im MilkShape3D-ASCII-Format vorliegen. Der ursprüngliche Java-Code hierzu, der einige Anpassungen an die Game-Engine erfahren hat (insbesondere einen Port von LWJGL zu JOGL), stammt von Nate Johnson[29]. Zum anderem verfügt die Engine über einen MD2-Loader, der in seinen Grundzügen von Malte Mathiszig entwickelt, nun aber neu strukturiert und für diese Engine bearbeitet wurde.

Im nachfolgenden UML-Klassendiagramm (Abbildung 27) wird die ganze Vererbungshierarchie der einzelnen Mesh-Klassen untereinander deutlich.

---

28 Das joglutils-Projekt, Quelle des 3ds-Model-Loaders - https://joglutils.dev.java.net/

29 Quelle des MilkShape3D-ASCII-Loaders von Nate Johnson - http://mypage.iu.edu/~natjohns/najgl/ (Stand: 09.09.2007)

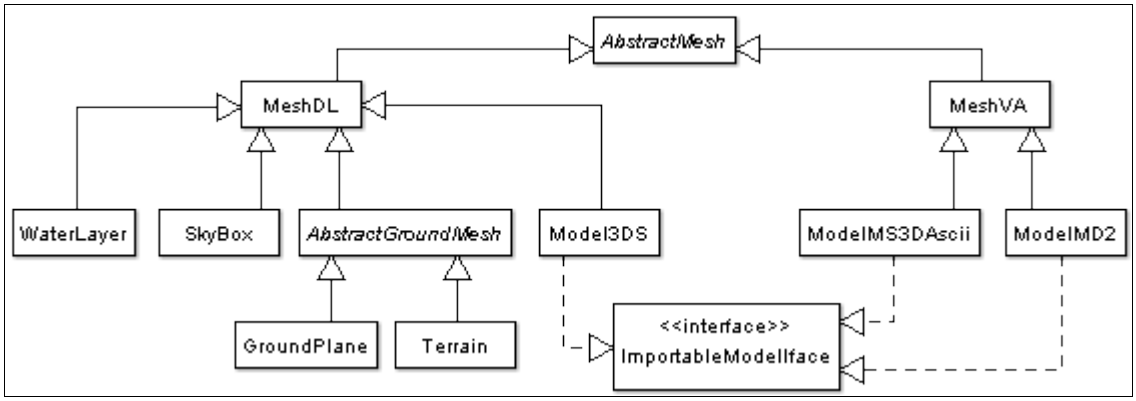


Abbildung 27: Vererbungsbeziehungen der Mesh-Klassen

4.1.3.3 Implementierung des Kamera-Systems

Das Kamera-System der Engine arbeitet nach dem Prinzip anwendungsweit nur ein einziges Kamera-Objekt zu benötigen. Diesem können allerdings verschiedene vordefinierte Kamera-Perspektiven, Objekte der Klasse `CameraPerspective`, zugewiesen werden. Bei Bedarf lassen sie sich über einen Methodenaufruf auswählen und aktivieren. Die Hauptkomponente des Kamera-Systems ist die Klasse `Camera`. Dem genannten Prinzip folgend, implementiert sie das Singleton-Pattern. Das Singleton-Pattern gehört zur Gruppe der Software-Entwurfsmuster. Seine Anwendung auf eine Klasse stellt sicher, dass stets nur *eine* Instanz dieser Klasse existieren kann. Auf dieses Objekt kann jedoch global, von jedem Ort des Programms aus, zugegriffen und dessen Operationen ausgeführt werden [Freeman2004]. `Camera` verfügt unter anderem über Methoden zum Verändern der Kamera-Position und -Rotation. Dadurch wird es möglich, die Kamera durch die 3D-Szene zu bewegen und die Spielwelt fließend aus unterschiedlichen Blickwinkeln zu betrachten. Durch die assoziierten Klassen `CameraViewPort` und `ViewingFrustum` wird einerseits der OpenGL-Viewport und anderseits das Kamera-Frustum mit seinen Clipping-Planes gekapselt.

Ein besonderes Feature des Kamera-Systems ist die Fähigkeit, Kamerabewegungen aufzuzeichnen und wiederzugeben. Es können beliebige Kameraflüge durch die Spielwelt aufgenommen und in eine beständige Datei gespeichert werden. Bei Bedarf lässt sich die Datei laden und der Kameraflug als automatisierte Kamerabewegung abspielen. Für die Aufzeichnung ist die Klasse `Recorder` implementiert worden. Sie registriert, mit einer zuvor festgelegten Frequenz, die gegenwärtige Orientierung der Kamera (Position, Rotation). Die Werte werden in ein Objekt der Klasse `Orientation` gekapselt und anschließend als neuer Checkpoint einem `CameraFlight`-Objekt hinzugefügt. Ist es gewünscht, kann der vollständige Kameraflug durch Javas Serialisierungsmechanismus in ein persistentes Format überführt werden. Wiedergeben lässt sich das (deserialisierte) `CameraFlight`-Objekt mit der `CamPlayer`-Klasse. Der Zusammenhang der Klassen des Kamera-Systems wird noch einmal im nachfolgenden UML-Diagramm (Abbildung 28) verdeutlicht.

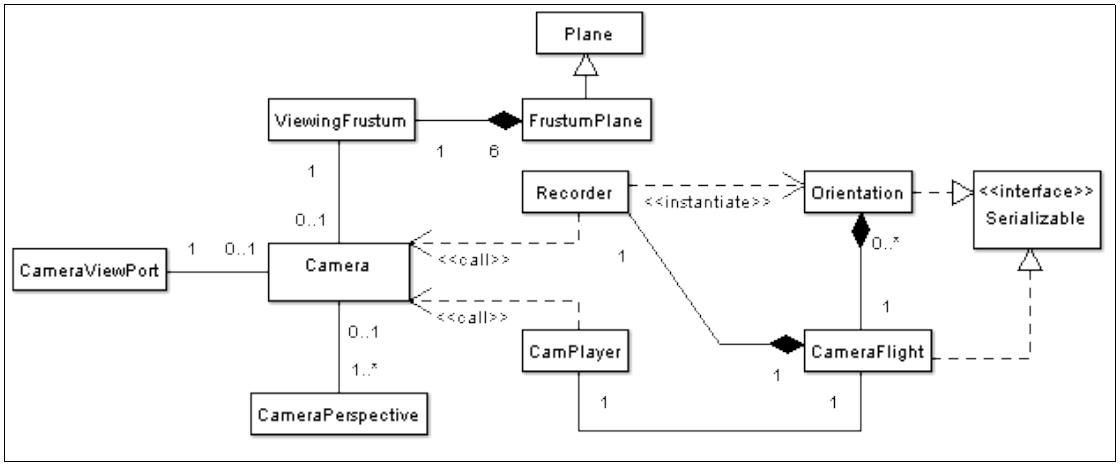


Abbildung 28: UML-Diagramm des Kamera-Systems

4.1.3.4 Frustum-Culling

Beim Frustum-Culling wird untersucht, ob sich das Hüll-Volumen eines 3D-Objektes im Kamera-Frustum befindet. Ist dies nicht der Fall, liegt das Objekt nicht im Blickfeld der Kamera und kann somit beim Zeichnen der Szene vernachlässigt werden. Auf diese Weise ist die Herbeiführung eines signifikanten Performance-Gewinns möglich. Der Effekt ist umso stärker, wenn das aussortierte Modell aus vielen Polygonen besteht. Nicht nur für das Frustum-Culling, sondern auch für etwaige Kollisionserkennungen wurden eine Reihe von Klassen eingeführt, die das Interface `BoundingVolumeIface` implementieren. Diese Klassen repräsentieren verschiedene Formen von Hüll-Volumina (Kugel, Quader, Zylinder). Allen gemein ist die Implementierung der Methode `isWithinFrustum()`, die im zuvor genannten Interface definiert wurde. Sie gibt einen Boolean-Wert zurück, der anzeigt, ob sich der umhüllte Körper, ein `AbstractMesh`-Objekt, im Kamera-Frustum aufhält. Zum Ende des Projekts steht diese Funktionalität für die Klassen `BoundingSphere` und `BoundingBox` korrekt zur Verfügung. Eine vollständige Implementierung der Culling-Funktion für die Klasse `BoundingCylinder` steht jedoch noch aus. Exemplarisch wird in Abbildung 29 der Quell-Code vorgestellt, der testet, ob sich eine Hüll-Kugel im Frustum befindet. Drei Elemente spielen eine Rolle in diesem Test: Der Mittelpunkt der Kugel, ihr Radius und die mathematische Ebenengleichung der Form $Ax + By + Cz + D = 0$, die die Clipping-Planes beschreibt. Die Koeffizienten A, B, C und D liefert die bereits bekannte Klasse `FrustumPlane` als aufbereitete Werte. Die Koordinaten x, y und z entsprechen dem Zentrum der Kugel. Der Algorithmus prüft, ob die Kugel, vollständig spezifiziert durch Mittelpunkt und Radius, hinter einer der sechs Clipping-Planes liegt. Wird er fündig, kann gefolgert werden, dass das assoziierte Mesh-Objekt nicht gerendert werden muss.

```
public boolean isWithinFrustum(ViewingFrustum frustum)
{
      // organize the planes in an array to be able to use
      // a simple loop for processing them
      FrustumPlane[] planes = new FrustumPlane[6];
      planes[0] = frustum.getFrontPlane();
      planes[1] = frustum.getBackPlane();
      planes[2] = frustum.getLeftPlane();
      planes[3] = frustum.getRightPlane();
      planes[4] = frustum.getTopPlane();
      planes[5] = frustum.getBottomPlane();

      // check sphere with each plane
      for(int i=0; i<planes.length; i++)
      {
            if (planes[i].getA() * this.getCenter().getX()
                  + planes[i].getB() * this.getCenter().getY()
                  + planes[i].getC() * this.getCenter().getZ()
                  + planes[i].getD() <= -this.getRadius())
                  return false;
      }

      // return true if none of the planes caused a return of false
      return true;
}
```

Abbildung 29: Implementierung des Frustum-Tests für Bounding-Spheres

4.1.3.5 Werkzeuge für die Kollisionserkennung

Mit der Klasse `CollisionFinder` steht ein Werkzeug zur Verfügung, dass verschiedenste Arten von Kollisionen erkennen kann. Zwei Blöcke kristallisieren sich jedoch heraus, wenn man die Klasse betrachtet. Einerseits ermittelt sie Überschneidungen zwischen gleichartigen Hüll-Volumina (überladene `collide()`-Methoden) und andererseits berechnet sie Schnittpunkte zwischen Mesh-Dreiecken und 3D-Linien sowie Hüll-Volumina und 3D-Linien (überladene `getPointOfIntersection()`-Methoden). In Abbildung 30 werden die Methoden der Klasse in einem UML-Diagramm aufgezeigt. `CollisionFinder` enthält ausschließlich statische Methoden. Dass heißt, es ist nicht notwendig eine Instanz zu erstellen, um sie aufrufen zu können. Die `getPointOfIntersection()`-Methoden erwarten als Teil der Parameterliste entweder ein Objekt der Klasse `Line3D` oder zwei Koordinaten des Typs `int`. Die Instanz von `Line3D` repräsentiert eine Linie im dreidimensionalen Raum. Ihr bevorzugtes Einsatzgebiet ist im Mesh-Picking anzusiedeln. Berechnet man eine Linie, ausgehend von einer 2D-Bildschirmposition (z.B. Maus-Koordinaten), in den Raum hinein, ist es möglich zu prüfen, welches Objekt von der Linie geschnitten wird. Das geschnittene Objekt könnte dann gegebenenfalls als selektiert betrachtet werden. Theoretisch ist es jedoch auch denkbar, mit der Linie den geraden Weg eines Geschosses zu simulieren und festzustellen, welches Mesh von seiner Wirkung betroffen ist.

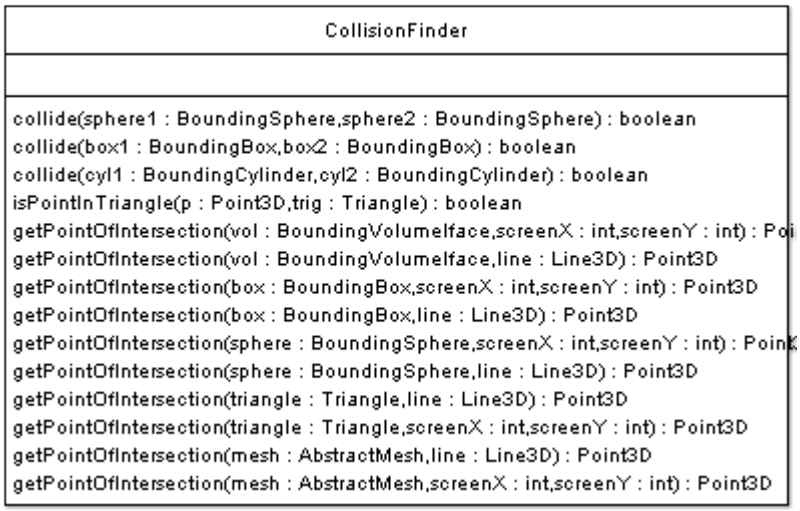


Abbildung 30: Methoden der Klasse CollisionFinder

4.1.3.6 Das Wegfindungs-System auf Basis von A*

David Brackeen stellt in [Brackeen2004] eine generische Implementierung des A*-Wegfindungsalgorithmus vor. Sie fungiert als Grundlage für die Wegfindung in diesem Projekt. Die Generizität wird durch die Spezifikation der abstrakten Klasse `AStarNode` erreicht, die die abstrakten Methoden `getCost()`, `getEstimatedCost()` und `getNeighbors()` enthält. Sie müssen für den speziellen Anwendungsfall, sprich dieses Projekt, konkret ausprogrammiert werden. Der generische Suchalgorithmus auf den Knotenpunkten, befindet sich in der Klasse `AStarSearch`. Um diese beiden Klassen herum, habe ich weitere Klassen erschaffen, die den Algorithmus auf ein quadratisches Bodengitter, dem `SearchGrid`, anwenden. Das Bodengitter besteht aus einer Anzahl gekachelter `GridNode`-Instanzen. `GridNode` wurde von `AStarNode` abgeleitet und stellt demnach dessen konkrete Implementierung dar. Jede Gitterkachel vertritt einen potenziellen Wegpunkt, den es algorithmisch zu untersuchen gilt. Als Erweiterung der `AStarNode`-Klasse, hat `GridNode` deren abstrakte Methoden zu implementieren. Im Sinne eines möglichst kurzen Ergebnispfades, gilt der abstrakten `getEstimatedCost()`-Methode besondere Aufmerksamkeit. Ihre Implementierung liefert den heuristischen Schätzwert, dessen Präzision direkt mit der Qualität des Suchvorgangs in Verbindung steht. Es muss möglichst genau geschätzt werden, wie hoch die Kosten vom Start-Knoten zum Ziel-Knoten voraussichtlich sein werden. Auf die Gitterstruktur bezogen, verwende ich als heuristischen Wert, die ungefähre Anzahl der Kacheln, die sich in der Luftlinie zwischen Start und Ziel befinden, multipliziert mit den Kosten für eine passierbare Kachel. Standardmäßig lege ich fest, dass für explizit begehbare Grid-Nodes der Kostenfaktor 1 zu veranschlagen ist, während Kacheln, die es zu meiden gilt, um ein Tausendfaches teurer sind. Auf diese Weise wird A* hartnäckig dazu gedrängt, die vorgesehenen Wegpunkte einzuschlagen.

Für die Engine nutzbar gemacht wird das Wegfindungs-System letztendlich durch die Klasse `PathFinder`. Sie enthält die statische Methode `getPath()`, die einen Startpunkt und einen Zielpunkt sowie einen Verweis auf die zu durchsuchende `SearchGrid`-Instanz als Übergabeparameter erwartet. Als Rückgabewert erhält man ein Objekt der Klasse `Path`, das die Wegpunkte des algorithmisch ermittelten Pfades kapselt. Abbildung 31 stellt den Zusammenhang zwischen den beschriebenen Komponenten in einem UML-Diagramm dar.

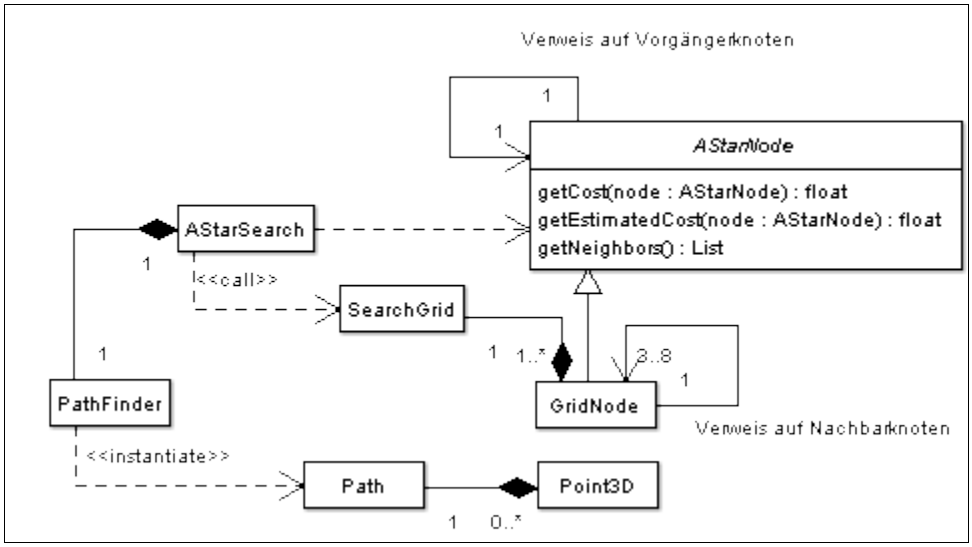


*Abbildung 31: Architektur des Wegfindungs-Systems auf Basis von A**

4.1.3.7 Komfortable Kapselung der OpenGL-Beleuchtungsfunktionen

In Kapitel 4.1.2.8, dem Entwurf des Lichtsystems, wurde festgehalten, dass eine Art Vererbungshierarchie der Attribut-Werte zwischen den drei vorgestellten Lichttypen besteht. Sie eignet sich besonders für die Nachahmung in einer Java-Implementierung. Ausgangspunkt der Kapselung des OpenGL-Lichts ist die abstrakte Oberklasse `AbstractLight`. Sie enthält vor allem Methoden zur Manipulation der Lichtfarbe. Davon direkt abgeleitet werden die beiden Klassen `DirectionalLight` und `PositionalLight`. Sie verfügen jeweils über spezialisierte Operationen, die der Natur des Lichtes, dass sie repräsentieren, geschuldet sind. Somit enthält `DirectionalLight` zum Beispiel die Option, die Lichtrichtung festzulegen, während `PositionalLight` es vorsieht, die Lichtposition spezifizieren zu können. Als Spezialfall eines positionellen Lichts tritt die Klasse `SpotLight` auf. Demzufolge ist sie als Erweiterung von `PositionalLight` angelegt worden und erlaubt die Konfiguration der Scheinwerfer-Einstellungen.

Hat man eine der drei potenziellen Lichtklassen instantiiert und nach Wunsch parametrisiert, kann die Objekt-Beleuchtung durch das Licht über den Aufruf der gemeinsamen `bind()`-Methode aktiviert und mit `release()` wieder deaktiviert werden. Anhand einer parallel existierenden Hierarchie von Interfaces, werden die OpenGL-Lichtklassen dazu verpflichtet, diese und weitere Methoden zu implementieren. Abbildung 32 zeigt ein Klassendiagramm, welches die Beziehungen der Licht-Klassen und -Interfaces zueinander beschreibt.

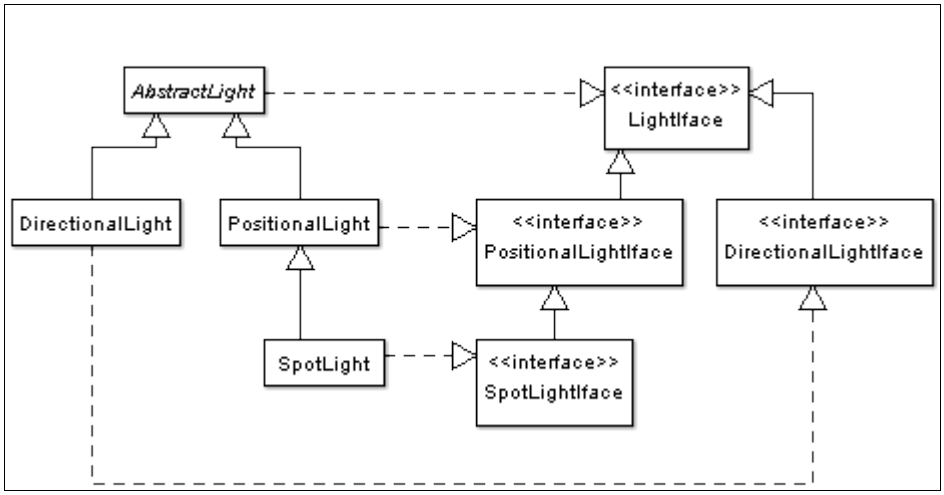


Abbildung 32: Architektur des Beleuchtungs-Systems

Die Herangehensweise, die drei Lichttypen in Interfaces abzubilden, hat einen positiven Effekt auf die Erweiterungsfähigkeit der Architektur des Lichtsystems. Die Interfaces stellen Ansatzpunkte dar, die es ermöglichen, neben der klassischen OpenGL-Beleuchtung, andere Techniken zur Umsetzung von Lichtfunktionalitäten zu implementieren. In Zukunft wäre beispielsweise die Integration moderner Shader-Programme denkbar, die die optische Brillanz des Lichteinfalls dramatisch verbessern.

4.1.3.8 Integration von Grafik- und Ton-Effekten

In die Engine wurden einige Klassen integriert, die für die Darstellung von Effekten zuständig sind, die als für die Spiel-Atmosphäre förderlich betrachtet werden können. Manche von ihnen sind in einzelne Klassen gekapselt, die für sich allein stehen, wie zum Beispiel die Nebelfunktionalität der Klasse `Fog` oder der Linseneffekt der Klasse `LensFlare`. Andere wiederum verfügen über etwas komplexere Strukturen. Zu Letzteren kann eventuell die Implementierung der beiden Environment-Mapping-Techniken Sphere- und Cube-Mapping gezählt werden, auf jeden Fall aber die Umsetzung des Partikel-Systems. Beide Komponenten werden nachfolgend kurz beschrieben, bis dann im Anschluss auf die Tonwiedergabe-Klassen eingegangen wird.

Integration des Environment-Mapping

Environment-Mapping ist der Oberbegriff für Techniken, die versuchen, die Umgebung einer dreidimensionalen Szene, spiegelnd auf enthaltene 3D-Objekte abzubilden. In dieser Engine wurden zwei Ansätze des Environment-Mappings verwirklicht: Das Sphere-Mapping und das Cube-Mapping. Beide Verfahren werden hardwareseitig inhärent durch OpenGL unterstützt.

Das Interface `EnvironmentMappingIface` stellt die gekapselten Implementierungen der CubeMap und SphereMap-Klasse auf eine gemeinsame funktionale Basis, so dass sie sich innerhalb einer Anwendung leicht austauschen lassen. Das Environment-Mapping ist eine Texturierungstechnik auf Basis automatisch berechneter Textur-Koordinaten. Dementsprechend repräsentieren Instanzen des `EnvironmentMappingIface`-Interfaces eigentlich komplexere Texturen. In der Engine ist daher vorgesehen, solche Instanzen über

einfache Zuweisungen (`set()`-Methode) an ein Objekt der Klasse `AbstractMesh` binden zu können. Abbildung 33 stellt die Beziehung zwischen den Klassen kurz dar.

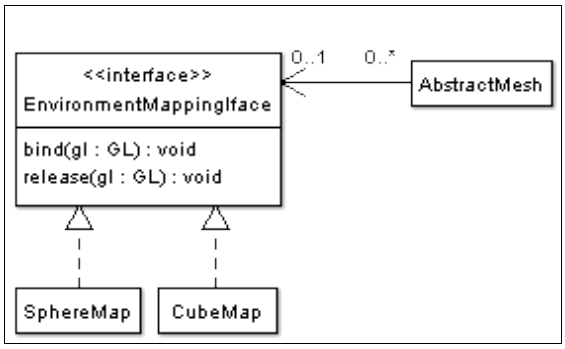


Abbildung 33: Environment-Mapping in Korrelation zu Mesh-Objekten

Implementierung des Partikel-Systems

Wie im Entwurf bereits angedeutet (Kapitel 4.1.2.9), fußt das Partikel-System auf drei wesentlichen Komponenten. Die erste Komponente beschreibt den Partikel-Effekt. In der Engine wird sie durch die abstrakte Klasse `AbstractParticleEffect` widergespiegelt. Neben anderen Methoden, enthält sie die abstrakte Methode `generateParticles()`. Neue Partikel-Effekte werden erstellt, indem von `AbstractParticleEffect` abgeleitet und die `generateParticles()`-Methode konkret implementiert wird. Vorgesehen ist, dass die konkrete Implementierung Code enthält, der ein Partikel instantiiert und anschließend seine Eigenschaften festlegt. Insbesondere die Bewegungsrichtung und Farbe eines Partikels gilt es zu spezifizieren. Von Hause aus bringt die Engine eine prototypische Implementierung verschiedener Effekte mit sich, die zwar ohne weiteres Zutun eingesetzt werden können, jedoch eher simpel ausgeprägt sind.

Die zweite Komponente wird durch die Partikel gebildet. Sie findet in der abstrakten Klasse `AbstractParticle` ihre Entsprechung. Drei Unterklassen leiten sich von `AbstractParticle` ab. Sie stehen für drei verschiedene Verfahren, ein Partikel intern zu repräsentieren. Die Klasse `DDParticle` zeichnet sie als texturierte Quadrate direkt auf den Bildschirm. Es wird also kein Gebrauch von präkompilierten Display-Listen gemacht. Partikel der Klasse `DLParticle` hingegen, bestehen zwar ebenfalls aus texturierten Quadraten, machen sich aber Display-Listen zu Nutze. Einen anderen Weg schlägt die Klasse `PSParticle` ein. Zum Zeichnen ihrer Partikel-Instanzen werden Point-Sprites verwendet. Point-Sprites basieren auf einer OpenGL-Extension, die seit der Version 1.4 verfügbar ist. An die Stelle von Quadraten treten Punkte, die jedoch auch texturiert werden können. Die Datenmenge reduziert sich, weil nicht mehr vier Vertizes, sondern nur noch ein Punkt gezeichnet werden muss, um ein Partikel darzustellen.

Die letzte Komponente ist der Partikel-Container. Die Java-Klasse `ParticleContainer` übernimmt deren Funktion. Ein Objekt dieser Klasse ist fest mit `AbstractParticleEffect` verankert. Sobald ein neues Partikel instantiiert wird, kann es dem Partikel-Container zugeführt werden. Dieser verwaltet die Partikel eines Effekts und stellt sie bereit, wenn sie

gerendert werden sollen. Das Klassendiagramm aus Abbildung 34 demonstriert die Architektur des Partikel-Systems mitsamt den prototypischen Effekten.

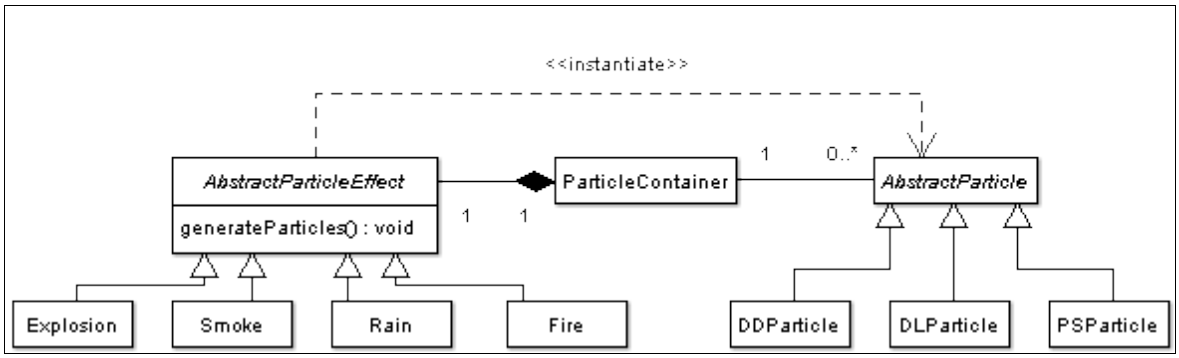


Abbildung 34: Architektur des Partikel-Systems

Implementierung der Audio-Klassen

Die Game-Engine unterstützt die Wiedergabe verschiedener Audioformate. Dazu gehören einerseits unkomprimierte Sound-Sample-Files, wie das WAV-Format und andererseits die Speicherplatz schonenden Formate MIDI und Mp3. Alle drei Typen werden über eigene Klassen in die Engine integriert. Damit sich Erweiterungen am Audio-System nahtlos einfügen lassen, haben alle Audio-Klassen das `SoundFileIface`-Interface zu implementieren. Es definiert Methoden, die allen Umsetzungen gemein sein sollten. Besonders hervorzuheben sind die `play()`- und die `stop()`-Methode, die das jeweilige Musik-Stück starten respektive stoppen sollen.

Die Klasse `SampledSound` dient vor allem zum Abspielen von WAV-Dateien. Sie verwendet intern ein Objekt der Java-Standardklasse `AudioClip`, die ursprünglich für Applets gedacht war, um Audiodaten wiederzugeben. Die Klasse zum Abspielen von MIDI-Files (`MidiSound`) habe ich [Brackeen2004] entnommen. Es flossen jedoch einige Veränderungen meinerseits ein, die Vereinfachungen und gleichförmige Integration zum Ziel hatten. Letztlich enthält das Sound-Paket noch die Klasse `Mp3Sound`. Ihre Instanzen kapseln komprimierte Dateien im Mp3-Format. Zum Einlesen und Wiedergeben des Mp3-Datenstroms greift die Klasse auf die freie JLayer-Bibliothek[30] der Firma JavaZOOM zurück. Abbildung 35 zeigt eine Übersicht der Audio-Klassen mit der Verbindung zu ihrem gemeinsamen Interface.

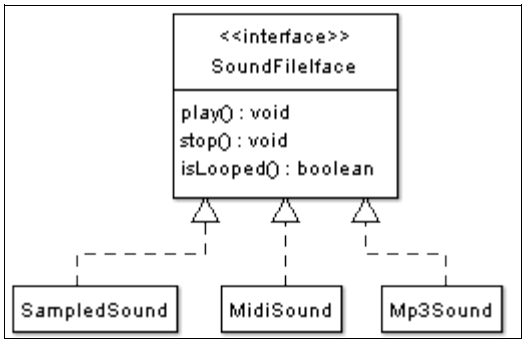


Abbildung 35: Die Audio-Wiedergabeklassen im Überblick

30 JLayer-Library für Mp3-Wiedergabe - http://www.javazoom.net/javalayer/javalayer.html

4.1.3.9 Umsetzung der 2D-Overlay-Funktionalitäten

Nachdem im Entwurf dargelegt wurde, aus welchen Bestandteilen die Utility-Klassen zur orthographischen Projektion von Overlay-Elementen bestehen sollen, folgt nun die Beschreibung der konkreten Umsetzung derselbigen.

Das Overlay-System besteht aus den drei Basis-Klassen `TextRenderer`, `RectangleRenderer` und `ImageRenderer`. Abstrakte Oberklasse und Basis aller weiteren Overlay-Klassen ist `AbstractOverlay`. Eine Besonderheit des orthographischen Projektionsmodus, den die Engine nutzt, ist die Invertierung der Y-Achse des 2D-Koordinatensystems. Ungleich der Standardeinstellung, lege ich den Ursprung der Y-Achse am oberen linken Bildschirmrand fest. Auf diese Weise nähern sich die Overlay-Klassen den Konventionen des Java-2D-API an.

`TextRenderer` bietet überladene `drawString()`-Methoden an, die Zeichenketten an spezifizierte Bildschirmkoordinaten rendern. Die Implementierung des Text-Renderers basiert intern auf der GLUT-Funktion `glutBitmapString()`, so dass die Schriftarten *Helvetica* in den Pixelgrößen 10, 12 und 18 sowie *Times New Roman* in den Größen 10 und 24 zur Verfügung stehen. Sie sind in der Java-Enumeration `Font` gekapselt. Instanzen der Klasse `RectangleRenderer` zeichnen farbige Rechteckflächen über das 3D-Bild, die auf Wunsch Transparenz annehmen können. Die Aufgabe der `ImageRenderer`-Objekte ist die grafische 2D-Projektion von Bilddateien wie JPG oder PNG. Im Grunde handelt es sich um die Visualisierung eines texturierten Rechtecks. Die Textur wird in ein Objekt der Klasse `OverlayImage` gekapselt, das zusätzlich eine designierte Höhe und Breite für den Zeichenvorgang der Bildgrafik aufnehmen kann. Ähnlich dem Rectangle-Renderer kann der Image-Renderer die Overlay-Grafiken ebenfalls transparent zeichnen.

Aus der Kombination der beschriebenen Komponenten, sind die Klassen `TextAreaRenderer` und `ListMenuRenderer` hervorgegangen. Die Aufgabe der `TextAreaRenderer`-Instanzen ist das Zeichnen von Texten innerhalb einer rechteckigen Fläche. Im Gegensatz zum einfachen Text-Renderer kann der Text mehrere Zeilen lang sein. Er wird innerhalb des Textfeldes korrekt umgebrochen. Des weiteren können Textfelder mit einem Hintergrundbild ausgestattet werden, das sich der Größe des Feldes anpasst. Dies gilt auch für Instanzen der Klasse `ListMenuRenderer`. Hauptanliegen ist es jedoch die Grundfunktionen eines einfachen Listen-Menüs anzubieten. Zu einem Listen-Menü lassen sich dynamisch Einträge hinzufügen und entfernen. Durch die Angabe der aktuellen Mauskoordinaten lässt sich zudem ermitteln, auf welchem Eintrag die Maus gerade verweilt. Mit etwas Geschick ist es somit denkbar, ein einfaches Spiel-Menü zu realisieren. Erneut soll ein UML-Klassendiagramm die Beziehungen der einzelnen Klassen zueinander klarstellen (Abbildung 36).

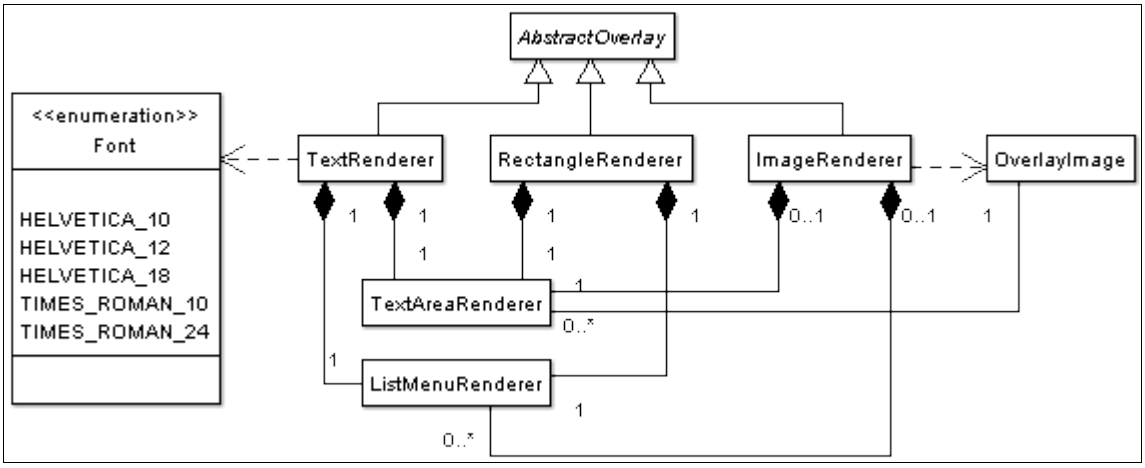


Abbildung 36: Zusammenhang der 2D-Overlay-Klassen

4.1.3.10 Registrierung von Maus- und Tastatureingaben

Die Komponente zur Verarbeitung von Input-Events basiert im Grunde auf einer recht flexiblen Lösung aus [Brackeen2004]. Zwei Klassen verbergen den Umgang mit dem Java-AWT-Event-Modell vor dem Programmierer. Die Klasse `InputAction` kapselt Informationen über eine potenzielle Eingabe-Aktion. Ihre Instanzen erhalten die Einstellung, ob ein Tastendruck diskret oder kontinuierlich registriert werden soll. Des weiteren kann eine textuelle Beschreibung assoziiert werden. Über die boolesche Methode `isPressed()` wird abgefragt, ob die Eingabe-Aktion ausgeführt wurde. Alle Eingabe-Aktionen werden von der Klasse `InputManager` verwaltet. Während eine `InputAction`-Instanz dem `InputManager` zugewiesen wird, legt man die Maus- beziehungsweise Keyboard-Taste fest, die mit der Eingabe-Aktion verknüpft werden soll. `InputManager` implementiert die essenziellen AWT-Input-Event-Interfaces `KeyListener`, `MouseListener`, `MouseMotionListener` und `MouseWheelListener`. Tritt ein Input-Event auf, wird die entsprechende Interface-Methode aufgerufen und das assoziierte `InputAction`-Objekt in den Zustand *„ausgelöst“* versetzt. `InputManager` kann außerdem verwendet werden, um die aktuellen Mauskoordinaten abzufragen und einen Mouse-Look-Modus zu aktivieren. Abbildung 37 zeigt die Input-Klassen im Überblick.

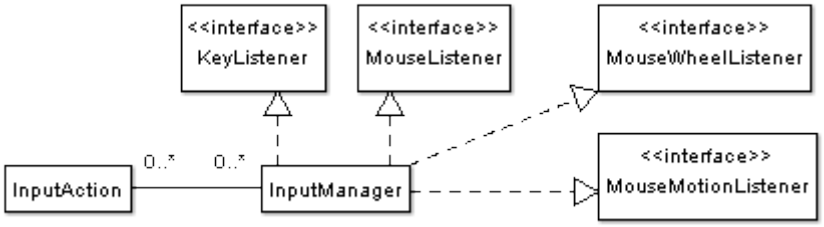


Abbildung 37: Darstellung des Eingabe-Verarbeitungssystems

4.2 Teil 2 – Das spezifische Adventure-Framework

Nachdem in Teil 1 die Umsetzung einer eigenständigen Game-Engine beschrieben wurde, folgt nun in Teil 2 die Beschreibung der Analyse, des Entwurfs und der Implementierung des adventure-spezifischen Frameworks, das auf der Engine aufsetzen wird.

4.2.1 Analyse und Definition der Anforderungen

Für dieses Projekt habe ich es mir zum Ziel gesetzt, ein Framework zu erstellen, mit dem man in die Lage versetzt wird, Point-and-Click-Adventures zu entwickeln, deren Spielmechanik sich an professionellen Produkten orientiert. Um diesem Anspruch gerecht werden zu können, muss das Framework eine Reihe von Anforderungen erfüllen, die im Folgenden vorgestellt werden.

4.2.1.1 3D-Game-Engine als Grundlage für das Framework

Es klingt nach einer trivialen Anforderung, dennoch ist es wichtig sie zu erwähnen: Das zu entwickelnde Adventure-Framework soll auf der 3D-Game-Engine aufsetzen, die ebenfalls Teil dieses Projekts ist. Dementsprechend unterliegt das Framework den technischen Vorgaben der Engine. Beispielsweise ist die Auswahl möglicher 3D-Modelle auf die Formate begrenzt, die von der Engine unterstützt werden. Ebenso verhält es sich zum Beispiel mit der Wegfindung. Adventures werden das gridbasierte A*-Verfahren (Kapitel 4.1.3.6) benutzen müssen, um Laufwege einer Spielfigur berechnen zu lassen.

4.2.1.2 Kulissen und Hotspots

Die Spielwelten von Point-and-Click-Adventures werden für gewöhnlich in einzelne Räume unterteilt. In diesen Räumen befinden sich funktionslose Requisiten, die als Kulisse bezeichnet werden können und gegebenenfalls interaktive Zonen, so genannte *Hotspots* oder *Points-of-Interest*. Hotspots treten in verschiedenen Gestalten auf. Sie können entweder direkt mit Nicht-Spieler-Charakteren assoziiert sein oder in Form von (auflesbaren) Gegenständen, den so genannten *Items,* auftreten. Es sind auch virtuelle Hotspots denkbar, die interessante Stellen in der Spielwelt markieren, welche sich jedoch weder den Charakteren noch den Items zuordnen lassen. Solche Hotspots können zum Beispiel für Bereiche definiert werden, die in erster Linie für die Kombination mit einem Gegenstand aus dem Inventar des Spieler vorgesehen sind. Die folgenden Unterkapitel erläutern die Anforderungen an die passiven und interaktiven Komponenten eines Raums.

Die Kulisse

Kulissen haben keine spezielle Funktion in Adventure-Spielen. Sie dienen lediglich zur Gestaltung der Umgebung. Auf diese Weise tragen sie aber erheblich zur Wahrnehmung der Spielwelt als einer der Realität nachempfundenen Umgebung bei. Da die Räume eines 3D-Adventures naturgegeben aus modellierten 3D-Objekten bestehen, muss das Framework eingelesenen 3D-Modellen konkret die Rolle eines passiven Requisits zuordnen können.

Items

Denkt man an Point-and-Click-Adventures, dürfte einem auch in den Sinn kommen, dass das Aufsammeln und Benutzen von Items charakteristische Merkmale des Genres sind. Selbstverständlich dürfen sie in einem solchen Framework nicht fehlen. Somit muss die Möglichkeit bestehen, 3D-Modelle in der Rolle von Items auftreten zu lassen.

Charaktere

Charaktere, insbesondere Nicht-Spieler-Charaktere, sind in einem modernen Adventure unentbehrlich. Nicht nur, dass sie die Szenerie beleben, sie können fundamental in die Handlung integriert werden. Oftmals erfährt man von Nicht-Spieler-Charakteren wichtige Informationen für das Voranschreiten im Spielverlauf. Nicht selten werden sie sogar zum zentralen Element eines Rätsels. Damit Charaktere im Spiel dieser Aufgabe gerecht werden können, müssen sie Interaktionsmöglichkeiten bieten. In der Regel beziehen sich diese auf das Führen von Unterhaltungen und auf die Anwendung von Items auf den Charakter. Ähnlich wie bei den Items und Requisiten, gilt es, einem 3D-Modell im Framework explizit die Rolle eines Charakters zuweisen zu können.

Virtuelle Hotspots

Virtuelle Hotspots sind interaktive Bereiche in der Spielwelt, die nicht direkt mit einem Mesh verknüpft sind. Es sind einfach nur Markierungen, die eine interaktive Stelle kennzeichnen. Häufig unterscheidet man zwischen Hotspots, die mit Items kombiniert werden müssen und solchen, auf denen man Aktionen durchführen kann, ohne Inventar-Gegenstände dafür zu bemühen. Das Adventure-Framework muss die Definition beider Hotspot-Typen erlauben.

4.2.1.3 Kontaktsensitive Bodenbereiche: Trigger

In Video-Spielen trifft man häufig auf Ereignisse, die ausgelöst werden, sobald der Spieler sich bestimmten Stellen eines Levels nähert. Beispielsweise öffnen sich Türen gelegentlich automatisch, wenn eine Spielfigur in die Nähe einer solchen tritt. Dieses Verhalten kann durch so genannte *Trigger* gesteuert werden. Trigger sind kontaktsensitive Bereiche, die auf die Kollision mit Spielobjekten reagieren können. Dringt ein Spielobjekt in den sensitiven Bereich eines Triggers ein, wird ein definiertes Ereignis ausgelöst; zum Beispiel das Öffnen eines nahe gelegenen Tors. Trigger können folglich den Eindruck einer dynamischen Spielwelt verstärken. Sie lassen sich aber auch ganz konkret in das Rätseldesign einbinden, da assoziierte Ereignisse beliebig sein können. Weil Trigger ein Spiel um interessante Aspekte ergänzen können, sollen sie unbedingt Einzug in das Adventure-Framework halten.

4.2.1.4 3D-Kamera zur Präsentation der Spielszene

Der augenscheinlichste Unterschied in der Entwicklung von 3D-Adventures zu klassischen 2D-Adventures ist die freie Wahl der Kameraperspektive zur Laufzeit des Spiels. Während man im 2D-Bereich für verschiedene Ansichten der gleichen Szene im Voraus ein vollständig neues Hintergrundbild erstellen muss, kann im dreidimensionalen Raum die Kamera fließend bewegt werden, um das Geschehen aus einem neuen Blickwinkel zu präsentieren.

Point-and-Click-Adventures zeigen die Spielfigur klassischerweise in der dritten Person. Daraus ergeben sich zwei Ansätze zur Integration des Kamera-Systems. Zum einem kann

eine Kamera verwendet werden, die die Spielfigur permanent im Fokus behält. Durch dynamische Veränderung ihrer Position, passt sie sich der Laufrichtung des Charakters an. Man spricht von einer Verfolgerkamera. Die andere Vorgehensweise zeigt die Szene aus unbeweglichen Blickwinkeln. Je nach Position der Spielfigur wird eine andere Ansicht aktiviert. In diesem Fall spricht man von einer statischen Kamera mit wechselnden Perspektiven. Im weiteren Verlauf der Arbeit gilt es abzuwägen, welche Methode für dieses Adventure-Framework attraktiver erscheint.

4.2.1.5 Dateiformat zur Spezifikation von Räumen

Die Spezifikation des Aufbaus einzelner Räume mittels hartem Programmcode ist zwar möglich, aber als ausgesprochen unflexibel zu bezeichnen. Dem Prinzip des Data-Driven-Programming entsprechend, sollten die Inhalte eines Spiels vom Programmcode entkoppelt werden. Auf diese Weise wird es für einen Level-Designer deutlich einfacher, Spielinhalte zu variieren, ohne auf die Mithilfe eines Programmierers angewiesen zu sein. Der Einsatz von Skriptdateien zur Deklaration kausalen Verhaltens außerhalb des Programmcodes deutet in die gleiche Richtung [DeLoura2002].

Damit die Konstruktionsvorschrift eines Raums persistent in eine externe Datei gespeichert werden kann, muss zuvor ein Dateiformat festgelegt werden, das verbindlich regelt, welche Informationen an welcher Stelle abzulegen sind. Auf dieser Grundlage ist es möglich, die Datei strukturiert einzulesen und einen Raum anhand der enthaltenen Angaben zu rekonstruieren. Verbindliche Formatdefinitionen begünstigen zudem die Entwicklung von Software-Tools wie Level-Editoren zum Zusammenbau der Spielwelten. Eine Spezifikation, die diese Anforderung erfüllt, muss für das Framework entworfen werden.

4.2.1.6 Sequenziell ausführbare Spiel-Aktionen und Ereignisse

In einem Point-and-Click-Adventure verfügen die agierenden Charaktere über ein gewisses Repertoire an Handlungsmöglichkeiten. Besonders gängig sind das Aufnehmen von Items, das Untersuchen von Hotspots oder das Wandern zu bestimmten Positionen im Raum. Denkt man etwas darüber nach, gelangt man mitunter zu der Erkenntnis, dass es wichtig ist, diese Aktionen diskret hintereinander ausführen zu können. Manche Handlungen dürfen demnach erst dann vollzogen werden, wenn andere abgeschlossen sind.

Betrachten wir einmal folgendes Szenario: Die Spielfigur befindet sich an Punkt A. Ein interessantes Item befindet sich an Punkt B. Der Spieler möchte das Item aufnehmen. Wenn er den Befehl zum Einstecken des Gegenstandes gegeben hat, sollte sich die Spielfigur automatisch zu Punkt B, also dem Standort des Items, begeben. Sobald der Zielpunkt erreicht ist, wird das Item tatsächlich eingesammelt. Man sieht also, dass die Aktion *„Item aufnehmen“* erst nach dem Abschluss der implizit befohlenen Aktion *„Zur Item-Position begeben“* erfolgen darf. Anderweitiges Verhalten hinterließe einen verwirrenden Eindruck beim Spieler, weil es nicht dem logisch kausalen Verlauf entspräche. Aber nicht nur Steuerbefehle müssen sequenziell ausführbar sein. Es sind durchaus Aktionen und Ereignisse denkbar, die ebenfalls nicht unmittelbar stattfinden dürfen. So müssen beispielsweise die Sätze, die Charaktere in Dialogen von sich geben, in vorgegebener Reihenfolge geäußert werden können. Ein weiteres Beispiel ist die Wiedergabe mehrerer aufeinander folgender Charakteranimationen.

Da ich gerade die Nützlichkeit und Notwendigkeit sequenzieller Befehlsausführungen umrissen habe, wird ersichtlich, dass es unbedingt erforderlich ist, eine solche Befehlsverarbeitung in das Framework aufzunehmen.

4.2.1.7 Inventarsystem

Ein vom Spieler gesteuerter Adventure-Charakter liest während des Spielverlaufs in der Regel eine Fülle von Items auf. An vorgesehenen Stellen können sie dann hervorgeholt werden, um einen Beitrag zur Bewältigung spezieller Rätselaufgaben zu leisten. Die Gesamtheit aller eingesteckten Gegenstände wird als Inventar bezeichnet. Damit ein komfortabler Zugriff auf das Inventar erfolgen kann, wird die Entwicklung eines Verwaltungssystems für seinen Inhalt nahe gelegt. Durch eine übersichtliche Darstellung soll eine schnelle Selektion des gewünschten Items gewährleistet werden.

4.2.1.8 Dialogsystem

Ein Adventure lebt von der Interaktivität zwischen Spielfigur und Umwelt. In vielen Adventures ist es deshalb wichtig, mit Nicht-Spieler-Charakteren zu kommunizieren. Die Gespräche können unter Umständen nebensächliche Themen behandeln, oftmals versorgen sie den Spieler jedoch mit interessanten Auskünften, die Relevanz für den weiteren Spielverlauf besitzen.

Damit die Gesprächsführung nicht zu statisch gerät und die Handlungsfreiheit des Spielers einschränkt, soll sich das Framework eines klassischen Dialogsystems bedienen. Beginnt der Spieler ein Gespräch, soll ihm eine Auflistung von Sätzen zur Verfügung stehen, aus der beliebig selektiert werden kann. Jede der Wahlmöglichkeiten soll das Gespräch durch einen vorgegebenen Dialogstrang führen. Wichtig ist, dass die Auflistung nachträglich um weitere Sätze ergänzt werden kann, um den Fall zu simulieren, dass bestimmte Informationen zu neuen Dialogoptionen führen.

4.2.1.9 Intuitives Kommando-Interface

Die Art und Weise, wie der Spieler Steuerbefehle zur Interaktion mit der Spielwelt absetzen kann, ist entscheidend für die Akzeptanz eines Adventures. Tendiert die Steuerung dazu, eher komplex und sperrig auszufallen, nimmt die Motivation des Spielers ab, sich längere Zeit mit dem Spiel zu beschäftigen. Der Eindruck eines unhandlichen Kommando-Interfaces kann den Spielspaß auf Dauer zunichte machen. Adventures, die mit dem zu entwickelnden Framework erstellt werden, sollen über intuitive Steuerungsmechanismen verfügen, die den Aufwand zur Befehlsplatzierung möglichst niedrig halten. Führt ein einzelner Mausklick in der Regel zur gewünschten Aktion, so kann von einer effizienten Lösung gesprochen werden.

4.2.1.10 Game-Scripting

Game-Scripting ist ein weit verbreitetes Instrument, um die Spiellogik in externe Dateien auszulagern. Dadurch können Änderungen vorgenommen werden, ohne den Quellcode des Frameworks berühren zu müssen. Game-Scripting ist deshalb ein fundamentaler Baustein im Gefüge eines generischen Ansatzes. Aus diesem Grund soll das Adventure-System die Schnittstelle zu einer Skriptsprache bereitstellen. Darin verfasste Skripte sollen benutzt

werden, um die Auswirkungen der Aktionen des Spielers festzulegen und vordefinierte Ereignisse zu deklarieren.

4.2.2 Entwurf

Nachdem im Kapitel zuvor die wichtigsten Anforderungen umrissen wurden, beschreibt dieses Oberkapitel den Entwurf des Adventure-Frameworks. Er wird die Basis für eine konkrete Umsetzung in der Implementierungsphase sein. Die nachfolgenden Unterkapitel heben jeweils bedeutsame Aspekte des zu entwerfenden Systems hervor.

4.2.2.1 Strukturierung der Spielwelt

Es ist sinnvoll, die Spielwelt in separate Abschnitte zu gliedern. Auf diese Weise muss nur der Spielinhalt des jeweiligen Bereichs im Speicher gehalten werden. In Adventures findet sich oft eine Aufteilung des gesamten Spiels in separate Kapitel. Ist die Handlung eines Kapitels abgeschlossen, gelangt man in der Regel nicht mehr an die Orte vorheriger Kapitel zurück.

Das Framework dieses Projekts wird ebenfalls die Unterteilung in Kapitel unterstützen. Jedem Kapitel wird dabei eine Menge von Räumen zuzuordnen sein, deren Inhalte beim Starten des Kapitels zu laden sind. Das hat zur Folge, dass zwar etwas Wartezeit vor dem Beginn eines jeden Kapitels entsteht, jedoch im Verlauf des Spielabschnitts keine Pausen durch das Nachladen weiterer Ressourcen verursacht werden.

Ein Kapitel wird aus einem oder mehreren Räumen bestehen. Räume können als Container-Struktur betrachtet werden, die passive und interaktive Szenekomponenten enthalten können. Die passiven Komponenten umfassen insbesondere die statischen Kulissenaufbauten. Die interaktiven Teile sind Charaktere, Items und sonstige Hotspots. Ein Raum muss in der Lage sein, seine Bestandteile zu verwalten. Deshalb werden eine Reihe von Methoden benötigt, die neue Komponenten hinzufügen und überflüssige entfernen können.

4.2.2.2 Die interaktiven Szenekomponenten

Den interaktiven Szenekomponenten gebührt besondere Aufmerksamkeit, weil sie die Essenz eines jeden Point-and-Click-Adventures ausmachen. Gäbe es sie nicht, wäre das Spiel seiner Mechanik beraubt. Adventures leben davon, den Spieler die Umgebung erkunden zu lassen, Gespräche führen zu können und Rätsel durch den Einsatz von Inventargegenständen zu lösen.

Reine Point-and-Click-Adventures werden ausschließlich über die Maus gesteuert. Der Spieler bewegt den Cursor über die Szene, um interaktive Bereiche zu entdecken. Um dem Spieler einen Fund zu signalisieren, wird das Framework den Namen der Szenekomponente einblenden, über der sich die Maus gerade befindet.

Interaktive Bereiche sind eng mit dem Game-Scripting verbunden. Das Anklicken eines solchen Bereichs wird im Regelfall zur Ausführung verknüpften Skript-Codes führen.

Die genannten Eigenschaften sind relevant für alle Typen interaktiver Szenekomponenten. In anderen Teilbereichen sind jedoch spezialisierte Fähigkeiten notwendig, die in nachfolgenden Unterkapiteln gesondert behandelt werden.

Items

Die wesentliche Funktionalität eines Items ist es zunächst, sich auflesen lassen zu können. Im Bedarfsfall wird es aber möglich sein, ein Item (zeitweilig) als inaktiv zu deklarieren, so dass es zwar angeschaut, aber nicht genommen werden kann. Wenn die Spielfigur ein Item einsteckt, muss es zum Inventar hinzugefügt werden. Für diesen Zweck ist es wichtig, eine Grafik mit dem Item zu assoziieren, die das Item im Inventar symbolisch repräsentiert.

Charaktere

Charaktere im Framework zeichnen sich vor allem als Dialogpartner aus. Sie werden deshalb eine Sprachfunktionalität aufweisen. Spielbare Charaktere erhalten zusätzlich ein Inventar, in das sie Items aufnehmen können. Die Anwendung von Gegenständen auf Nicht-Spieler-Charaktere wird eine Option darstellen.

Virtuelle Hotspots

Virtuelle Hotspots werden im Adventure-System als unsichtbare Kugeln und Quader auftreten. Mit ihnen lassen sich beliebige Stellen in Raum als interaktive Zone markieren. Ihre Hauptaufgabe besteht darin, zu erkennen, ob sich die Maus des Spielers im Bereich des virtuellen Hotspots befindet. Dazu muss der potenzielle Schnittpunkt zwischen einer gedachten Linie, die von der Mausposition in die Szene zeigt, und der Hotspot-Geometrie ermittelt werden.

Es gilt zu unterscheiden zwischen Hotspots, die mit Items aus dem Inventar kombiniert werden müssen und solchen, die auch ohne Anwendung von Items Skript-Codes ausführen.

Trigger

Die Trigger im Adventure-Framework passen nicht ganz in die bisherige Aufzählung der interaktiven Szenekomponenten, weil sie nicht unmittelbar durch den Mausklick des Spielers ausgelöst werden. Stattdessen reagieren sie auf Charaktere, die in ihren sensitiven Bereich eintreten. Trigger werden im Framework durch unsichtbare Kreise und Rechtecke auf der Bodenebene repräsentiert werden. Mit diesen Formen lässt sich leicht berechnen, ob sich ein Charakter innerhalb des Trigger-Bereichs befindet. Ist das der Fall, wird das System automatisch einen assoziierten Skript-Code ausführen.

4.2.2.3 Entscheidung für ein Kamera-Prinzip

Wie im Anforderungskapitel zu lesen war (Kapitel 4.1.2.4), kommen für 3D-Point-and-Click-Adventures, die die Spielfigur in der dritten Person zeigen, in erster Linie zwei Kamera-Modelle in Frage. Einerseits die bewegliche Verfolgerkamera und andererseits die starre Kamera mit wechselnden Perspektiven. Ein potenziell drittes Prinzip ist die um 360° frei rotierbare Ego-Perspektive. Sie ist im Adventure-Genre jedoch eher unbeliebt und soll im Rahmen dieser Arbeit nicht berücksichtigt werden.

Die Verfolger-Kamera behält den Spieler-Charakter ständig im Mittelpunkt. Bewegt er sich zum Beispiel ein paar Schritte zur Seite, so ahmt die Kamera die Seitwärtsbewegung unverzüglich nach, damit die Spielfigur weiter in Zentrum des Blickfelds steht. Die Kamera wandert dem Spieler hinterher. Dieser genießt eine große Bewegungsfreiheit. Daher entfaltet

dieses Prinzip seinen Nutzen primär in weitläufigen Arealen. Der Spieler kann die Umgebung auf einfache Weise Stück für Stück erkunden. Ist das Territorium genügend groß, wird er dabei nicht so rasch an Level-Grenzen stoßen.

Gleichwohl ist zu beachten, dass Adventures oftmals überwiegend in Gebäuden beziehungsweise überschaubaren Anlagen inszeniert werden, die nur sehr begrenzte räumliche Ausdehnungen annehmen. Vordefinierte Kamera-Perspektiven sind hier womöglich besser geeignet, bringen sie doch einige Vorteile mit sich. Beispielsweise kann die Wahl fester Blickwinkel direkt in das Rätseldesign einfließen. Manche Hotspots könnten so platziert werden, dass sie nur in bestimmten Perspektiven sichtbar sind. Außerdem kann der Aufwand zur Erstellung der Levelarchitektur gegebenenfalls verringert werden, wenn die Szene nur aus festen Blickwinkeln präsentiert wird. Spielfiguren können in aller Regel nur Positionen einnehmen, die im Sichtfeld der Kamera liegen. Schließlich ist der zielgebende Mausklick ebenfalls auf dieses Gebiet beschränkt. Durch die geschickte Auswahl der Kamera-Ansichten ist man in der Lage, dem wahrnehmbaren Bereich der Levelarchitektur indirekt Grenzen zu setzen. Folglich müssen nur die Teile eines Raums angelegt werden, die bei Begehung eventuell sichtbar werden können, weil sie sich im Blickfeld einer der Kamera-Perspektiven befinden.

Ein deutlicher Vorteil statischer Kamerawinkel gegenüber der Verfolgerkamera ist zweifellos die Einfachheit der Umsetzung. Die Implementierung einer komfortablen Verfolgungsfunktion ist ein kompliziertes Feld, gilt es doch einige Fallstricke zu überwinden. Eine solche Kamera muss zum Beispiel fähig sein, automatisch einen versetzten Blickwinkel zu suchen, sollte der Bereich, in dem sich die Spielfigur befindet, einmal durch ein Objekt dauerhaft verdeckt werden, das sich näher an der Kameralinse befindet. Gute Lösungen sind schwer zu finden, wie man daran erkennen kann, dass die Verfolgerkamera selbst in kommerziellen Produkten oft unzuverlässig arbeitet. Aufgrund der praktischen Eigenschaften der starren Kamera und der gerade erwähnten Komplexität der Verfolgerkamera, werde ich für das Framework ersteres Kamera-Prinzip realisieren.

4.2.2.4 Entwicklung des Dateiformats zur Spezifikation von Rauminhalten

Um ein Dateiformat für Adventure-Räume zu entwickeln, muss erstens definiert werden, welche Informationen in die Datei aufzunehmen sind und zweitens verbindlich festgehalten werden, nach welcher Vorschrift sie strukturiert werden sollen.

Bestandteile eines Raums

Berücksichtigt man auch die Kapazitäten der Game-Engine (z.B. Partikeleffekte), so erhält man die folgende Menge (Abbildung 38), welche alle potenziellen Bestandteile eines Raums enthält.

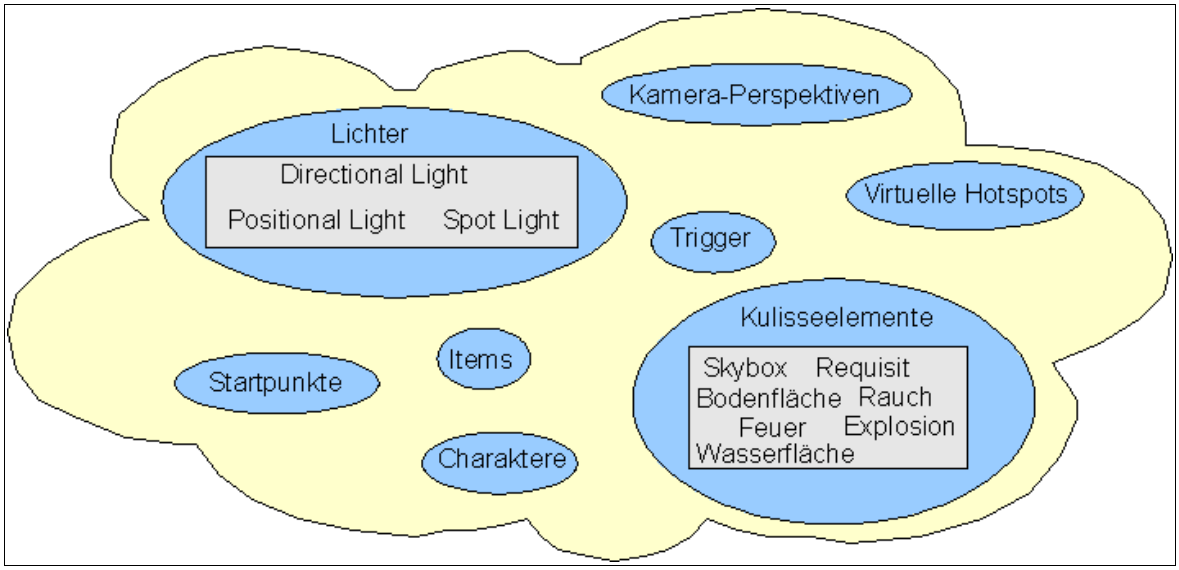


Abbildung 38: Mögliche Bestandteile eines Raums

Organisation der Bestandteile in ein festes Format

Nachdem die potenziellen Bestandteile aufgezählt wurden, kann die Festlegung in ein verbindliches Format erfolgen. Ich entscheide mich für nachstehend präsentierte Struktur (Abbildung 39), deren Sektionen theoretisch jedoch auch in anderer Reihenfolge angeordnet werden könnten. Wichtig ist nur, dass verwandte Komponenten in logische Gruppen zusammengefasst werden, um ein systematisches Einlesen der Informationen zu ermöglichen.

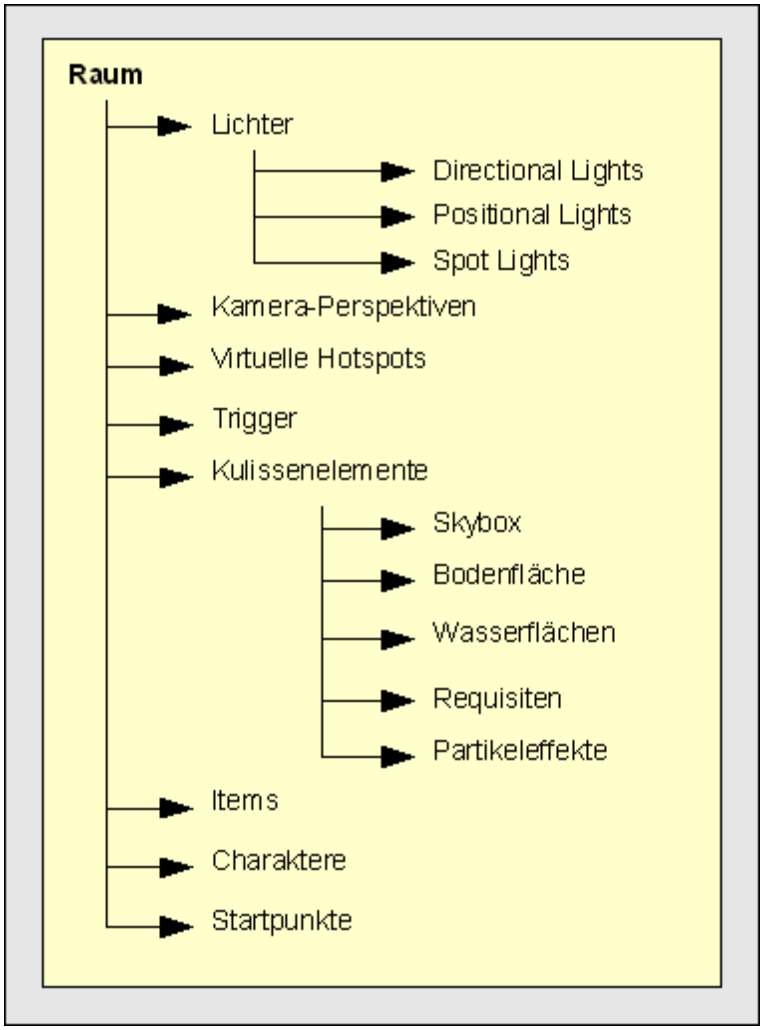


Abbildung 39: Spezifikation des Dateiformats für Raum-Dateien

4.2.2.5 System zur sequenziellen Ausführung von Aktionen und Ereignissen

In den Anforderungen (Kapitel 4.2.1.6) wurde verdeutlicht, warum es sinnvoll ist, Spielaktionen und Ereignisse sequenziell ausführen zu können. In diesem Kapitel wird deshalb erklärt, wie ein sequenziell arbeitendes System innerhalb des Frameworks realisiert werden kann.

Der Grundgedanke hinter dem System ist die Schaffung eines Pufferspeichers, dem in geordneter Form, das heißt nacheinander, Handlungsanweisungen hinzugefügt werden können. Die Anweisungen werden nicht zwangsläufig unmittelbar nach dem Hinzufügen ausgeführt, sondern erst dann, wenn sie an der Reihe sind. Die Bestimmung der Reihenfolge ist nach dem FIFO-Prinzip (first in, first out) zu vollziehen. Der Befehl, der zuerst eingefügt wird, wird demnach als erstes aktiviert. Nach dem Ende seiner Lebensdauer entfernt man ihn vom Puffer, so dass die nächsten Aktionen in der Liste nachrücken können. Abbildung 40 veranschaulicht das Prinzip in einer übersichtlichen Grafik.

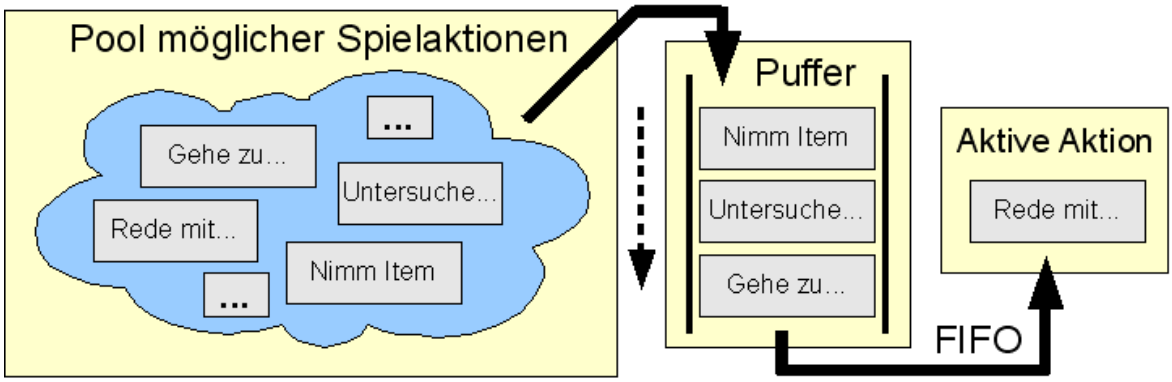


Abbildung 40: Abarbeitung sequenzieller Befehlsfolgen

4.2.2.6 Ein einfaches Inventarsystem

Kein Point-and-Click-Adventure klassischer Prägung kommt ohne Inventar aus. Eingesammelte Items müssen unkompliziert ausgewählt werden können, damit ein flüssiger Spielablauf gewährleistet bleibt. Im Genre haben sich vor allem zwei grundlegende Varianten von Inventarsystemen durchgesetzt.

Die erste Variante tritt meist in Form eines waagerechten Balkens am oberen oder unteren Bildschirmrand auf. Die enthaltenen Items werden von links nach rechts in einer Reihe angeordnet. Jeder Gegenstand im Inventarbalken wird durch eine kleine Grafik symbolisiert. Ein Mausklick auf das entsprechende Bildchen führt zur Selektion des assoziierten Items für die weitere Verwendung im Spiel. Die Abbildung 41 skizziert die Struktur des beschriebenen Inventarsystems.

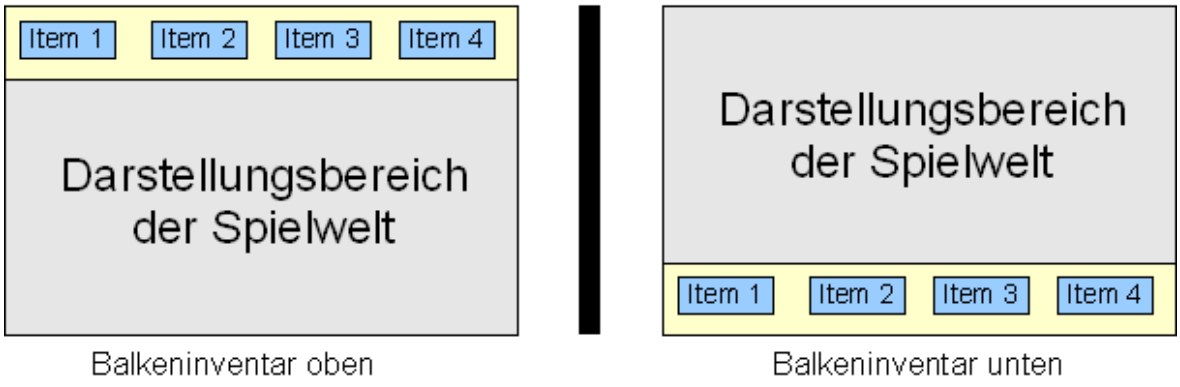


Abbildung 41: Skizzierung der Inventarvariante "Balken"

Variante Nummer Zwei setzt auf eine matrixartige Anordnung der Items. Der Inhalt des Inventars wird in Zeilen- und Spaltendarstellung auf dem Bildschirm platziert. Viele Spiele übertragen diese Form in die nahe liegende Analogie einer rechteckigen Schachtel oder Kiste. Die Darstellung beansprucht in der Regel einen großen Teil der verfügbaren Bildschirmfläche. Aus diesem Grund wird das Inventar nur so lange angezeigt, bis ein Item selektiert wurde. Danach wird es ausgeblendet, um die Darstellung der Spielwelt in den Vordergrund zu rücken. Abbildung 42 illustriert, wie man sich das Inventarsystem im geöffnetem Zustand vorzustellen hat.

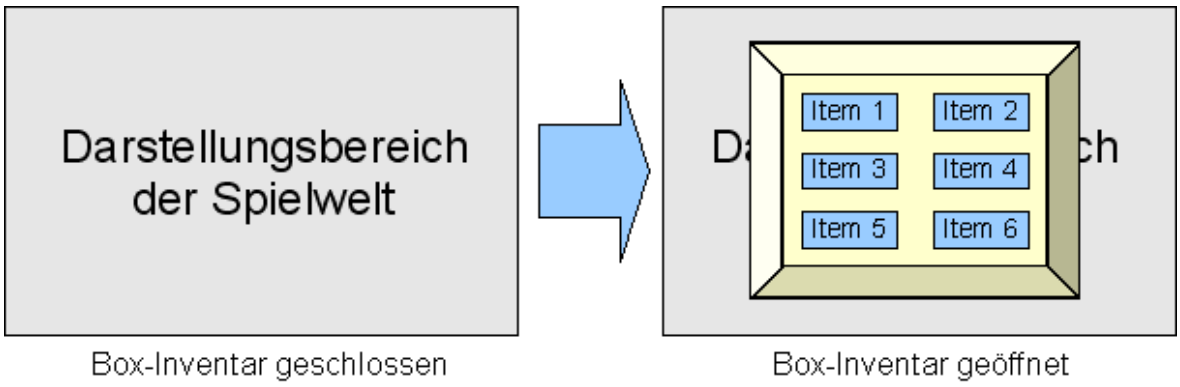


Abbildung 42: Skizzierung der Inventarvariante "Box"

Ich werde im Adventure-Framework die Variante „Balken-Inventar“ implementieren. Durch ein generisches Vorgehen, ist es jedoch nicht ausgeschlossen, nachträglich ein Box-Inventar in das System aufzunehmen.

4.2.2.7 Entwurf des Dialogsystems

Dialogsysteme in Spielen lassen sich häufig durch folgenden Aufbau beschreiben: Ein Dialogmenü bietet verfügbare Themen zur Auswahl an. Hinter jedem Thema verbirgt sich dann ein Gesprächszweig, der sich wiederum aus einzelnen Sätzen der Dialogpartner zusammensetzt. Folgende Grafik (Abbildung 43) verdeutlicht den geschilderten Zusammenhang bildlich.

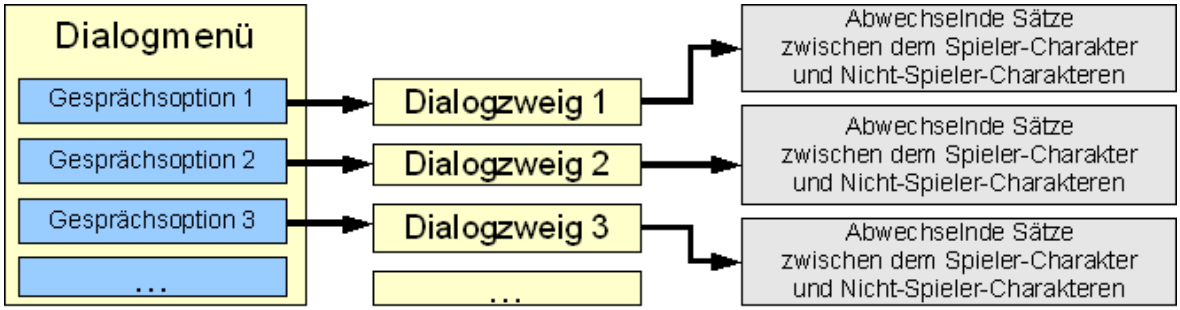


Abbildung 43: Schema des Aufbaus eines Dialogsystems

Das Dialogsystem des Adventure-Frameworks wird sich grundlegend an diese Architektur halten.

Ein Dateiformat für Dialogdateien

Äquivalent zur Spezifikation der Rauminhalte, sollen Dialoge strukturiert in persistente Dateien abgelegt werden. Es empfiehlt sich erneut, ein festes Format zu bestimmen, damit sie leicht maschinell eingelesen werden können. Eine einzelne Datei wird dabei die Dialoge eines ganzen Spiel-Kapitels speichern. Abbildung 44 zeigt die vorgesehene Grundstruktur des Dateiformats zum Speichern von Dialogdaten.

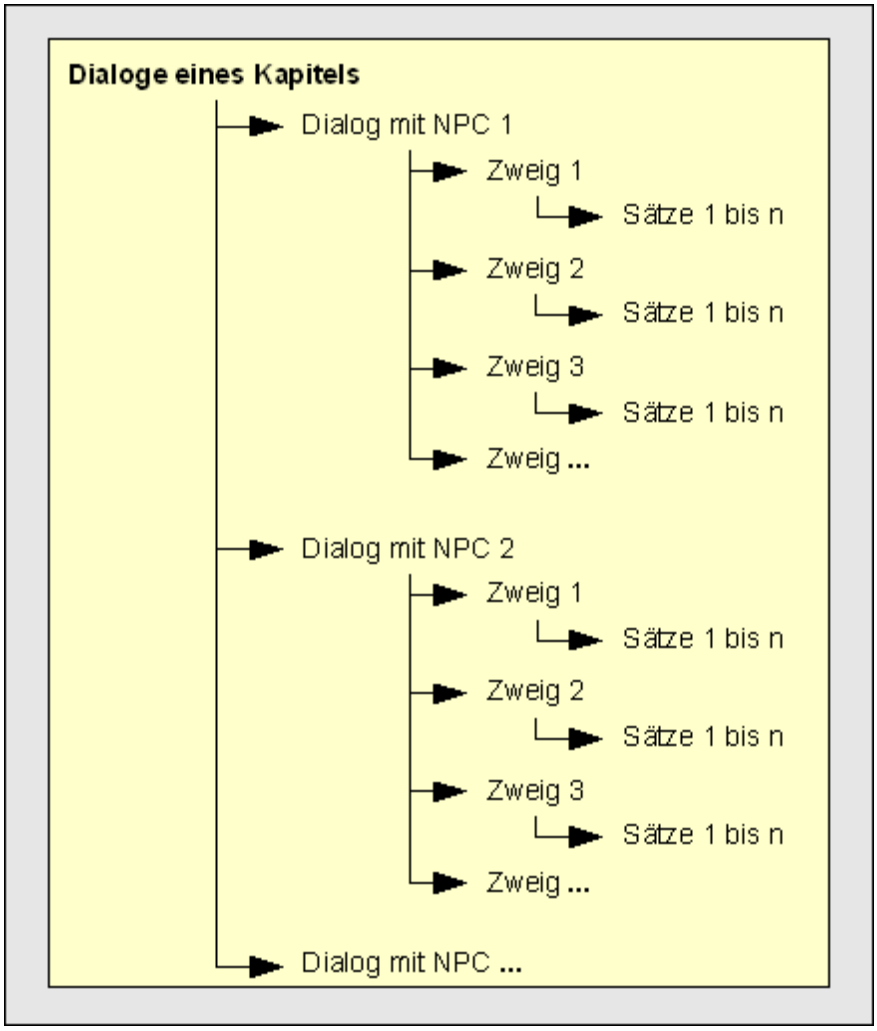


Abbildung 44: Spezifikation des Dateiformats für Dialog-Dateien

4.2.2.8 Die Mechanik der Kommandoschnittstelle zur Spielwelt

Die Spiel-Steuerung der mit dem Framework erstellten Adventures, wird genretypisch durch die Maus erfolgen. Sie reagiert dabei kontextsensitiv auf Hotspots, die in der Spielwelt verteilt sind. Durch den Mausklick auf einen solchen Punkt, ist eine Spielaktion auszuführen, die sich aus dem Zusammenhang ergibt. Das Prinzip der kontextsensitiven Benutzerschnittstelle ist eine erhebliche Erleichterung der Adventure-Steuerung. Im Vergleich zum früher üblichen Verbensystem wirkt sie straff und intuitiv. In diesem Entwurfskapitel gilt es nun zu bestimmen, welche Situation welches Kommando zur Folge haben soll. In der Tabelle aus Abbildung 45 wird somit das Spielverhalten definiert, welches bei Klicks mit der linken Maustaste auf Hotspots zum tragen kommen soll.

Hotspot-Typ	Mausklick ohne gewähltes Inventar-Item	Mausklick mit gewähltem Inventar-Item
Item	Item einstecken	-- Keine Aktion vorgesehen --
Nicht-Spieler-Charakter	Konversation starten	Benutze Item mit dem Nicht-Spieler-Charakter
Virtueller Hotspot	Benutze Hotspot (gegebenenfalls nicht ohne gewähltes Item möglich)	Benutze Item mit Hotspot

Abbildung 45: Kontextsensitives Verhalten bei Links-Klicks auf Hotspots

Der rechten Maustaste werden weniger Funktionen zugesprochen. Erfolgt im normalen Spielzustand ein Rechts-Klick auf einen Hotspot, so soll der Hotspot untersucht, das heißt, genauer betrachtet werden. Ansonsten dient die rechte Maustaste zur Abwahl des gegenwärtig selektierten Inventar-Items.

4.2.2.9 Implementierungsvorgaben für das Game-Scripting

Blickt man auf die Kapitel zu den technologischen Grundlagen zurück, fällt auf, dass dort die Skriptsprache BeanShell näher beleuchtet wurde (Kapitel 3.4.2). Dies geschah selbstverständlich mit der Aussicht auf die zukünftige Verwendung im Framework. In diesem Kapitel beschreibe ich nun, wie das Game-Scripting im Adventure-System umzusetzen ist.

Die wichtigste Voraussetzung für den Zugriff auf Programm-Objekte durch BeanShell-Skripte, ist das Binden der Objekte an den Skript-Interpreter. Zu diesem Zweck werden eindeutige Identifikationsnamen für die Objekte benötigt, über die sie in den Skripten referenziert werden können. Durch die Aufnahme der Referenz-Bezeichner in das Dateiformat der Raum-Dateien kann die Information persistent gespeichert werden. Für jeden Raum muss genau eine Skriptdatei angelegt werden, die das Verhalten seiner interaktiven Elemente beschreibt.

Der Code in den Skriptdateien hat die Aufgabe, die Spiellogik eines Adventures festzulegen. Allen Interaktionsmöglichkeiten eines Raums sollen spezielle Skript-Methoden zugeordnet werden, die den weiteren Spielverlauf regeln. Die konkrete Implementierung der Methoden macht im Grunde das Herzstück eines Adventures aus. Folgende Tabelle (Abbildung 46) gibt wieder, für welche potenziellen Interaktionsmöglichkeiten Skript-Methoden zu implementieren sind.

Hotspot-Typ	Untersuchen	Benutzen	Einsammeln	Unterhalten
Item				
Nicht-Spieler-Charakter				
Virtueller Hotspot				

Abbildung 46: Überblick der notwendigen Skript-Methoden für Hotspots

Ein Wort zur Erläuterung. Man könnte einwenden, dass für Items in der Spalte „Benutzen“ ein Plus-Zeichen stehen müsste, weil der Zweck von Items doch gerade ihre Benutzung ist. Doch gilt es zu bedenken, dass es um das Benutzen von Hotspots geht, die sich aktiv in der Spielwelt befinden. Items können jedoch erst dann benutzt werden, wenn sie physikalisch aus der Szene entfernt und dem Inventar hinzugefügt wurden. Dann aber dürfen sie auf NPCs und virtuelle Hotspots angewandt werden. Es ist allerdings nicht vorgesehen, sie mit anderen Items in der Welt zu kombinieren.

4.2.3 Implementierung

Die Anforderungen an das Adventure-Framework wurden abgesteckt und der Entwurf zentraler Komponenten beschrieben. Wie schon in den Ausführungen zur 3D-Game-Engine, folgt in der letzten Phase die nähere Beschreibung der Implementierung des Frameworks.

4.2.3.1 Implementierung der Szenekomponenten

Die Komponenten Requisit, Item, Charakter, virtueller Hotspots und Trigger sind in spezialisierte Klassen gekapselt worden. Sieht man von den Triggern ab, erweitern sie allesamt die abstrakte Basisklasse `AbstractEntity`. Diese Klasse stellt gemeinsame Grundfunktionen bereit. Dazu zählen insbesondere die Methoden zur Zuweisung und Abfrage der komponenteneigenen Skript-ID. Bei der Skript-ID handelt es sich um den eindeutigen Referenznamen, der den Zugriff auf das Objekt in einem BeanShell-Skript vermittelt. `AbstractEntity` bietet aber auch die Möglichkeit, einen zuvor zugewiesenen Bildschirmnamen der Komponente an beliebiger Stelle auf dem Monitor auszugeben. Dabei wird vom Text-Renderer der 3D-Engine Gebrauch gemacht.

Die Spezialisierung der abgeleiteten Klassen erfolgte durch die Implementierung von Attributen und Methoden, die auf die jeweilige Komponente zugeschnitten wurden. Während die Umsetzung der Requisiten und Items mit Hilfe zweier überschaubarer Java-Klassen (`Prop` und `Item`) durchgeführt werden konnte, war es angebracht die Charaktere und virtuellen Hotspots weiter aufzugliedern. Zum einen wurde eine abstrakte Oberklasse `AbstractCharacter` eingeführt, von der sich die beiden Klassen `NonPlayerCharacter` für Nicht-Spieler-Charaktere und `PlayableCharacter` für Spieler-Charaktere ableiten. Zum anderen existiert mit `AbstractInteractiveHotspot` eine abstrakte Klasse für virtuelle

Hotspots. Ihre konkrete Implementierung führte zu den Klassen `CubicalHotspot` und `SphericalHotspot` für quader- beziehungsweise kugelförmige Instanzen.

Die Trigger-Klassen verfügen über eine eigene Ableitungshierarchie außerhalb von `AbstractEntity`, da sie sich von ihrem Wesen her nicht nahtlos eingefügt hätten. Trigger im Rahmen des Frameworks sind sensitive Levelmarkierungen, die ausschließlich dazu dienen, unter bestimmten Umständen (ein Charakter betritt den Triggerbereich) Skriptcode auszuführen. In der Spielwelt treten sie weder sichtbar hervor noch können sie gezielt mit der Maus angesprochen werden. Ihre Implementierung erfolgte über die abstrakte Klasse `AbstractTrigger`. Zwei Arten von Triggerformen, die auf `AbstractTrigger` basieren, werden unterstützt: Kreisförmige Trigger nutzen die Klasse `CircleTriggerArea`, während rechteckige Triggerflächen auf `RectTriggerArea` beruhen. Das UML-Diagramm in Abbildung 47 fasst die Vererbungsbeziehungen zwischen den Klassen dieses Kapitels zusammen.

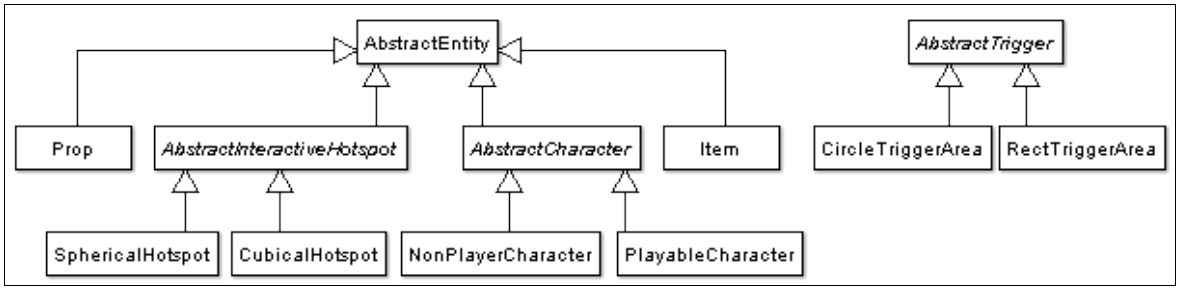


Abbildung 47: Klassenhierachie der verschiedenen Szenekomponenten eines Raums

4.2.3.2 Strukturierungsbeziehungen von Kapitel und Räumen

Adventures, die man mit dem Framework erstellt, können in einzelne Kapitel unterteilt werden. Jedes Kapitel kann über eine beliebige Menge an Räumen verfügen. Nur die Daten des aktuellen Kapitels werden dabei im Arbeitsspeicher vorgehalten. Ladezeiten beim Wechsel von einem Raum zu einem anderen fallen weg. Lediglich zu Beginn eines Kapitels müssen gewisse Ladezeiten in Kauf genommen werden, die von der Komplexität der Raumarchitekturen abhängen. Abbildung 48 zeigt die Zusammenhänge zwischen den Bestandteilen von Kapiteln und Räumen in einem UML-Klassendiagramm. Eine Erläuterung folgt im Anschluss.

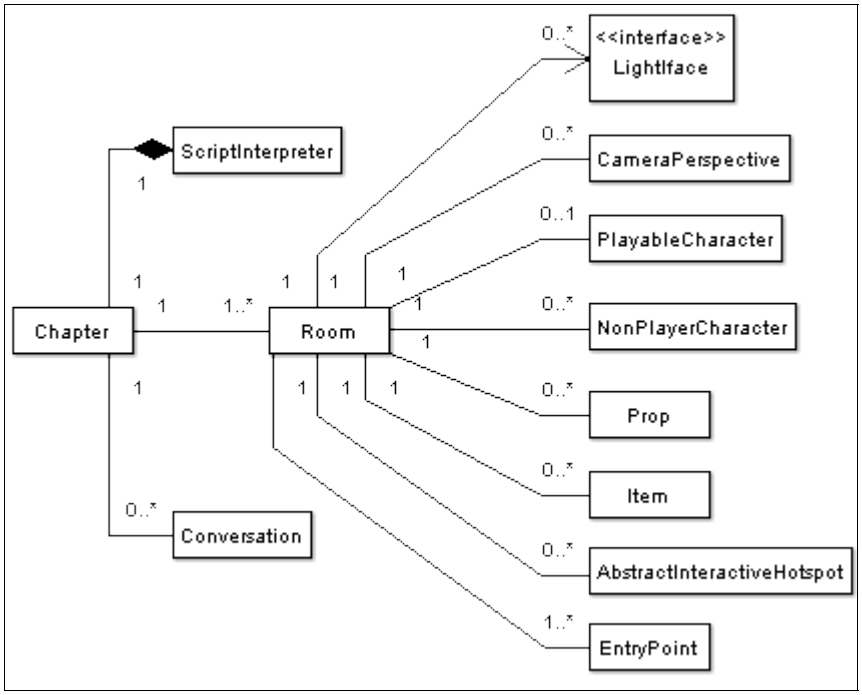


Abbildung 48: Die Klassenbeziehungen zwischen Kapitel- und Raumbestandteilen

Eingangs ist festzustellen, dass jedes Kapitel über eine Instanz der Klasse `ScriptInterpreter` verfügt. Sie ist als Vermittler zwischen Spiel-Objekten und Skriptcode zu verstehen. Dem Skript-Interpreter werden alle skriptrelevanten Spiel-Objekte eines Kapitels bekannt gemacht. Auf diese Weise ist es möglich, von beliebigen Räumen aus, dynamische Veränderungen an Spiel-Objekten vorzunehmen, die sich in irgendeinem (anderen) Raum des Kapitels befinden. Beispielsweise könnte demnach eine Aktion in Raum A zu einer modifizierten Situation in Raum B führen, die ein Voranschreiten zu Raum C erlaubt. Diese raumübergreifenden Handlungsmöglichkeiten sind übrigens auch der Grund dafür, dass nicht nur die Daten des aktuell besuchten Raums im Speicher gehalten werden, sondern immer die Daten eines ganzen Kapitels.

Des weiteren verwaltet jedes Kapitel alle seine Multiple-Choice-Dialoge zwischen Spieler und Nicht-Spieler-Charakteren. Deren Anzahl ist theoretisch unbeschränkt. Eine Liste von `Conversation`-Objekten übernimmt diese Aufgabe intern.

Die Räume eines Kapitels setzen sich aus Referenzen zu Instanzen einer ganzen Reihe von Klassen zusammen. Im Grunde entspricht deren Aufzählung den Komponenten des entworfenen Dateiformats für Raum-Dateien in Kapitel 4.2.2.4. Besonderer Erwähnung bedarf es nur bezüglich der abgebildeten Kardinalitäten. Die Menge der Lichter, Kamera-Perspektiven, Requisiten, Items, NPCs und virtuellen Hotspots erstreckt sich im Bereich von null bis beliebig vielen Instanzen. Dem gegenüber kann ein Raum maximal einen gespielten Charakter enthalten. Außerdem muss für einen Raum mindestens *ein* Startpunkt, repräsentiert durch Instanzen der Klasse `EntryPoint`, bestimmt werden, damit die kontrollierte Spielfigur beim Betreten eine vordefinierte Position einnehmen kann.

4.2.3.3 Die Datei zur Spezifikation von Kapiteln

Jedes Kapitel eines Adventures muss in einer Kapitel-Datei mit der Endung *„.chp“* spezifiziert werden. Sie enthält unter anderem Verweise auf die beteiligten Raum-Dateien und gibt das Spieler-Modell an. Genau wie die Raum-Dateien selbst, werden die Kapitel-Dateien in einem XML-Textformat verfasst. Über die komfortable Verarbeitungsbibliothek *dom4j* wird es eingelesen. Das ist womöglich etwas langsamer, als die Verwendung von Binärdateien, jedoch bleiben die Vorteile der Menschenlesbarkeit und der manuellen Erstell- sowie Manipulierbarkeit des XML gewahrt. In Abbildung 49 ist eine exemplarische Kapitel-Datei zu sehen. Die Zahlenmarkierungen geben Bereiche des Dokuments an, die ich nun Stück für Stück genauer erläutern möchte.

1. In dieser Zeile kann ein Ladebildschirm spezifiziert werden. Die Pfadangabe muss auf eine Bilddatei verweisen.

2. Dieser Eintrag gibt den Pfad zu einer Datei an, die die Multiple-Choice-Dialoge des Kapitels enthält. Er ist optional, da es möglich sein soll, Kapitel ohne Konversationen zu entwerfen.

3. An dieser Stelle werden die Pfadangaben zu den einzelnen Raum-Dateien des Kapitels angegeben. Das Attribut `cam` gibt an, welche Kamera-Perspektive initial für den Raum gelten soll.

4. Das Start-Tag zur Definition der Spielerfigur. Über den Parameter `scriptid` erhält sie eine Referenz-ID für den Zugriff in den Skript-Dateien. Das Attribut `type` gibt die Art des verwendeten 3D-Modell-Formats an und `smooth` legt fest, ob für das Modell automatisch weiche Normalen berechnet werden sollen, die es weniger kantig erscheinen lassen. Der `shadow`-Parameter bestimmt, ob die Figur einen Schatten werfen soll, falls ein Licht für den Schattenwurf vorgesehen wurde.

5. In dieses Tag wird die Skript-ID des Raums eingetragen, in dem der Spieler eingangs starten soll.

6. Diese Zeile entscheidet über die Art des zu verwendenden Inventar-Systems. Gegenwärtig wird nur das Balken-Inventar unterstützt, welches durch den Eintrag `"bar"` gekennzeichnet wird.

7. Hier wird der konkrete Name des Spieler-Charakters im Adventure angegeben.

8. Es folgt die Pfad-Angabe zur 3D-Modell-Datei der Spielfigur und anschließend der Pfad zur assoziierten Textur.

9. Abschließend werden die Animationssequenzen des Spielers und seine initiale Orientierung festgelegt. Die Beschreibung der Animationen geschieht durch mehrere erläuterungsbedürftige Attribute. Zuerst wird ein Referenzname angegeben, danach folgt der Framebereich sowie die Zeitdauer, für die ein Frame sichtbar sein soll. Die optionalen Attribute `sound` und `loopsound` verknüpfen ein Geräusch (im Mp3-Format) mit der Animationssequenz beziehungsweise definieren, ob das Geräusch während der Dauer der Animation fortlaufend wiederholt werden soll.

```
<chapter>
  <loadscreen file="./images/screens/load_screen.png"/> 1
  <conversations file="./locations/chapter1/chapter1.con"/> 2
  <room file="./locations/chapter1/buro.room" cam="0"/>
  <room file="./locations/chapter1/map_chicago.room" cam="0"/>
  <room file="./locations/chapter1/bibliothek_foyer.room" cam="0"/> 3
  <player id="player" scriptid="player" type="ms3dascii" smooth="false" shadow="true"> 4
    <room>buro</room> 5
    <inventory type="bar"/> 6
    <name>Alfons Catraz</name> 7
    <file>./models/ms3d/player/player.txt</file>
    <texture unit="0" mipmap="true" rotate="false" flip="false">./texture.jpg</texture> 8
    <animation name="walk" start="1" end="9" timeperframe="75" sound="./footsteps.mp3" loopsound="true"/>
    <animation name="pickup" start="10" end="12" timeperframe="115" sound="./zipper.mp3" loopsound="false"/>
    <animation name="give" start="10" end="12" timeperframe="115"/>
    <animation name="idle" start="16" end="22" timeperframe="85"/>
    <position x="-3.0" y="0.0" z="-4.0"/>
    <rotation x="0.0" y="0.0" z="0.0"/>
    <scale x="0.15" y="0.15" z="0.15"/>
  </player>
</chapter>
```

9

Abbildung 49: Beispiel für eine Kapitel-Datei

4.2.3.4 Die Datei zur Spezifikation von Räumen

Die Spezifikation der Inhalte von Räumen des Adventures wird in einer XML-Datei mit der Endung „*.room*“ vorgenommen. Wie im vorherigen Kapitel werde ich nun markante Punkte einer solchen Datei näher beleuchten. Die Beschreibung geschieht anhand einer präparierten Beispieldatei. Ich werde sie abschnittsweise erörtern, da sie zu umfangreich ist, um sich als Ganzes passend ins Seitenlayout zu fügen.

Spezifikation der Lichtquellen

Abbildung 50 zeigt, wie die verschiedenen Typen von Lichtquellen im XML-Dokument der Raum-Datei angegeben werden. Bevor ich darauf eingehe, möchte ich jedoch kurz auf das Wurzel-Element `<room>` zu sprechen kommen. Wie alle spielrelevanten Entitäten, bekommen auch Räume eine Skript-ID zugewiesen, über die das Raum-Objekt in Skripten referenziert werden kann. Über das Attribut `scriptfile` wird dem Raum eine Skriptdatei zugeordnet, die den Code für dessen Hotspots enthält.

```
<room scriptid="bibliothek_foyer" scriptfile="./bibliothek_foyer.bsh">
  <lights shadowref="0">
    <positional nr="0">
      <position x="0.0" y="50.0" z="0.0"/>
      <ambient r="0.0" g="0.0" b="0.0" a="1.0"/>
      <diffuse r="1.0" g="1.0" b="1.0" a="1.0"/>
    </positional>
    <directional nr="1">
      <direction x="0.26726124" y="0.5345225" z="-0.80178374"/>
      <ambient r="0.0" g="0.0" b="0.0" a="1.0"/>
      <diffuse r="1.0" g="1.0" b="1.0" a="1.0"/>
    </directional>
    <spot nr="2">
      <position x="0.0" y="0.0" z="0.0"/>
      <ambient r="0.0" g="0.0" b="0.0" a="1.0"/>
      <diffuse r="1.0" g="1.0" b="1.0" a="1.0"/>
      <rotation x="25.0" y="25.0"/>
      <cutoff value="25.0"/>
      <exponent value="0.0"/>
    </spot>
  </lights>
  ...
  ...
</room>
```

Abbildung 50: Spezifikation der Lichtquellen in einer Raum-Datei

Die Angabe der Lichtquellen wird über das `<lights>`-Tag eingeleitet. Danach werden die konkreten Lichter spezifiziert. Der `nr`-Parameter verweist auf die Verwendung des OpenGL-Lichts mit der entsprechenden Nummer. Weiterhin werden spezielle Initialisierungsdaten, wie Position und Farbe zugewiesen. Bisher können nur die gezeigten Einstellungen in XML vorgenommen werden. Einige OpenGL-Lichtparameter, wie zum Beispiel der Dämpfungsfaktor, lassen sich also nicht konfigurieren. Prinzipiell ist es zwar möglich diese Werte stattdessen im Skript-Code zu setzten, doch sollte im Sinne der Einheitlichkeit eine spätere Ergänzung des Dateiformats um die relevanten Faktoren angestrebt werden.

Spezifikation der Kamera-Perspektiven

Abbildung 51 zeigt das Vorgehen zur Definition der Kamera-Perspektiven eines Raums. Die einzelnen Perspektiven werden durch das `<camera>`-Tag eingerahmt. Der numerische Ganzzahlwert `id` identifiziert den jeweiligen Blickwinkel innerhalb einer Raumdatei eindeutig, während das `fov`-Attribut den Field-of-View-Winkel der sinnbildlichen Kameralinse spezifiziert. Die Orientierung der Perspektiven erfolgt durch die Angabe von dreidimensionalen Positionskoordinaten sowie Rotationswinkeln. Die nahe und ferne Clipping-Ebene wird durch das `<clipping>`-Tag ebenfalls im Dokument verhaftet.

```
<room scriptid="bibliothek_foyer" scriptfile="./bibliothek_foyer.bsh">
  ...
  ...
  <camera>
    <perspective id="0" fov="45.0">
      <position x="-0.72" y="7.0" z="9.88"/>
      <rotation x="23.5" y="-10.8"/>
      <clipping near="0.1" far="2000.0"/>
    </perspective>
    <perspective id="1" fov="45.0">
      <position x="-0.29" y="4.68" z="-2.59"/>
      <rotation x="23.5" y="-105.9"/>
      <clipping near="0.1" far="2000.0"/>
    </perspective>
    <perspective id="2" fov="45.0">
      <position x="-1.06" y="8.29" z="2.94"/>
      <rotation x="51.7" y="-52.9"/>
      <clipping near="0.1" far="2000.0"/>
    </perspective>
  </camera>
  ...
  ...
</room>
```

Abbildung 51: Spezifikation der Kamera-Perspektiven in einer Raum-Datei

Spezifikation der Kulissen-Elemente/Requisiten

Kulissen-Elemente werden innerhalb des `<props>`-Tag-Paars angegeben. Gegenwärtig lassen sich vor allem eine Bodenebene, und 3D-Modelle als Bestandteile der Kulisse eines Raums spezifizieren. Transparente Wasserflächen und Partikeleffekte können ebenfalls im Dokument initialisiert werden, jedoch ist ihre Handhabung noch etwas unausgereift. Stattdessen ist es komfortabler, sie direkt in der assoziierten Skript-Datei zu formulieren. Nacheinander zeige ich nun exemplarisch, wie der entsprechende XML-Text für das Einbinden der erstgenannten Elemente ausschaut.

In Abbildung 52 ist zu sehen, wie man die Bodenebene zu einem Raum hinzufügt. Das `<ground>`-Tag leitet die Deklaration ein. Der Wert `"plane"` des Attributs `type`, identifiziert, die Bodenfläche als Ebene. Für die Zukunft ist es denkbar, weitere Typen von Bodenflächen in das Adventure-Framework aufzunehmen. Height-Map-Terrains sind potenzielle Kandidaten in diesem Zusammenhang. Der Parameter `size` erwartet Zweierpotenzen als Eingabewerte. Bei anders lautenden Eingaben wird eine interne Konvertierung zur nächsten Zweierpotenz vorgenommen. Der `texrepeat`-Faktor legt fest, wie oft die Textur, die über die Bodenebene gespannt wird, auf der Fläche wiederholt werden soll. Die Angaben zu Textur-Pfad und Orientierung des Bodens bedürfen keiner weiteren Erläuterung, da sie in gleicher Form bereits wiederholt vorgestellt wurden. Interessant wird es beim `<pathgrid>`-Tag. Hier werden diejenigen Kacheln des Bodengitters für die A*-Wegfindung aufgelistet, auf denen sich die Spielfiguren bewegen dürfen. Das `<pathgrid>`-Attribut `nodesize` erwartet ebenfalls eine Zweierpotenz als Eingabewert. Wie zuvor wird jedoch auch eine Konvertierung zur Zweierpotenz durchgeführt, sollte diesem Kriterium nicht entsprochen werden.

```
<room scriptid="bibliothek_foyer" scriptfile="./bibliothek_foyer.bsh">
  ...
  ...
  <props>
    ...
    ...
    <ground id="ground plane" scriptid="ground" type="plane" size="16" texrepeat="20.0">
      <texture unit="0" mipmap="true" rotate="false" flip="false">./mamor.jpg</texture>
      <position x="0.0" y="0.0" z="0.0"/>
      <rotation x="0.0" y="0.0" z="0.0"/>
      <scale x="1.0" y="1.0" z="1.0"/>
      <pathgrid nodesize="1">
        ...
        <walkable nodeid="0"/>
        <walkable nodeid="1"/>
        <walkable nodeid="2"/>
        ...
      </pathgrid>
    </ground>
    ...
    ...
  </props>
  ...
  ...
</room>
```

Abbildung 52: Spezifikation der Bodenebene

Abbildung 53 demonstriert die Definition zweier Requisiten-Objekte, die auf 3D-Modell-Dateien beruhen. Es wird hervorgehoben, dass sowohl animierte, als auch statische Requisiten möglich sind. Beim ersten Requisit handelt sich um ein Objekt, welches aus einer MilkShape3D-ASCII-Datei eingelesen werden soll. Der Wert `"ms3dascii"` des `type`-Attributs macht dies deutlich. Das `volume`-Attribut kann die Werte `"box"` und `"sphere"` annehmen. Es sagt aus, ob eine Bounding-Box oder eine Bounding-Sphere als Hüll-Volumen für das assoziierte Mesh zu verwenden ist. Durch das Hinzufügen optionaler `<animation>`-Tags, können Animationsphasen für das Objekt bestimmt werden. Dies verhält sich identisch mit den entsprechenden Angaben zum Spieler-Charakter in den Kapitel-Dateien (Kapitel 4.2.3.3). Zu beachten ist allerdings, dass die `<animation>`-Tags nur für Objekte der Typen `"ms3dascii"` und `"md2"` verarbeitet werden. Für Objekte des Typs `"3ds"`, die im Rahmen des Frameworks immer statisch sind, sind sie bedeutungslos und werden ignoriert.

```
<room scriptid="bibliothek_foyer" scriptfile="./bibliothek_foyer.bsh">
  ...
  ...
  <props>
    ...
    ...
    <prop id="maus" scriptid="maus" type="ms3dascii" smooth="false" volume="box" shadow="false">
      <file>./models/ms3d/maus.txt</file>
      <texture unit="0" mipmap="true" rotate="false" flip="false">./maus.jpg</texture>
      <position x="-6.546577" y="1.95" z="2.0"/>
      <rotation x="0.0" y="270.0" z="0.0"/>
      <scale x="0.0075" y="1.0" z="0.01"/>
      <animation name="walk" start="8" end="15" timeperframe="150"/>
      <animation name="idle" start="1" end="7" timeperframe="100"/>
    </prop>
    <prop id="pflanze" scriptid="planze" type="3ds" smooth="false" volume="sphere" shadow="true">
      <file>./models/3ds/plants/plant01a.3ds</file>
      <texture unit="0" mipmap="true" rotate="true" flip="true">./plant01a.jpg</texture>
      <position x="-6.040181" y="1.9318323" z="-1.8847847"/>
      <rotation x="0.0" y="0.0" z="0.0"/>
      <scale x="0.03" y="0.03" z="0.03"/>
    </prop>
    ...
    ...
  </props>
  ...
  ...
</room>
```

Abbildung 53: Spezifikation von (animierten) Requisiten auf Basis von 3D-Modell-Dateien

Spezifikation von Items

Wie Abbildung 54 deutlich zeigt, unterscheidet sich die Spezifikation von Items in einer Raum-Datei nur im Detail von den zuvor angeführten Beispielen bezüglich einfacher Kulissen-Elemente. Hervorzuheben sind jedoch die drei in der Abbildung aufeinander folgenden Tags `<name>`, `<image>` und `<pickable>`. Ersteres enthält den Bildschirm-Namen des Items. Das ist der Name, der angezeigt wird, wenn der Spieler mit der Maus über das Item fährt. Im `<image>`-Tag wird der Pfad zu einer Bilddatei abgelegt, die als symbolische Grafik für das Item im Inventar geführt werden soll. Damit das Item überhaupt im Inventar erscheinen kann, muss es vom Spieler eingesammelt werden. Mittels des `<pickable>`-Tags wird definiert, ob das Item initial ohne weiteres aufgelesen werden darf.

```
<room scriptid="bibliothek_foyer" scriptfile="./bibliothek_foyer.bsh">
  ...
  ...
  <items>
    ...
    <item id="stift_item" scriptid="stift" type="ms3dascii" smooth="false" volume="box" shadow="false">
      <name>Schwarzer Stift</name>
      <image>./images/items/stift.png</image>
      <pickable>true</pickable>
      <file>./models/ms3d/stift/stift.txt</file>
      <texture unit="0" mipmap="true" rotate="false" flip="false">./stift.jpg</texture>
      <position x="-4.5931225" y="0.0" z="-1.2049298"/>
      <rotation x="0.0" y="55.0" z="0.0"/>
      <scale x="0.03" y="0.03" z="0.03"/>
    </item>
    ...
  </items>
  ...
  ...
</room>
```

Abbildung 54: Spezifikation von Items in einer Raum-Datei

Spezifikation von Nicht-Spieler-Charakteren (NPCs)

Bei der Spezifikation von Nicht-Spieler-Charakteren wiederholen sich viele der bereits bekannten Elemente (Abbildung 55). Es ist jedoch zu beachten, dass auch hier gilt, dass Animationen nur für Objekte der Typen "ms3dascii" und "md2" deklariert werden können.

```
<room scriptid="bibliothek_foyer" scriptfile="./bibliothek_foyer.bsh">
  ...
  ...
  <characters>
    ...
    <character id="m1" scriptid="mann" type="ms3dascii" smooth="false" volume="box" shadow="false">
      <name>Bibliothekar</name>
      <file>./models/ms3d/bibliothekar/bibliothekar.txt</file>
      <texture unit="0" mipmap="true" rotate="false" flip="false">./texture.jpg</texture>
      <position x="-7.016505" y="0.0" z="-0.07949951"/>
      <rotation x="0.0" y="90.0" z="0.0"/>
      <scale x="0.1" y="0.1" z="0.1"/>
      <animation name="walk" start="10" end="15" timeperframe="120"/>
      <animation name="pickup" start="8" end="9" timeperframe="300"/>
      <animation name="idle" start="1" end="7" timeperframe="100"/>
    </character>
    ...
  </characters>
  ...
  ...
</room>
```

Abbildung 55: Spezifikation von Nicht-Spieler-Charakteren in einer Raum-Datei

Spezifikation virtueller Hotspots

Virtuelle Hotspots können mit wenig Aufwand in Raum-Dateien integriert werden. Es gibt zwei Typen virtueller Hotspots im Adventure-Framework: Quaderförmige, die über das Attribut `type` mit dem Wert "box" eingeleitet werden und kugelförmige, die den Wert "sphere" benötigen. Abbildung 56 stellt beide Varianten vor. Für jeden virtuellen Hotspot muss angegeben werden, ob er nur in Kombination mit einem Item angesprochen werden kann (`needsitem="true"`), oder sich durch einen einfachen Linksklick, ohne zuvor gewählten Inventar-Gegenstand, benutzen lässt (`needsitem="false"`). Der Angaben zur Position, Bildschirmnamen und räumlicher Ausdehnung bedarf es keiner näheren Erläuterung, da deren Funktion aus dem Kontext hervorgeht.

```
<room scriptid="bibliothek_foyer" scriptfile="./bibliothek_foyer.bsh">
  ...
  ...
  <hotspots>
    ...
    <hotspot scriptid="namensliste" type="box" needsitem="true">
      <position x="-6.538187" y="2.0" z="2.0357165"/>
      <name>Namensliste</name>
      <dimension width="0.9" height="0.1" length="0.6"/>
    </hotspot>
    <hotspot scriptid="knopf" type="sphere" needsitem="false">
      <position x="-5.4694743" y="1.9318366" z="-0.55987835"/>
      <name>Roter Knopf</name>
      <dimension radius="0.65"/>
    </hotspot>
    ...
  </hotspots>
  ...
  ...
</room>
```

Abbildung 56: Spezifikation virtueller Hotspots in einer Raum-Datei

Spezifikation von Trigger-Flächen

Wie bei den virtuellen Hotspots, unterstützt das Adventure-Framework zwei Varianten von Triggerflächen. Da sie für die Anwendung auf flachen Bodenebenen konzipiert wurden, wird aus der Quader- beziehungsweise Kugelform der virtuellen Hotspots jedoch die zweidimensionale Rechteck- beziehungsweise Kreisform. In Abbildung 57 kann das Attribut `type` demzufolge die zwei Werte `"rect"` und `"sphere"` annehmen. Weiterhin erwähnenswert ist das Attribut `enabled`. Es legt fest, ob der Trigger initial dafür empfänglich sein soll, ob die Spielfigur die sensitive Fläche betreten oder verlassen hat.

```
<room scriptid="bibliothek_foyer" scriptfile="./bibliothek_foyer.bsh">
  ...
  ...
  <triggers>
    ...
    <trigger enabled="false" scriptid="triggerOpenDoor" type="rect">
      <position x="-3.8892226" z="-1.5727978"/>
      <dimension width="1.0" length="1.0"/>
    </trigger>
    <trigger enabled="true" scriptid="triggerChangeCam" type="circle">
      <position x="-2.497609" z="-2.7917147"/>
      <dimension radius="0.75"/>
    </trigger>
    ...
  </triggers>
  ...
  ...
</room>
```

Abbildung 57: Spezifikation von Trigger-Flächen in einer Raum-Datei

Spezifikation von Startpunkten (Entry-Points)

In Abbildung 58 wird demonstriert, wie die Startpunkte für den Spieler-Charakter im Dokument anzugeben sind. Neben der Position muss auch die Rotation der Figur eingetragen werden, damit eine anfängliche Blickrichtung mit dem Startpunkt verknüpft wird.

```
<room scriptid="bibliothek_foyer" scriptfile="./bibliothek_foyer.bsh">
  ...
  ...
  <entrypoints>
    ...
    <entrypoint scriptid="entryPointBiblioFoyer">
      <position x="2.5" y="0.0" z="0.5"/>
      <rotation x="0.0" y="-90.0" z="0.0"/>
    </entrypoint>
    ...
  </entrypoints>
  ...
  ...
</room>
```

Abbildung 58: Spezifikation von Startpunkten (Entry-Points) in einer Raum-Datei

4.2.3.5 Sequenzielle Spielaktionen mit dem FIFO-Pufferspeicher

Die sequenzielle Verarbeitung von Spielaktionen ist gemeinsam mit dem Game-Scripting das Herzstück des Adventure-Frameworks. Der notwendige FIFO-Pufferspeicher basiert intern auf der `LinkedList`-Datenstruktur des JDK. Sie ist für diesen Zweck hervorragend geeignet, weil sie spezielle Methoden zur Verfügung stellt, die FIFO-Operationen nachstellen. In der Klasse `ActionScheduler` wurde der FIFO-Pufferspeicher, hier auch Action-Scheduler genannt, schließlich implementiert. Sie folgt dem Singleton-Entwurfsmuster, so dass ein einfacher Zugriff von jeder Stelle des Programmcodes erfolgen kann.

Spielaktionen werden durch die abstrakte Basisklasse `AbstractGameAction` repräsentiert. In von ihr abgeleiteten Klassen, werden sie konkret implementiert. Das Framework verfügt mittlerweile über eine ganze Reihe solcher Implementierungen, die einen ansehnlichen Aktionsfundus darstellen. Bei Bedarf können leicht weitere Aktionen hinzugefügt werden. Über die `add()`-Methode des Action-Schedulers werden Instanzen der Spielaktionen in die Warteschlange zur Verarbeitung eingereiht. Die folgende Tabelle (Abbildung 59) listet die wichtigsten Game-Actions mitsamt kurzer Erläuterung auf. Konkrete Details zu notwendigen Eingabeparametern sind jedoch der Javadoc-Dokumentation zu entnehmen, die dem Projekt beiliegt.

Game-Action-Klasse	Erläuterung
`Walk`	Instanzen dieser Spielaktion lassen einen Charakter einen Pfad entlang laufen, der entweder mit A* berechnet oder von einer serialisierten „.path“-Datei eingelesen wurde
`MoveForward`	Verschiebt ein Requisit, Item oder Charakter in Vorwärtsrichtung
`MoveBackward`	Verschiebt ein Requisit, Item oder Charakter in Rückwärtsrichtung
`MoveLeft`	Verschiebt ein Requisit, Item oder Charakter in Linksrichtung
`MoveRight`	Verschiebt ein Requisit, Item oder Charakter in Rechtsrichtung
`MoveUp`	Verschiebt ein Requisit, Item oder Charakter nach oben
`MoveDown`	Verschiebt ein Requisit, Item oder Charakter nach unten
`FadeEntityToTransparent`	Verändert den Transparenzwert eines Requisits, Items oder Charakters in einem zeitlichen Verlauf
`FadeOut`	Blendet den Bildschirm zu einer bestimmten Farbe hin aus
`Delay`	Verzögert die weitere Ausführung des Action-Schedulers um einen definierbaren Zeitraum
`PlayAnimation`	Spielt vordefinierte Animationsphasen eines Requisits, Items oder Charakters ab. Nach Beendigung wird die vorherige Animation wieder aktiviert
`SetAnimation`	Wechselt die aktive Animation eines Requisits, Items oder Charakters
`PlayCameraFlight`	Spielt einen aufgezeichneten Kameraflug ab, der aus einer serialisierten „.rec“-Datei eingelesen wird
`PlayMP3`	Dient zum Abspielen einer Audiodatei im MP3-Format
`ShowScreen`	Zeigt eine Bilddatei über die gesamte Größe des Spielbildschirms an
`SetToCamera`	Dient zum selektieren einer (anderen) Kamera-Perspektive
`EnterRoom`	Lässt den Spieler-Charakter den gegenwärtigen Raum verlassen und einen anderen betreten
`SetChapter`	Dient zum Wechseln des Kapitels
`RunCode`	Führt übermittelten Script-Code im Action-Scheduler sequenziell aus

Abbildung 59: Tabelle wichtiger Spielaktionen, die sequenziell ausgeführt werden können

Es mag verwundern, dass in der Tabelle so zentrale Aktionen, wie das Reden oder das Einsammeln von Items fehlen. Dazu sei gesagt, dass solche Aktionsklassen zwar existieren, sie jedoch in der Regel nicht direkt angesprochen werden müssen. Alle Charaktere verfügen über spezielle Methoden zum Sprechen von Sätzen, die die jeweilige Game-Action kapseln. Weiterhin hat der Spieler-Charakter zum Beispiel eine Methode erhalten, die das Aufnehmen von Items durchführt, indem intern die korrespondierende Game-Action erstellt und eingereiht wird.

4.2.3.6 Die Umsetzung des Inventarsystems

Das Adventure-Framework unterstützt standardmäßig den Typ Balkeninventar, implementiert in der Klasse `BarInventory`. Mit der abstrakten Oberklasse `AbstractInventory` wurde jedoch ein relativ generischer Ansatz gewählt, der es erlaubt, weitere Inventar-Typen in das System einzufügen.

Der Inventar-Balken befindet sich am oberen Ende des Bildschirms und nimmt seine komplette Breite ein. Zur Maximierung des sichtbaren Bereichs der Spielfläche, wurde das Inventar so angelegt, dass es nur dann erscheint, wenn sich der Mauszeiger in der Nähe des oberen Bildschirmrands befindet. Die Abbildung 60 zeigt den Inventar-Balken. Er ist beispielhaft mit einigen Items gefüllt. Es ist zu erkennen, dass eines der Items grün eingefärbt erscheint. Die grüne Farbe ist ein Hover-Effekt. Sie zeigt an, dass sich der Mauszeiger über dem Item befindet. Unter dem Item ist derweil sein Name zu lesen, so dass deutlicher wird, worum es sich handelt. Ein Linksklick auf ein Item selektiert es für die weitere Verwendung in der Spielwelt, während ein Rechtsklick das gewählte Item wieder ablegt.

Abbildung 60: Anzeige des Inventar-Balkens am oberen Bildschirmrand

Das Inventarsystem ist in der Lage, die Items in gleichmäßigen Abständen anzuordnen. Bisher kann die Liste allerdings nicht seitlich gescrollt werden, so dass man als Spieldesigner

den Überblick bewahren muss, wie viele Items die Spielfigur gleichzeitig mitführen darf, ohne auf Darstellungsprobleme zu stoßen.

4.2.3.7 Implementierung des Dialogsystems

Das Dialogsystem für Multiple-Choice-Unterhaltungen zwischen Spieler und NPCs basiert auf drei wesentlichen Klassen: `Conversation`, `Branch` und `Sentence`. Instanzen von `Conversation` verwalten die einzelnen Zweige eines Dialogs, welche ihrerseits durch `Branch`-Objekte repräsentiert werden. Wählt der Spieler einen Dialogzweig aus, werden der Reihe nach die Dialogabschnitte vorgetragen, die zuvor über `Sentence`-Instanzen mit ihm assoziiert wurden. Erwähnenswert ist, dass parallel zum Text auch Audiodateien abgespielt werden können. Das Framework gestattet somit die Realisierung einer vollständigen Sprachausgabe.

Einhergehend mit dem Dialogsystem sind die XML-Dateien zu nennen, mit denen die Multiple-Choice-Dialoge spezifiziert werden können. Sie beruhen auf dem Dateiformat, das in Kapitel 4.2.2.7 entworfen wurde und haben per Definition die Dateiendung „*.con*“ als Kurzform für *conversation*. Der nachstehende Auszug aus einer solchen Datei (Abbildung 61) zeigt, wie die Dialoge eines Kapitels korrekt anzugeben sind. Erläuterungen folgen im Anschluss.

```
<conversations>
  ...
  <conversation scriptid="conv1">
    ...
    <branch id="wer" title="Wer bist du?" available="true">
      <sentence by="player">Hallo. Mein Name ist Frank. Wer bist denn du?</sentence>
      <sentence by="mann" code="...Skriptcode...">Ich bin John Rico.</sentence>
    </branch>
    <branch id="durst" title="Hast du etwas zu trinken?" available="false">
      <sentence by="player">Hast du vielleicht etwas zu trinken?</sentence>
      <sentence by="man">Ja klar. Hier ist etwas Wasser.</sentence>
      <sentence by="player" file="./danke.mp3">Vielen Dank.</sentence>
    </branch>
    ...
    <branch id="quit" title="Bis bald" available="true">
      <sentence by="player">Ich gehe dann mal. Bis bald.</sentence>
    </branch>
  </conversation>
  ...
</conversations>
```

Abbildung 61: Spezifikation der Multiple-Choice-Dialoge eines Kapitels

Eine Conversations-Datei kann im Prinzip beliebig viele Einzelkonversationen enthalten. Jede Konversation, eingeleitet durch das `<conversation>`-Tag, wird mit einer Skript-ID bedient, die den manipulativen Zugriff auf ihre Zweige gestattet. Dialogzweige werden innerhalb eines `<branch>`-Tags definiert. Das Attribut `id` dient als interner Identifikationsname, über den ein Zweig innerhalb eines Multiple-Choice-Dialogs ausfindig gemacht und seine Eigenschaften verändert werden können. Eine dieser Eigenschaften wird über den `title`-Parameter gesetzt. Er gibt an, unter welcher Überschrift der Dialogzweig im Auswahlmenü erscheinen soll. Maßgeblich ist jedoch auch, ob eine Dialogoption direkt für den Spieler verfügbar sein soll. Gegebenenfalls kann es durchaus sinnvoll sein, sie erst dann bereitzustellen, wenn eine bestimmte Aufgabe im Spiel erledigt wurde. Das boolesche Attribut `available` legt schließlich fest, ob eine Dialogoption initial angeboten wird. Eine

Besonderheit stellt der Dialogabbruchzweig dar. Wählt der Spieler ihn aus, wird der Dialog beendet. Per Konvention muss er den Identifikationsnamen "quit" bekommen und sollte vorzugsweise an letzter Stelle einer Konversation platziert werden.

Innerhalb der <branch>-Tags werden die einzelnen Sätze (<sentence>-Tags) eines Dialogzweiges spezifiziert. Das Attribut by erwartet die Skript-ID des Charakters, der den Satz sprechen soll. Das optionale Attribut code erlaubt die Ausführung von Skriptcode in Verbindung mit dem gesprochenen Satz. Audiodateien für die Sprachausgabe können ebenfalls mit dem Satz verknüpft werden. Dazu erwartet das (ebenfalls optionale) file-Attribut den Dateipfad zu einer MP3-Datei.

4.2.3.8 Umsetzung der Benutzerschnittstellen-Mechanik

Die Mechanik der Benutzerschnittstelle zur Spielsteuerung ist exakt den Forderungen des Entwurfskapitels 4.2.2.8 nachempfunden worden. Ergänzt wurde sie allerdings um die Möglichkeit zur komfortablen Einblendung aller Hotspots mit Hilfe der dritten Maustaste. Auf diese Weise wird sichergestellt, dass der Spielfluss nicht unterbrochen wird, weil bestimmte interaktive Bereiche eines Raums vom Spieler übersehen werden.

Anzumerken bleibt noch die Verwendung verschiedenartiger Mauszeiger-Symbole, die je nach Kontext zum Einsatz gelangen. Die Tabelle in Abbildung 62 zeigt die dem Projekt beigefügten Maus-Icons, die sich gleichwohl leicht durch andere ersetzen ließen.

Aktion	Icon	Beschreibung
Standard		Der Mauszeiger befindet sich im Ausgangszustand, symbolisiert durch einen einfachen Pfeil.
Benutzen		Das Zahnrad zeigt an, dass sich die Maus über einem Hotspot befindet, der durch einen Links-Klick benutzt werden kann.
Nehmen		Ruht die Maus über einem Item, das mitgenommen werden darf, verwandelt sich der Mauszeiger in dieses Hand-Symbol.
Unterhalten		Das Symbol stellt einen Mund dar. Es erscheint nur, wenn sich die Maus über einem Nicht-Spieler-Charakter befindet, mit der der Spieler eine Unterhaltung führen kann.

Abbildung 62: Übersicht der verwendeten Mauszeiger-Symbole

4.2.3.9 Integration des Game-Scriptings in das Framework

Die Logik eines Adventures auf Basis des entwickelten Frameworks wird in Skriptdateien deklariert. Für jeden Raum muss eine solche Datei erstellt werden. Da BeanShell als Skriptsprache zum Einsatz kommt, erhalten sie die Dateiendung „*.bsh*“.

Die Skriptdatei eines Raums besteht im Wesentlichen aus einer Reihe von Methodenrümpfen, die mit Code zu füllen sind. Durch den Code wird unter anderem beschrieben, welche Auswirkungen Benutzeraktionen auf Hotspots haben. Die Tabelle aus

Abbildung 46 zeigte, für welche Aktion eine Skript-Methode zu erstellen sei. Die Klasse `SkriptInterpreter` kapselt die Verarbeitung von Skript-Code.

Angenommen ein Raum mit der Skript-ID `raum1` enthält das Item mit der Skript-ID `stift`, den NPC mit der Skript-ID `mann`, den virtuellen Hotspot mit der Skript-ID `namensliste` und die Triggerfläche mit der Skript-ID `trigger1`. Dann müsste die zugehörige Skriptdatei mindestens folgende Methoden (Abbildung 63) zur Ausprogrammierung bereitstellen.

```
// Methoden, die beim Betreten und Verlassen
// des Raums "room1" aufgerufen werden
room1OnEnter(){}
room1OnExit(){}

// Item "stift"
stiftOnLookAt(){}
stiftOnPickUp(){}

// Nicht-Spieler-Charakter "mann"
mannOnLookAt(){}
mannOnTalkTo(){}
mannOnUse(){}

// virtueller Hotspot "namensliste"
namenslisteOnLookAt(){}
namenslisteOnUse(){}

// Trigger "trigger1"
trigger1OnEnter(){}
trigger1OnExit(){}
```

Abbildung 63: Beispielhafte Methodenrümpfe für verschiedene Hotspot-Typen

Die Methodennamen werden dabei aus der jeweiligen Skript-ID und dem Namen der Spielaktion zusammengesetzt. Wie zu erkennen ist, sind nicht nur Hotspots zu berücksichtigen. Für jeden Raum ist eine `...onEnter()` und eine `...onExit()`-Methode zu erstellen, die immer dann aufgerufen werden, wenn der Raum vom Spieler betreten beziehungsweise verlassen wird. In gleicher Weise verhält es sich mit den Triggerflächen eines Raums.

Innerhalb der Skript-Dateien steht die vollständige Bandbreite der BeanShell-Syntax zur Verfügung, die äußerst kongruent zur Java-Syntax ist. Es können also nach Belieben die Klassen des Frameworks und der 3D-Engine referenziert werden, um die Logik eines Spiels zu programmieren. Zu beachten ist aber, dass Klassen vor ihrem Einsatz, gemäß den Richtlinien Javas, importiert werden müssen. Viele Spiel-Objekte können über eine Skript-ID direkt angesprochen werden. Manche Objekte, die für die Skript-Programmierung sehr bedeutsam sind, aber keine Skript-ID besitzen, wurden dem Skript-Interpreter unter festen Referenznamen bekannt gemacht. Abbildung 64 gibt einen Überblick.

Referenzname	Angesprochenes Objekt
`camera`	Liefert eine Referenz auf die Singleton-Instanz der `Camera`-Klasse
`scheduler`	Liefert eine Referenz auf die Singleton-Instanz der `ActionScheduler`-Klasse
`[room-scriptid]Light0`, `[room-scriptid]Light1, ...`	Liefert eine Referenz auf das Licht-Objekt mit der jeweiligen Nummer

Abbildung 64: Referenznamen wichtiger Objekte ohne eigene Skript-ID

5 Konstruktion eines Demo-Spiels

Das letzte größere Kapitel dieses Projektberichts erklärt anhand eines kleinen Demo-Spiels, wie man das Framework einsetzten kann, um ein Adventure zu erstellen. Dabei werde ich vor allem auf technische Aspekte eingehen, nicht jedoch auf Fragen des Spieldesigns, wie den Entwurf von Rätseln oder einer guten Rahmenhandlung.

5.1 Die Spielidee in groben Zügen

Das Konzept des Demo-Spiels basiert auf der Grundidee, den Spieler durch unterschiedliche Zeit-Epochen zu führen. Das Chicago des Jahres 1931 stellt dabei die ursprüngliche Gegenwart des Hauptcharakters dar. Der Spieler steuert die Figur des Alfons Catraz. Alfons ist Privat-Detektiv von Beruf und auf der Suche nach einer Spur, die den Verbleib seines Vaters klären könnte. Dieser ist seit mehr als 25 Jahren verschollen. Durch gewisse Umstände trifft Alfons auf eine geheimnisvolle Frau, die behauptet Informationen über seinen Vater zu haben. Es stellt sich heraus, dass die Frau aus der Zukunft stammt und Alfons um Hilfe bitten möchte, seinen im Mittelalter gestrandeten Vater zu retten. Beide gehen auf eine Zeitreise, die sie ins Jahr 2369 verschlägt. In der Zukunft angekommen, wird ein Plan entwickelt, mit dem Alfons' Vater aus dem Mittelalter zurückgeholt werden kann. Für dessen Umsetzung reist das Duo schließlich selbst ins Mittelalter.

Die drei zu besuchenden Epochen werden sich stilistisch deutlich voneinander absetzen. Nicht nur anhand ihrer Architektur werden sie zu unterscheiden sein, sondern auch in der Kolorierung. Die mittelalterliche Vergangenheit wird von einem bräunlichen Sepia-Ton dominiert. Die Gegenwart des Alfons Catraz im Jahre 1931 erhält eine schwarz-weiße Farbgebung. In der Zukunft bekommt der Spieler allerdings die korrekten Farben zu sehen. Erreicht werden kann dieser Effekt durch die Verwendung entsprechend eingefärbter Texturen.

Das Adventure wird klassische Kombinationsrätsel von eher geringer Komplexität enthalten, da der Aufwand zur Planung und Realisierung des Demo-Spiels nicht zu unterschätzen ist. Um die wichtigsten Eigenschaften des Frameworks vorzustellen, sollte es deshalb genügen, eine vergleichsweise simpel gestrickte Spielwelt zu schaffen.

5.2 Exemplarische Umsetzung

Das Kapitel der exemplarischen Umsetzung des Demo-Spiels kann in erster Linie als ausschnittsweise Einführung in den Umgang mit dem Framework verstanden werden. Anhand des Detektiv-Büros, in dem der Spieler zu Beginn startet, werden wichtige Funktionalitäten des Systems praxisnah verdeutlicht.

5.2.1 Das Detektiv-Büro als Beispiel für die Räume eines Spiels

Das Detektiv-Büro ist der Ausgangspunkt des Spiels. Hier erhält der Spieler per Audiobotschaft auf einer Schallplatte den Auftrag, sich zu einem Treffen in die städtische

Bibliothek von Chicago zu begeben. Bevor er allerdings das Büro verlassen kann, müssen zwei Gegenstände gefunden werden, die sich irgendwo im Zimmer befinden. Zum einen die Marke, die den Spieler als Detektiv auszeichnet und zum anderen Schreibmaterial in Form eines Stifts. Die Aufgabe zu Beginn des Spiels ist es also, diese Gegenstände an sich zu bringen. Die beiden folgenden Subkapitel beschäftigen sich mit der Realisierung dieser Spielelemente, um beispielhaft das notwendige Vorgehen zu demonstrieren.

5.2.1.1 Aufbau des Raums

Neben der passiven Kulissen-Geometrie verfügt das Detektiv-Büro über eine Vielzahl interaktiver Hotspots. Manche von ihnen sind belanglos für das Voranschreiten im Spiel (z.B. Lichtschalter, Bilder an der Wand), einigen wenigen kommt jedoch eine tragende Rolle zu (z.B. Stift, Schublade, Grammophon). Außerdem besitzt der Raum mehrere Kamera-Perspektiven, die über Triggerflächen, welche auf dem Boden verteilt sind, umgeschaltet werden. Die folgende Serie von Abbildungen (Abbildung 65, 66, 67) zeigt das Büro samt eingeblendeter Hotspot-Namen aus verschiedenen Blickwinkeln, die direkt dem Spiel entnommen wurden.

Abbildung 65: Screenshot einer Perspektive des Büros mit eingeblendeten Hotspots (1)

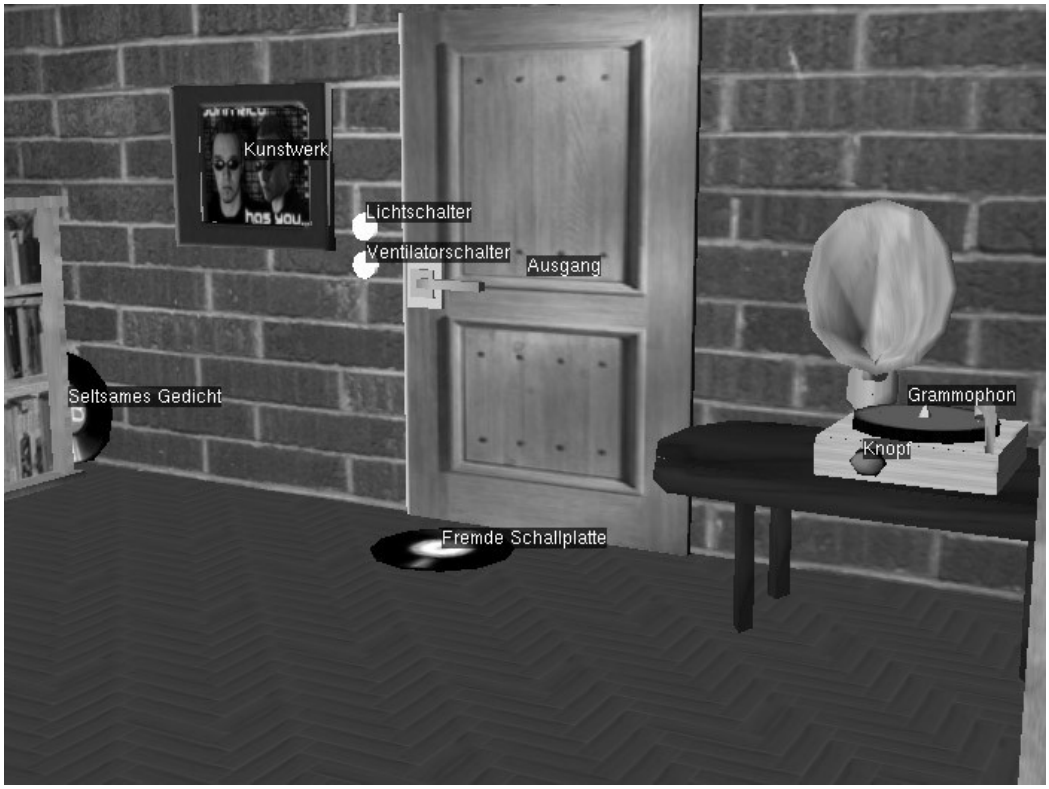


Abbildung 66: Screenshot einer Perspektive des Büros mit eingeblendeten Hotspots (2)

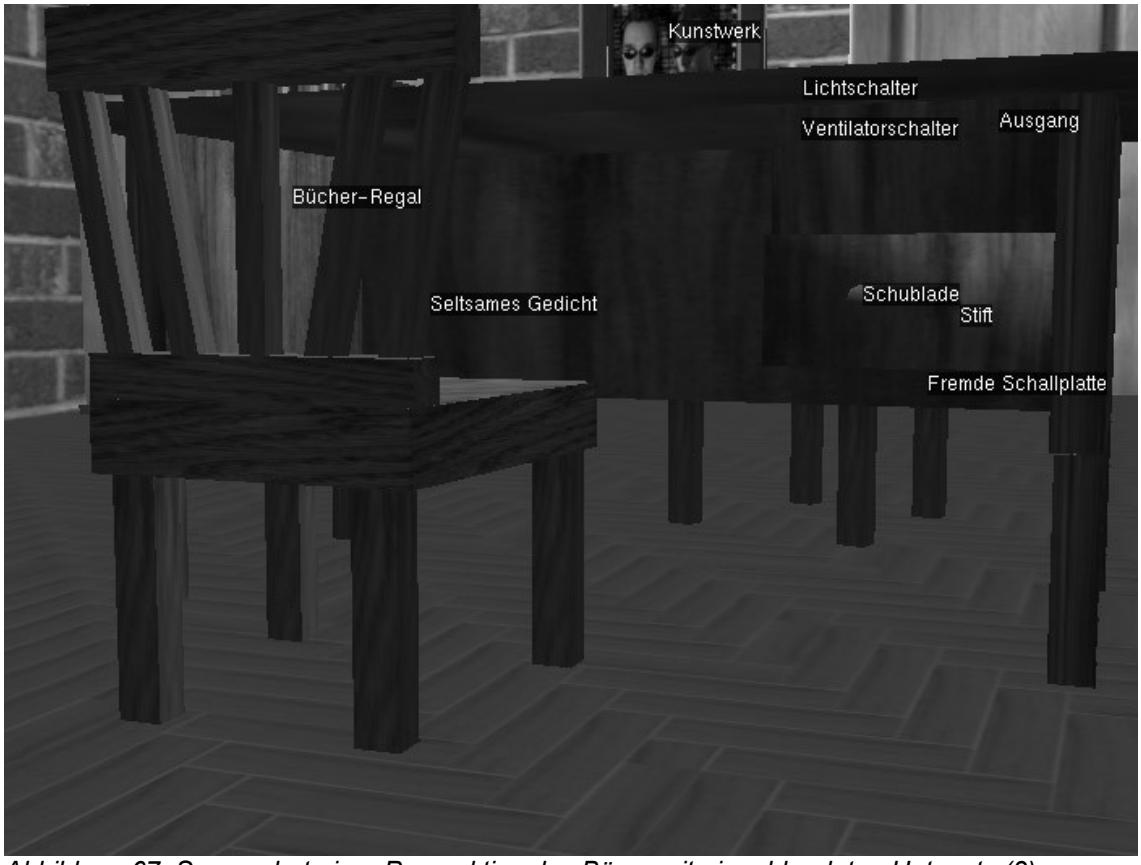


Abbildung 67: Screenshot einer Perspektive des Büros mit eingeblendeten Hotspots (3)

Wie bereits erwähnt, muss der Spieler im Büro sein Detektiv-Abzeichen (Marke) und einen Stift finden, bevor er durch die Tür gehen und sich auf den Weg zum Treffpunkt in der Bibliothek machen kann. Beide Gegenstände sind anhand der eingeblendeten Hotspot-Namen in den obigen Screenshots zu entdecken, wenn sie auch nicht direkt sichtbar sind. Abbildung 65 liefert den Hinweis, dass sich die Marke des Spieler-Charakters auf dem Boden hinter der Zimmerpflanze befindet, während Abbildung 67 offenbart, dass der Stift im Schreibtisch, respektive in dessen Schublade zu finden ist.

5.2.1.2 Implementierung der Spiellogik in der Skript-Datei

In diesem Kapitel möchte ich mehrere interessante Spielelemente des Büros aus Sicht des Skript-Codes näher beleuchten, so dass ein Eindruck davon entsteht, wie sich gewisse Vorgänge mit dem Adventure-System realisieren lassen.

Die Intro-Sequenz

Das Spiel beginnt mit einem kurzen Kameraflug in das Büro des Privat-Detektivs. Anschließend wird die Schallplatte mit der Botschaft an den Spieler in einer Art Filmsequenz unter der Tür hindurch geschoben. Nachdem die Platte ihre Zielposition erreicht hat, bekommt der Spieler ein Klopfgeräusch zu hören, dem der akustische Rückzug des Boten durch das Treppenhaus folgt. Ab hier übernimmt der Spieler die Steuerung des Privat-Detektivs.

Während der Kameraflug durch das Bürofenster aus einer Datei eingelesen wird, die aus zuvor aufgezeichneten Koordinaten besteht, wird die anonyme Überbringung der Audio-Botschaft manuell geskriptet. Abbildung 68 erlaubt einen illustrierenden Blick auf den Ablauf im Spiel.

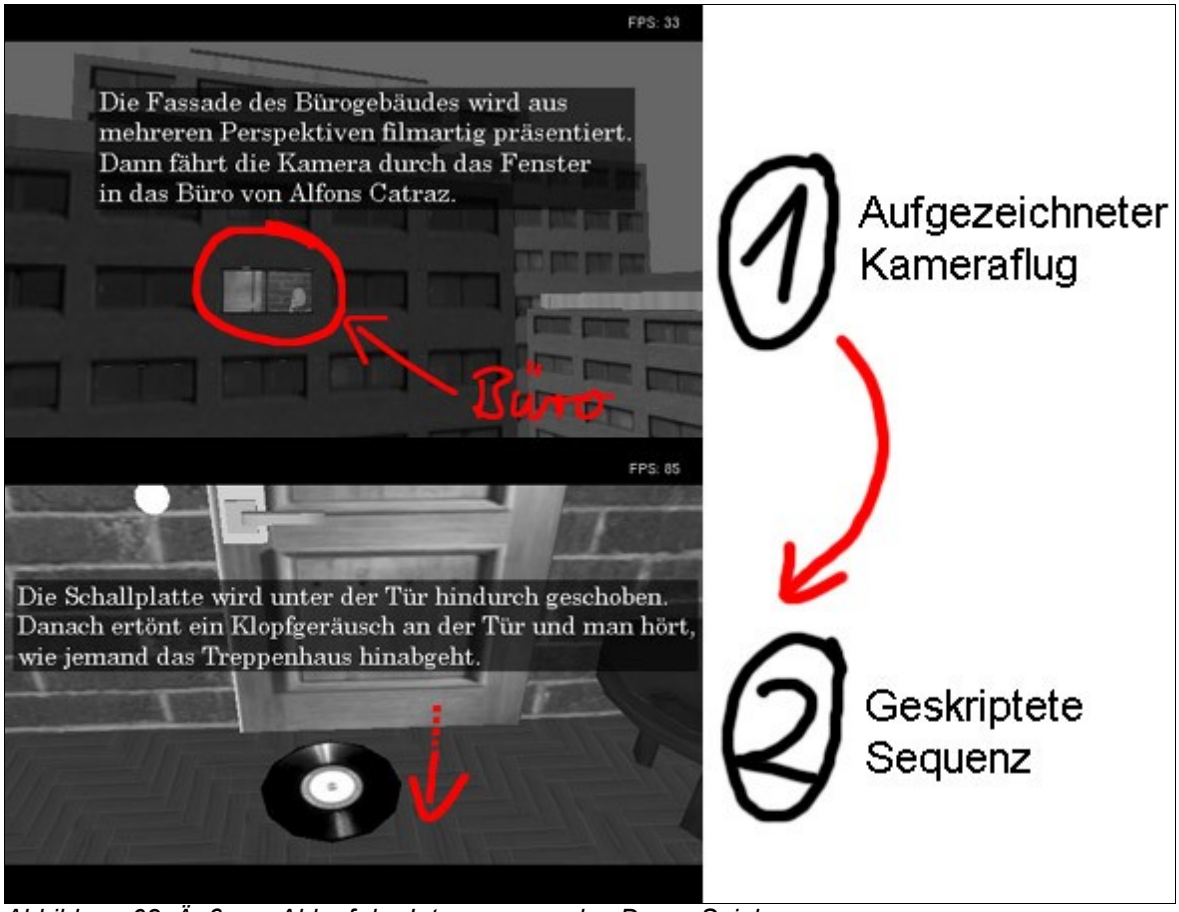


Abbildung 68: Äußerer Ablauf der Introsequenz des Demo-Spiels

Nach der Präsentation des Resultats, möchte ich nun, unter Bezugnahme auf realen Skript-Code, zeigen, wie es konkret umgesetzt wurde. Die folgenden Code-Zeilen (Abbildung 69) enthalten die gesamte Logik, die nötig ist, um den Kameraflug zu starten und die Schallplatte in einer Filmsequenz unter der Tür hindurch zu schieben.

```
// Starten des Kameraflugs unter Angabe der ".rec"-Datei
scheduler.add(new PlayCameraFlight("./locations/chapter1/buro_intro.rec"));

// Filmsequenz: Schallplatte unter der Tür hindurchschieben

// wechsle zur passenden Kamera-Perspektive
scheduler.add(new SetToCamera(6));

// Versetze das Spiel in einen Zustand, in dem schwarze Balken angezeigt
// und Steuerungsbefehle ignoriert werden --> kein manueller Abbruch der Szene!
scheduler.add(new SetToGameState(State.NO_INPUT_CINEMATIC));

// Bewege die Schallplatte für 1900 ms mit dem Faktor 0.001f in Vorwärtsrichtung
scheduler.add(new MoveForward(schallplatte,0.001f,1900));

// Spiele die MP3 mit den Klopfgeräuschen und Schritten ab
scheduler.add(new PlayMP3("./sounds/fx/klopfen_schritte.mp3",false,false));

// Wechsle die Kamera-Perspektive und versetze das Spiel in den Normal-Zustand
// --> Der Spieler übernimmt nun die Kontrolle über die Spielfigur
scheduler.add(new SetToCamera(0));
scheduler.add(new SetToGameState(State.NORMAL));

// Der Spieler-Charakter gibt einen Kommentar ab. Für die Sprachausgabe wurde der
// Pfad zur zugehörigen MP3-Datei angegeben
player.say("Ich glaube, da hat jemand etwas unter meiner Tür durchgeschoben.
             Ich sehe besser mal nach, was das war.",
             "./sounds/speech/alfons/botschaft_nachsehen.mp3");
```

Abbildung 69: Implementierung der Intro-Sequenz mit Skript-Code

Das Verschieben der Zimmerpflanze

Der Spieler muss die Zimmerpflanze verschieben, um das dahinter liegende Detektiv-Abzeichen einstecken zu können. Alfons merkt jedoch erst, nachdem die Pflanze untersucht wurde, dass seine Marke dort auf dem Boden liegt. Vorher sieht er keinen Grund, das Gewächs zu bewegen. Folgender Skript-Code (Abbildung 70) implementiert dieses Verhalten.

```
zimmerpflanzeOnUse() // Code wird ausgeführt, wenn auf die Pflanze geklickt wird
{
  // Das boolean flag "hatPflanzeAngeschaut" gibt an, ob der Spieler zuvor die
  // Zimmerplanze untersucht hat --> nur dann soll der Topf verschoben werden
  if(hatPflanzeAngeschaut == true)
  {
       player.say("Mal sehen, ob sie sich bewegen lässt...","[Pfad zu MP3]");

       // Die Animation mit der ID "give" soll einmal abgespielt werden. In diesem
       // Fall handelt es sich um eine einfache Armbewegung, die die Aktion des
       // Spieler-Charakters symbolisiert
       scheduler.add(new PlayAnimation(player,"give"));

       // Ein schrammendes Geräusch verstärkt den Eindruck des Schiebens. Der
       // "true" Parameter gibt an, dass die MP3 nebenläufig abgespielt werden soll.
       // Auf diese Weise wird sofort mit der nächsten Game-Action fortgefahren.
       scheduler.add(new PlayMP3("./sounds/fx/boden_schrammen.mp3", false, true));

       // Die Zimmerpflanze wird für eine Sekunde mit dem Faktor 0.001f verschoben
       scheduler.add(new MoveRight(pflanze1,0.001f,1000));
       player.say("Na wer sagst denn!","[Pfad zu MP3]");

       // Der Hotspot der Pflanze wird deaktiviert, weil sie nicht weiter
       // verschoben werden können soll. Diese Aktion wurde nicht in den Game-
       // Action-Scheduler gelegt, da sie sofort gültig werden darf und nicht
       // sequenziell abgearbeitet werden muss
       zimmerpflanze.setEnabled(false);
  }
  // Wurde die Pflanze noch nicht untersucht, weiß der Spieler-Charakter noch
  // nichts mit ihr anzufangen
  else player.say("Warum sollte ich die Pflanze verschieben?
                   Ich habe doch keinen Grund dazu.");
}
```

Abbildung 70: Implementierung des bedingten Verschiebens der Zimmerpflanze

Das Ergebnis des Skripts ist eine seitliche Verschiebung der Zimmerpflanze. Abbildung 71 stellt die Situation vor und nach Ausführung des Codes in einer Gegenüberstellung dar.

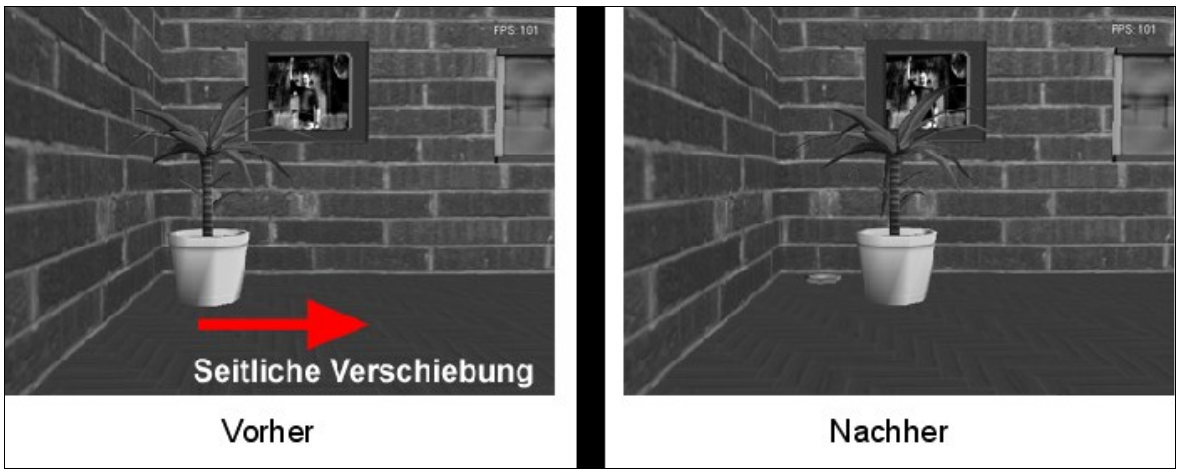


Abbildung 71: Seitliche Verschiebung eines Requisits (Zimmerpflanze) durch Skript-Code

Verlassen des Büros und Aufruf des nächsten Raums

Damit der Spieler sich zum Treffen in der Bibliothek aufmachen kann, müssen ein Stift und ein Detektiv-Abzeichnen gefunden werden. Nur wenn diese Bedingungen erfüllt sind, bringt ein Maus-Klick auf die Ausgangstür des Büros die Spielfigur zum nächsten Raum. Der nächste Raum ist in diesem Fall eine Art dreidimensionale Stadtkarte mit der Skript-ID `map_chicago`. Abbildung 72 demonstriert Skript-Code, der abfragt, ob sich die benötigten Items im Inventar befinden und im positiven Falle den Raumwechsel durchführt.

```
burotuerOnUse() // Code wird ausgeführt, wenn auf die Tür geklickt wird
{
  // nur erwägen zu gehen, wenn die Botschaft auf der Schallplatte angehört wurde
  if(nachrichtGehoert == true)
  {
       // prüfen, ob sich das Detektiv-Abzeichen (Marke) im Inventar befindet
       if(player.getInventory().contains(marke) == false)
       {
              player.say("Ich kann doch nicht ohne mein Detektiv-Abzeichen
                           los gehen. Wo könnte es nur sein?");

              return; // Methode vorzeitig verlassen
       }
       // prüfen, ob sich der Stift im Inventar befindet
       if(player.getInventory().contains(stift) == false)
       {
              player.say("Ich gehe eigentlich nie ohne etwas zum
                           Schreiben aus dem Haus.");

              return; // Methode vorzeitig verlassen
       }

       // den Bildschirm nach Schwarz hin ausblenden
       scheduler.add(new FadeOut(0f,0f,0f,500,250));

       // zum nächsten Raum (Stadtkarte) wechseln
       scheduler.add(new EnterRoom(player,map_chicago,"entryPointBeiBuro",0));
  }
  // Wenn die Botschaft nicht angehört wurde, dann hat Alfons keine Ahnung, wo
  // er denn hingehen könnte
  else player.say("Ich habe keine Ahnung, wo ich hingehen könnte.");
}
```

Abbildung 72: Implementierung eines (bedingten) Raumwechsels mit Skript-Code

Die bisher gezeigten Skript-Codes stellen selbstverständlich nur einen recht überschaubaren Auszug der Möglichkeiten dar, die das Scripting bietet. Als kurze Demonstration soll dies jedoch genügen. Weitere Einblicke sind den vollständigen Skript-Codes des Demo-Spiels oder der JavaDoc-Dokumentation des Projekts zu entnehmen.

6 Fazit und Ausblick

Besonders motivierend für dieses Projekt war die Tatsache, dass es sich thematisch im Bereich der Spieleprogrammierung bewegte. Mit diesem Gebiet kann ich mich rückhaltlos identifizieren, so dass die Bearbeitungphase überwiegend von leidenschaftlichem Engagement geprägt war. Das ganze Projekt war ein kontinuierlicher kreativer Prozess, der mich oft begeisterte und hin und wieder selbst in Erstaunen versetzte, wenn ich sah, wie durch meine Hand eine künstliche Welt entstand.

Da ich die Idee zu dieser Masterarbeit selbst entwickelt habe, verfüge ich über eine besondere Bindung zum Adventure-Framework. Es stellt für mich kein Produkt dar, welches ich nüchtern als Pflichteil einer Prüfungsleistung ansehen könnte. Vielmehr steckt mein ganzes Herzblut in jeder einzelnen der gut 250 Java-Klassen, die sich insgesamt auf etwa 50.000 Zeilen Programmcode erstrecken. Mit meiner Arbeit an dem Framework konnte ich sogar dabei behilflich sein, einen Bug in der JOGL-API zu lokalisieren, der zu einer fehlerhaften Environment-Mapping-Unterstützung führte.

Ich bin sehr zufrieden mit dem Ergebnis und durchaus auch stolz auf das Geleistete. Sieht man einmal von den grafischen Möglichkeiten der 3D-Engine ab, so bildet das entwickelte System einen guten Querschnitt der Funktionen moderner Adventure-Spiele nach. Es ist zum Beispiel möglich, Dialoge zu führen, deren Inhalte sukzessive freigeschaltet werden können und sogar über eine Sprachausgabe verfügen. Die Interaktion mit der Umwelt und dem Inventar hält sich an die Konventionen des Genres und ist durch den kontextsensitiven Cursor recht intuitiv. Ganz wesentlich ist jedoch die Integration des Game Scriptings. Durch den Einsatz der Skript-Sprache BeanShell eröffnet sich ein wahres Füllhorn an Möglichkeiten zur Gestaltung des Spielablaufs. Mit dieser Technik lassen sich auch kleine Zwischensequenzen skripten oder aufgezeichnete Kameraflüge wiedergeben, die etwaigen Spielen einen filmähnlichen Eindruck verleihen und so die Atmosphäre fördern.

Wie oben angedeutet finden sich Verbesserungsmöglichkeiten vor allem im Bereich der grafischen Präsentation. Ein wichtiger Schritt hin zu moderner Qualitätsgrafik bestünde im Einsatz von Pixel- und Vertex-Shadern. Die 3D-Engine müsste um weitere Strukturen ergänzt werden, so dass entsprechende Shader-Programme zur optischen Aufwertung der Szene nutzbar sind. Weiterer Handlungsbedarf gilt sicherlich den performance-steigernden Maßnahmen. Zwar tragen Frustum- und Occlusion-Culling bereits erfolgreich zur Leistungsoptimierung bei, doch bleiben diese Techniken bisher hinter ihren Möglichkeiten zurück. Algorithmen und Datenstrukturen zur Partitionierung der Szene, wie zum Beispiel Octrees und Szene-Graphen, könnten helfen, die Anzahl der notwendigen Culling-Tests zu minimieren und den Render-Prozess zu optimieren. Im Bereich des Spiele-Frameworks selbst ist bisher insbesondere eine fehlende Speicherfunktion zu beklagen. Dies ist ein Punkt, der in Zukunft unbedingt ergänzt werden sollte, damit sich umfangreiche Adventure-Projekte angenehm spielen lassen.

Wenn es die Zeit nach dem Eintritt in das Berufsleben erlaubt, weitere Arbeit in dieses Projekt zu investieren, bin ich zuversichtlich, dass sich das Framework auch weiterhin prächtig entwickeln und noch mehr Möglichkeiten zur Umsetzung eigener Spielideen bieten wird.

7 Literaturverzeichnis

[Astle2004] Astle, Dave; Hawkins, Kevin: *Beginning OpenGL Game Programming*, Premier Press, 2004

[Bourg2004] Bourg, David M.; Seemann, Glenn: *AI for Game Developers*, O'Reilly, 2004

[Brackeen2004] Brackeen, David; Barker, Bret, Vanhelsuwé. Laurence: *Developing Games in Java*, New Riders Publishing, 2004

[Davis2004] Davis, Gene: *Learning Java Bindings for OpenGL (JOGL)*, AuthorHouse, 2004

[Davison2005] Davison, Andrew: *Killer Game Programming in Java*, O'Reilly, 2005

[DeLoura2002] DeLoura, Marc (Hrsg.): *Spieleprogrammierung Gems 1*, mitp-Verlag, 2002

[Dunn2002] Dunn, Fletcher; Parberry, Ian: 3*D Math Primer for Graphics and Game Development*, Wordware Publishing, 2002

[Freeman2004] Freeman, Eric; Freeman, Elisabeth, Sierra, Kathy; Bates, Bert: *Head First Design Patterns*, O'Reilly, 2004

[Krüger2002] Krüger, Guido: *Handbuch der Java-Programmierung*, Addison-Wesley, 2002

[Lengyel2004] Lengyel, Eric: *Mathematics for 3D Game Programming and Computer Graphics*, Charles River Media, 2004

[Seeboerger2002] Seeboerger-Weichselbaum, Michael: *Java/XML Das bhv Taschenbuch*, vmi Buch, 2002

[Seeboerger2004] Seeboerger-Weichselbaum, Michael: *Das Einsteigerseminar XML*, vmi Buch, 2004

[Stahler2004] Stahler, Wendy: *Beginning Math and Physics for Game Programmers*, New Riders Publishing, 2004

[Watt2001] Watt, Alan; Policarpo, Fabio: *3D Games: Real-time Rendering and Software Technology*, Pearson Education Limited

[Wright2004] Wright, Richard S.; Lipchak Benjamin: *OpenGL Superbible*, SAMS, 2004

8 Abbildungsverzeichnis

Alle Abbildungen im Überblick

9 Stichwortverzeichnis

10 Erklärung der selbstständigen Anfertigung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.
Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

_____ _____

Ort, Datum Unterschrift: Frank Bruns

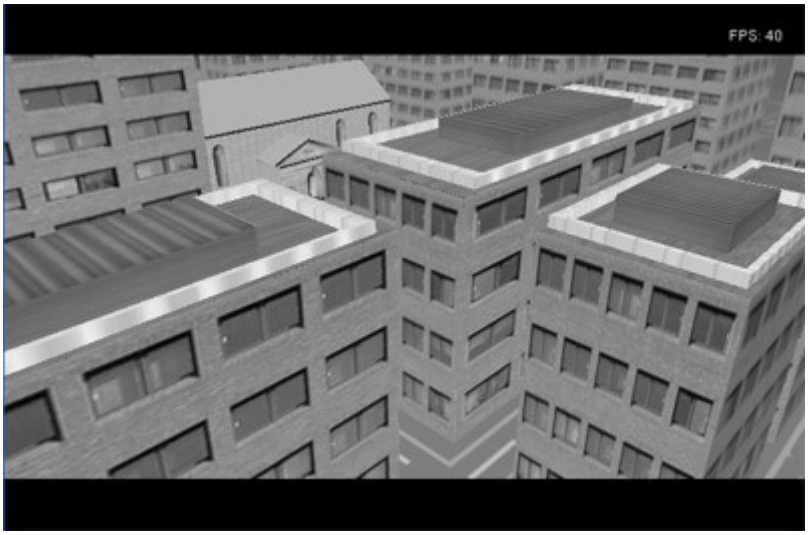
11 Anhang

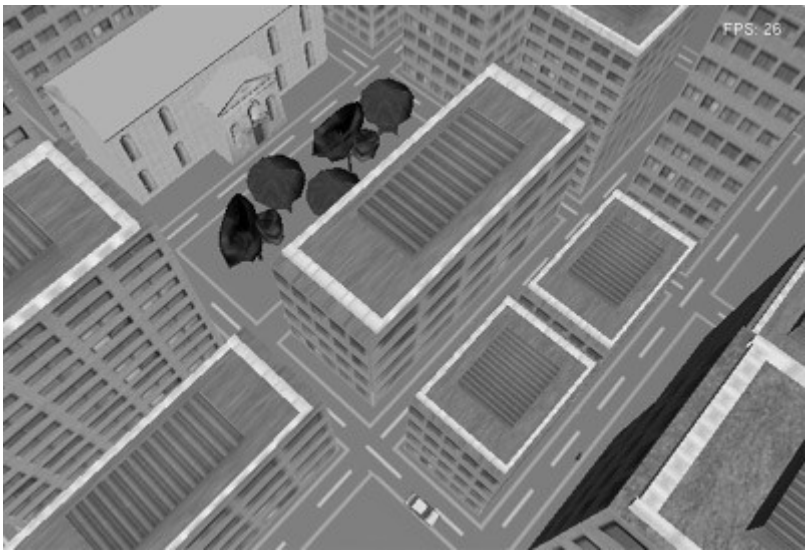
11.1 Ausgewählte Screenshots des Demo-Spiels

Das Demo-Spiel besteht insgesamt aus fünf verschiedenen Räumen. Die ersten vier sind im Chicago des Jahres 1931 situiert. Der letzte Raum spielt jedoch in der Zukunft des Jahres 2369. Die folgenden Abschnitte zeigen ausgewählte Screenshots eines jeden Raums, so dass auch der stilistische Bruch von der Graustufenästhetik der Vergangenheit zur Farbenpracht der Zukunft vermittelt werden kann.

11.1.1 Screenshots des Detektiv-Büros

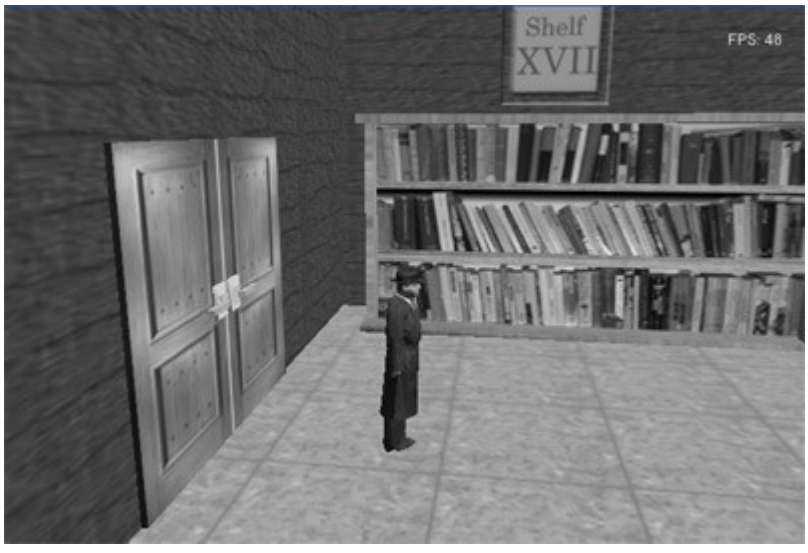
11.1.2 Screenshots der Stadtkarte von Chicago

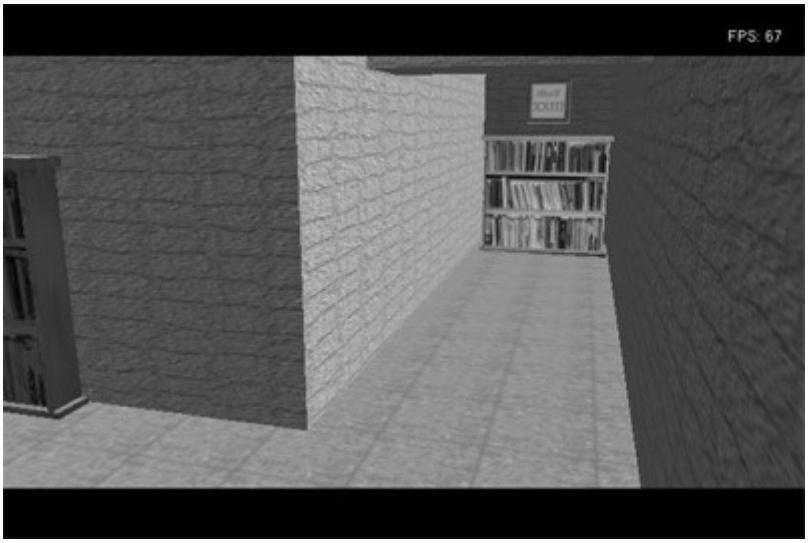




11.1.3 Screenshot des Bibliotheksfoyers

11.1.4 Screenshots des Lesesaals der Bibliothek




FPS: 67

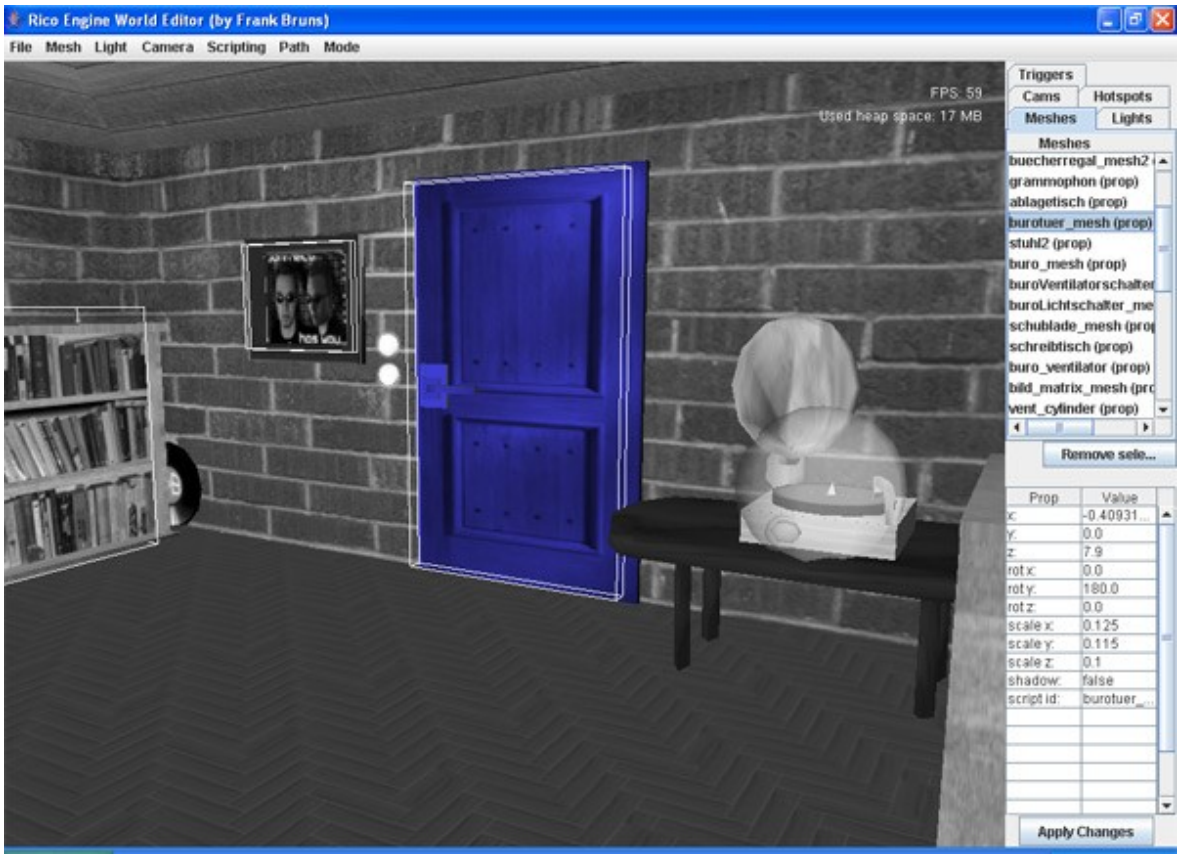

FPS: 40

11.1.5 Screenshots des Labors in der Zukunft

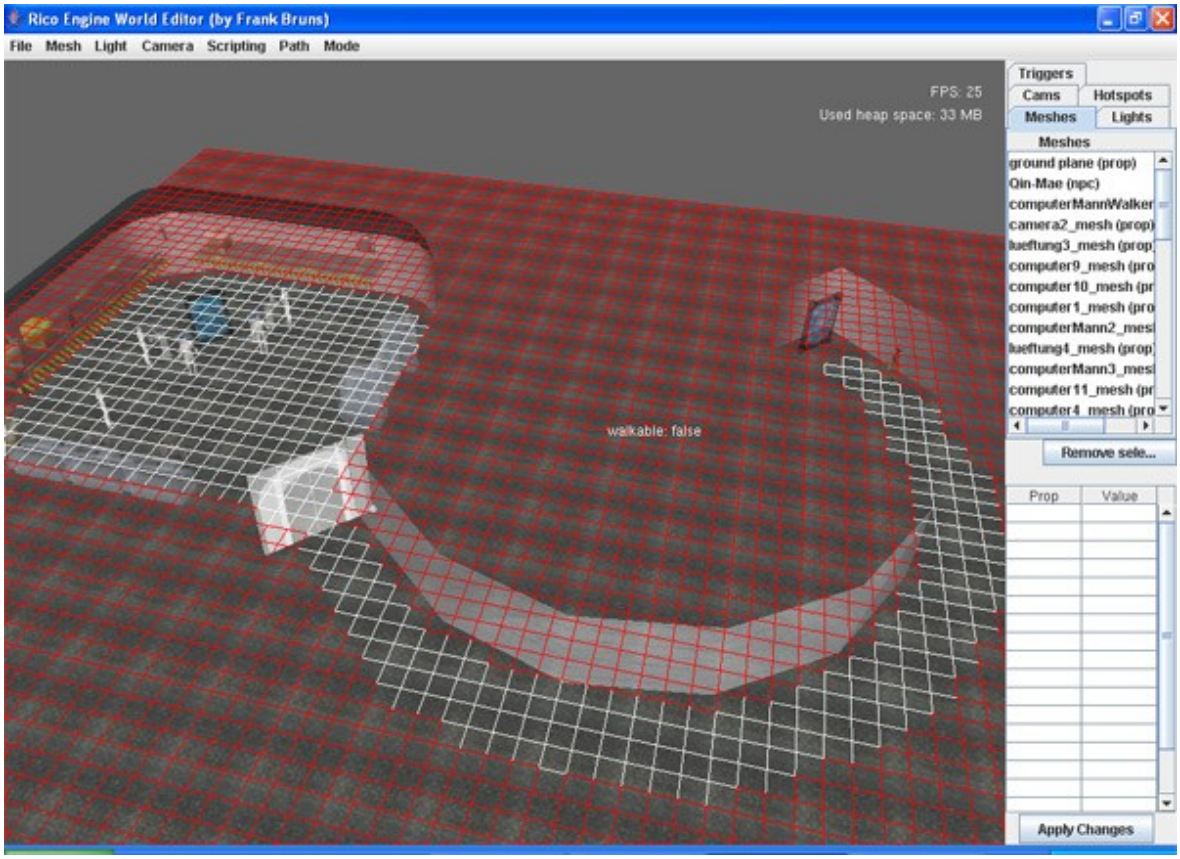
11.2 Ausgewählte Screenshots des 3D-World-Editors

Im Zuge des Projekts ist als inoffizielles Nebenprodukt ein 3D-Editor entstanden, mit dem sich die Räume eines Adventures relativ komfortabel zusammenstellen und als „.room“-Datei abspeichern lassen. Obwohl er vergleichsweise prototypisch und in manchen Belangen nur rudimentär ausgearbeitet wurde, leistete er bereits unschätzbare Dienste in der Umsetzung des Demo-Spiels. In den folgenden Abschnitten werden einige Screenshots präsentiert, die einen Eindruck des Editors vermitteln sollen.

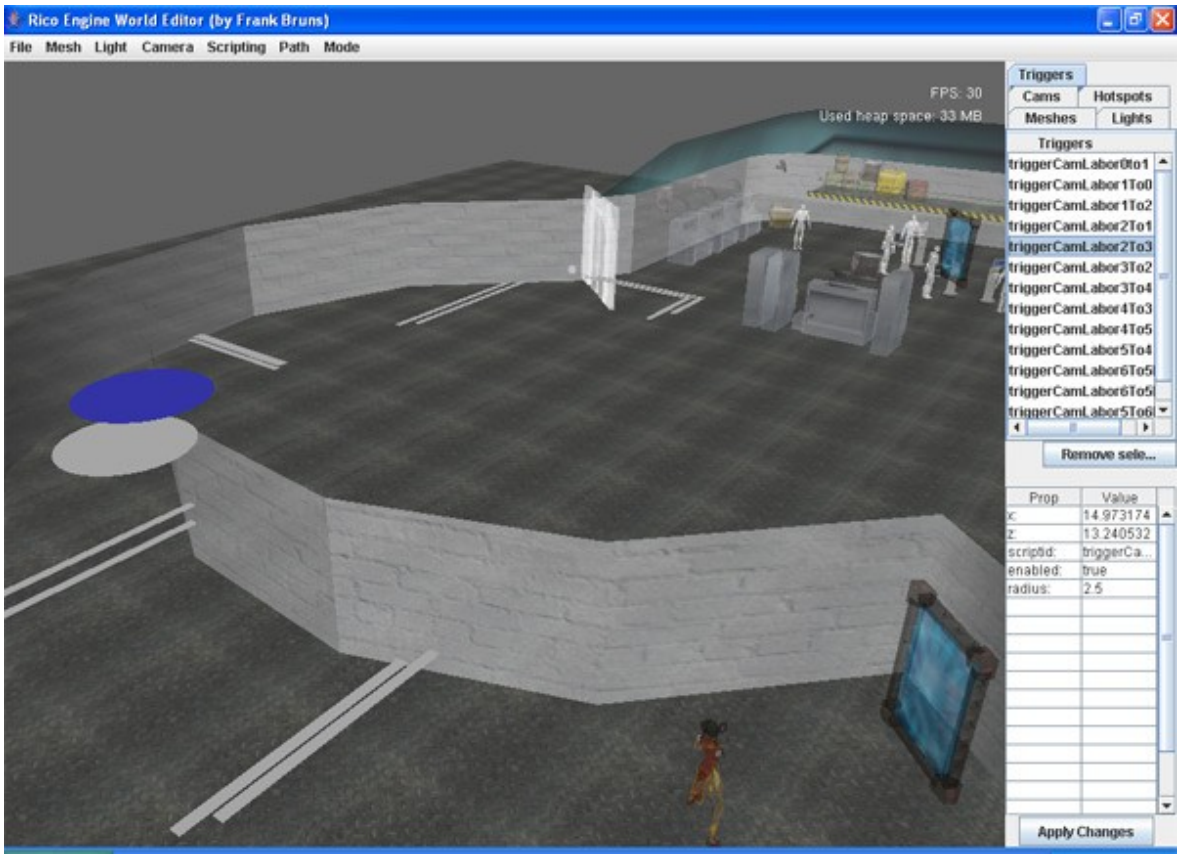
11.2.1 Screenshot des Konstruktions-Modus



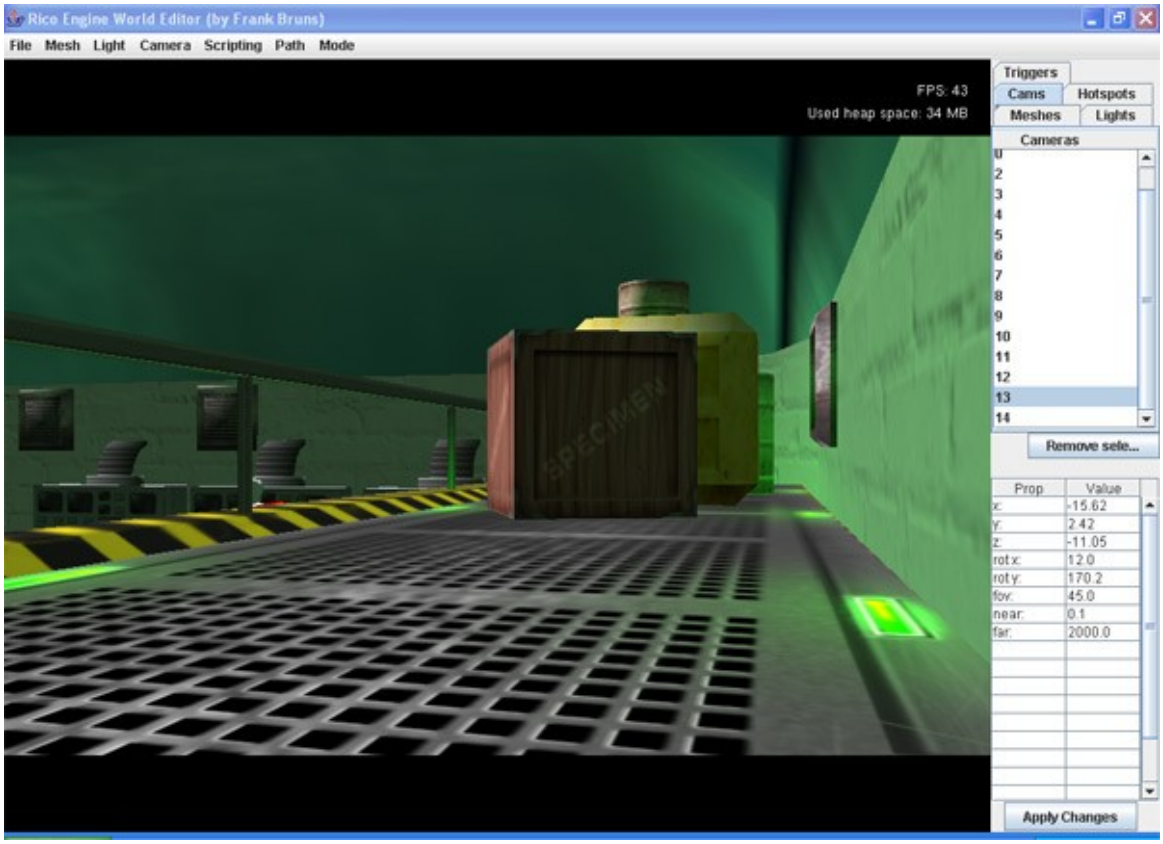
11.2.2 Screenshot des Path-Grid-Modus



11.2.3 Screenshot des Trigger-Modus



11.2.4 Screenshot des Kameraflug-Recoder-Modus



11.2.5 Screenshot des Pfad-Modus