HapticBillard

Physikalisch basierte 3D-Billardsimulation mit haptischer Interaktion

Albert-Ludwigs-Universität Freiburg
Lehrstuhl für Computergrafik

April 2009

Julian Bader, Benjamin Ummenhofer,
Johannes Wendeberg

Ausarbeitung zum
TeamProjekt im Masterstudiengang Informatik, WS 2008/09
Prof. Dr. Matthias Teschner

Zusammenfassung:

Billardsimulationen sind eine weit verbreitete Art von Computerspielen. Sie alle haben gemeinsam, dass eine durchschnittliche bis gute Physiksimulation mit einer Eingabesteuerung durch Tastatur und Maus kombiniert wird. Der erzielbare Realismus ist teilweise schon durch die Physik begrenzt. Spätestens aber durch die Bedienung des Queue mit der Maus wird der Immersion in das Spiel eine Schranke gesetzt. Das Gefühl, einen massigen Queue in der Hand zu halten, fehlt.

Man könnte ein realistischeres Eintauchen in das Computerspiel ermöglichen, indem man eine wirklichkeitsgetreue Physiksimulation von Billard mit einem 3D-Eingabegerät wie dem PHANTOM Omni® Haptic Device kombiniert. Diese Geräte erlauben die präzise Steuerung eines 3D-Cursors und vermitteln eine Kraftrückmeldung bei Kollisionen. Ihr stiftförmiger Manipulator ist geradezu prädestiniert dafür, sie als Billard-Queue zu verwenden.

Bisherige Arbeiten haben das nur im Ansatz untersucht. Wir wollen ihre Idee fortführen und entwickeln eine universelle Physiksimulation, eine Schnittstelle für das Haptic Device, die den Stift als Queue verwendet, eine grafische Visualisierung von Billard und kombinieren die drei Komponenten in ein lauffähiges Computerspiel. Darüber hinaus implementieren wir einige Komponenten, die das Spiel abrunden, etwa ein Soundsystem, die Spielregeln von 8-Ball und einen Computergegner.

Danksagung

Wir danken unseren Betreuern Marc Gissler und Rüdiger Schmedding für ihre hilfreiche Unterstützung bei der Realisierung unseres Projekts. Sie konnten uns bei unseren Tausenden von Fragen und Problemen stets mit guten Ratschlägen unter die Arme greifen. Desweiteren danken wir unseren Freunden für ihre Geduld, wenn wir ihnen unsere schier endlosen Ideen vortrugen.

Inhaltsverzeichnis

1 Einleitung

Der Vorschlag für den Entwurf einer Billardsimulation im Rahmen des Teamprojekts für den Masterstudiengang Informatik kam im Herbst 2008 von den Mitarbeitern des Lehrstuhls für Computergrafik an der Universität Freiburg. Was sich zunächst nach einer Wiederholung von bereits da gewesenem anhörte, stellte sich bei näherer Erläuterung als hochinteressantes und herausforderndes Projekt für alle Beteiligten dar. Wir beschlossen, es zu versuchen und sagten dem Projekt zu.

Fast allen bisherigen Billardspielen ist gemein, dass sie eine mehr oder weniger gelungene Spielphysik auf ein Billardspiel maßschneidern und dazu eine Steuerung für Tastatur und Maus entwerfen. Der Nachteil dieser Eingabemethode ist jedoch das mangelhafte Gefühl für einen Queue, den man in der Hand hält.

Desweiteren existieren für einen 3D-Eingabestift mit Force-Feedback-Funktion wie den *PHANTOM Omni® Haptic Device* von *SensAble technologies* bereits eine Vielzahl von Anwendungen im grafischen und medizinischen Bereich. Eine Kombination des Stiftes mit einem Computerspiel wurde von De Paolis et al. [1] kürzlich untersucht, jedoch wurde der Gedanke des Billardspiels nicht zuende geführt. Sie erschufen eine einfache Simulation von physikalischen Kugeln, die durch das PHANTOM Omni manipuliert werden können. Der Stift des Haptic Device übernimmt dabei die Aufgabe eines Queue mit Position und Orientierung. Kräfte werden per Force Feedback rückgemeldet. Die Verwendbarkeit ist wegen der Einfachheit der Simulation jedoch eingeschränkt.

Wir wollten den Gedanken weiterführen. *Lässt sich das Haptic Device so in ein Billardspiel integrieren, dass man das Gefühl bekommt, wirklich Billard zu spielen?* Drei Dinge sind es, die wir hierfür benötigten. Zunächst eine realitätsnahe Physiksimulation, die den Eindruck vermittelt, es würden sich echte, massige Kugeln nach den newtonschen Gesetzen bewegen. Sie sollte so universal gehalten werden, dass man damit alles machen kann, was mit Kugeln zu tun hat, unter anderem auch Billard spielen. Als zweites eine Eingabesteuerung für das PHANTOM Omni, die das Gefühl vermittelt, man halte einen Queue mit Masse und Trägheit in der Hand. Der Kontakt mit den Kugeln sollte eine Rückmeldung erzeugen und den Queue so abprallen lassen, wie man es in der Realität erwarten würde. Die dritte Komponente war eine gefällige Grafik, die den (hoffentlich) gelungenen Eindruck von Physik und Steuerung trägt. Wir wählten zur Visualisierung das *OGRE SDK* [2], eine objektorientierte, flexible Grafik-Engine für C++ zur Darstellung hardwarebeschleunigter 3D-Anwendungen. Sie besitzt Libraries für Direct3D und OpenGL und unterstützt sowohl Windows als auch Linux.

Abbildung 1.1: Das Spiel *BillardGL*. Links ist der Titelscreen, rechts eine beliebige Spielsituation. Auffällig sind die weichen Schatten und die stimmige Farbkombination der Beleuchtung und Texturen. [3]

Wir stellten fest, dass wir die drei Komponenten leicht unter uns aufteilen konnten. Johannes Wendeberg wollte die physikalische Simulation entwerfen, Benjamin Ummenhofer sagte die Queuesteuerung durch das Haptic Device zu und Julian Bader konnte sich für die grafische Gestaltung begeistern. Dieser Bericht fasst die Ergebnisse unserer sechsmonatigen Entwicklung an diesem Projekt zusammen.

1.1 Verwandte Arbeiten

Als wir Freunden erzählten, wir entwerfen eine Billardsimulation, war die spontane Antwort: „Noch eine?“. Tatsächlich existiert mittlerweile eine Vielzahl von Billardspielen, die teils kostenlos sind, teils kommerziell vertrieben werden. Die interessantesten stellen wir im folgenden vor. Mit dem Überblick von auf dem Markt vorhandenen Billardsimulationen fällt auch die Abgrenzung und Einordnung unseres eigenen Entwurfs leichter.

1.1.1 BillardGL

Gleich vorweg nimmt *BillardGL* [3] von Stefan Disch, Tobias Nopper und Martina Welte eine Sonderstellung ein. Das Projekt ist 2001 im Rahmen einer Vorlesung zu Computergrafik an der Universität Freiburg entstanden und wurde bis 2004 zu einer ansehnlichen Simulation weiterentwickelt. Damit setzen die drei Entwickler eine Referenz, die uns als Nachfolge-Projekt ihrer Implementation einordnet, und sie haben diese Messlatte sicherlich nicht zu niedrig gesetzt, wie sich zeigen wird.

Startet man das Spiel, beeindruckt ein animiertes Hauptmenü mit den Spielmodi „Training“, „8-Ball“ und „9-Ball“, einer Variante in der zunächst die niedrigeren Kugeln versenkt werden müssen. Im Spiel fällt die stimmige Farbkombination des Tisches auf, die zu den weichen Schatten passt, und dem leichten, durch den Tisch verursachten Grünschimmer der Kugeln (vgl. auch Abb. 1.1). Eine gewöhnungsbedürftige, aber sinnvolle Steuerung für Maus und Tastatur führt durch den Spielablauf. Der Zug eines Spielers ist gegliedert in die Modi „Betrachten“, in dem die Kamera frei schwenkbar ist, „Zielen“, bei dem man mit der weißen Kugel eine farbige Kugel anvisiert und „Stoß“ wo man einen Power-Balken auflädt und den Schuss auslöst. Ein Queue existiert nicht, die Position der Kamera bestimmt die Laufrichtung der weißen Kugel. Die Kamera kann federbasierte Schwenks vollführen, was bedeutet, dass sie stark beschleunigt und dann immer langsamer wird.

Die zweidimensionale Physik ist einfach gehalten. Die Positionen aller Kugeln werden, abhängig von einem Ausgangszustand, zu Beginn des Stoßes berechnet und zu den entsprechenden Zeitpunkten angezeigt. Eine unerwartete Interaktion nach Stoßbeginn, z.B. ein ungültiges Eingreifen des Spielers, ist damit nicht möglich. Zwischen den Kugeln findet Impulskollision statt, an den Banden werden die Einfallsvektoren der Kugeln gespiegelt. Nicht ganz klar wurde, ob die Kugeln an der Bande mit dem Mesh oder mit parametrischen Banden kollidieren. Ansonsten findet keine reguläre, trianglebasierte Meshkollision statt. Die Kugeln besitzen keine physikalische Rotationsträgheit, sondern sie werden immer in die Richtung rotiert, zu der sich eine Kugel hinbewegt. Der Tisch wurde als Drahtgitter-Modell erstellt, dabei wurden die Koordinaten der Vertices manuell eingegeben. Er besteht aus mehreren Komponenten. Die Fläche, die Löcher, die Innen- und die Holzbanden wurden separat modelliert und dann zusammengesetzt.

Das Ergebnis ist eine sehr stimmige Kombination der Einzelteile, wozu die angenehme Optik beiträgt. Mit begrenzten Mitteln wurde viel erreicht, so sehen die transformierten Schattentexturen fast wie echte, weiche Schatten aus. Auch die Idee mit der diffusen, grünen Lichtquelle von unten zur Nachahmung von Radiosity ist gut. Leider kommt richtiges Billard-Feeling aufgrund der sehr einfachen Physik nicht auf. Die Kugeln scheinen zu schweben, das Gefühl massiger Kugeln fehlt. Springende Kugeln können nicht simuliert werden. Schlittern und Effet (Anschneiden) ist nicht möglich, auch der Verzicht auf einen Queue ist gewöhnungsbedürftig. Sound wurde angekündigt aber noch nicht implementiert.

In jedem Fall macht *BillardGL* einen sehr runden, stimmigen und vollständigen Gesamteindruck. Aus dem begrenzten Framework wurde mit Sicherheit das Maximum herausgeholt. Das ist auch der gründlichen Nacharbeit zu verdanken, in mehreren Versionen wurde das Projekt immer wieder fehlerbereinigt und überarbeitet. Zwar wurde der Fokus stärker auf die Optik gelegt als auf grundsätzliche Fähigkeiten und Features, das aber mit Erfolg.

1.1.2 FooBillard

Das Computerspiel *FooBillard* [4] ist eine Entwicklung von Florian Berger und wurde zwischen 2002 und 2004 entwickelt. Wie BillardGL ist es Open Source und somit frei erhältlich. Es besitzt eine Vielzahl von Features, unter anderem einen Computer-Gegner und einen Netzwerkmodus zum Spiel an mehreren Rechnern. Mit FooBillard sind exzentrische Stöße unter Verwendung des grafischen Queue möglich. Der Queue scheint aber nur ein nebensächliches Feature zu sein, die Standardmethode um einen Schuss auszulösen ist auch hier das Aufladen eines Power-Balkens mit anschließendem Schuss durch eine Animation des Queue. Inwiefern das nach physikalischen Regeln geschieht, ließ sich nicht klären. Springende Kugeln sind zumindest nicht möglich, sie bewegen sich ansonsten glaubwürdig. Grafisch hat *FooBillard* einiges zu bieten. Betrachtet man das Tischfilz aus näherer Entfernung, wird sanft eine Detailtextur eingeblendet. Die Reflexionen der Kugeln sind sehr aufwändig gestaltet. Es spiegelt sich nicht nur der Tisch sondern auch die anderen Kugeln wieder. Interessant ist auch die Möglichkeit stereoskopische Halbbilder rendern zu lassen, die es erlauben mit einer Rot-Grün-Brille räumlich zu sehen.

1.2 Projekt HapticBillard

Die Billardsimulation *HapticBillard* ist ein Eigenentwurf im Rahmen des Masterstudiengangs Informatik und wurde von Julian Bader, Benjamin Ummenhofer und Johannes Wendeberg während des Wintersemesters 2008/09 entwickelt. Die zugrundeliegende Idee war der Entwurf einer Software, in der ein Billardspiel physikalisch korrekt simuliert wird, während die Steuerung und Kraftrückmeldung durch das 3D-Eingabegerät *PHANTOM Omni®Haptic Device* erfolgt. Natürlich sollte auch die Visualisierung ansprechend gestaltet sein. Ein Screenshot ist in Abb. 1.2 zu sehen.

Aufgrund der Bewertungsrichtlinien der Hochschule ist es notwendig, die Einzelleistungen der beitragenden Entwickler differenziert herauszustellen. Auch deshalb wurde schon zu Beginn großer Wert auf modulare Entwicklung der einzelnen Bestandteile gelegt. Auf der anderen Seite hat jeder Teilnehmer großes Interesse an den Entwicklungen der anderen, damit die Schnittstellen zwischen den Komponenten festgelegt werden können. So ergibt es sich, dass zwar jeder Entwickler im Groben die komplette Programmstruktur erklären könnte, im Detail weiß aber nur der jeweilige Experte über seinen Bereich genau Bescheid. Aufgrund des Umfangs der Software ist das ein sinnvoller Kompromiss.

Ein Teil der Funktionen ist so trivial, dass es nicht notwendig ist, sie zu erwähnen. In dem Fall führen wir auch den Autor nicht auf. Für die Hauptkomponenten ist der jeweilige Autor jedoch festgelegt, alle erwähnten Nebenkomponenten sind mit dem

Abbildung 1.2: Beliebige Spielsituation in *HapticBillard* mit dem Tisch im Vordergrund und Accessoirs des Zimmers im Hintergrund. Der Spieler versucht, die rote Kugel einzulochen.

Namens-Kürzel des Entwicklers (nicht des Autors vom Text!) am Ende des Abschnitts versehen.

- Die *Physik* und die *Kollision* stammen von **Johannes Wendeberg (J.W.)** (Teamprojekt).
- Die Interaktion mit dem *HapticDevice* und das *Inputsystem (für Haptic Device, Tastatur + Maus)* wurde durch **Benjamin Ummenhofer (B.U.)** (Teamprojekt) entwickelt.
- Die *grafische Visualisierung (Meshes, Materials etc.)* erfolgte durch verschiedene Entwickler, wobei **Julian Bader (J.B.)** sich im Rahmen eines Praktikums damit befasste. Der entsprechende Entwickler ist durch ein Namenskürzel hervorgehoben.
- *Peripherie: Eventsystem, Spielregeln, Soundsystem, AI Player.* Auch für die Vielzahl von Nebenkomponenten wird der jeweilige Entwickler durch ein Namenskürzel am Ende des Abschnitts angegeben.

Der Beitrag von Julian Bader ist von den anderen beiden abzugrenzen, da das Praktikum weniger Kreditpunkte als das Teamprojekt erbringt. Dementsprechend ist auch sein Beitrag geringer, wenngleich nicht unerheblich, ausgefallen.

2 Physik und Kollision

Die naturgetreue Simulation der physikalischen Gesetze, die auf alle beweglichen Objekte wirken, stellt das Rückgrat der Software *HapticBillard* dar. **Johannes Wendeberg** entschied sich, diese Komponente zu bearbeiten. Ein wichtiges Design-Kriterium war die universelle Verwendbarkeit für „Ballspiele“ aller Art. Zwar wurde die Software als Billardspiel implementiert, doch die Idee war, dass der Billardtisch einfach gegen eine Hindernisbahn ausgetauscht wird, die Billardkugeln gegen einen kleinen Plastikball und der Queue gegen einen Schläger, schon erhält man ein Minigolfspiel. Zwar schränkt die konkrete Implementation der Spielregeln und spielspezifischen Ereignisse diese Flexibilität ein, doch prinzipiell ist das möglich durch den Entwurf zweier grundlegender Objektklassen, dem *Ball* und dem *Face* als Bestandteil eines *Meshes*:

- **Ball:** Ein parametrisches, *Ball*-Objekt wird zur Darstellung sphärischer, physikalischer Objekte verwendet. Es besitzt die üblichen physikalischen Eigenschaften wie Größe, Gewicht, Trägheitstensor und den physikalischen Zustandsvektor für Position, (Winkel-) Geschwindigkeit, Kraft und Orientierung. Darüber hinaus lassen sich über eine Menge von Flags Eigenschaften wie Schwerkraft, Integration oder Kollision aktivieren bzw. deaktivieren. Am Ball ist eine Feder angebracht, die ihn sowohl linear als auch angular in eine bestimmte Position bzw. Orientierung zerrt (wenn aktiviert). Auf diese Weise kann er nicht nur als Implementation einer Billardkugel oder eines Golfballes dienen, sondern auch als Aufhängung für die Kamera, die dadurch die Fähigkeit zur Kollision mit Umgebungsobjekten oder Effekte wie ein wackelndes, nachschwingendes Bild erhält. Auch Kamerafahrten lassen sich so implementieren[1]. Desweiteren kommt der Ball als physikalisch aktive Spitze des Billard-Queues zum Einsatz.
- **Face:** Die Bezeichnung „Dreieck“ trifft die Funktion dieses Objekts nicht ganz. Der englische Begriff *Face* hat sich bei uns eingebürgert und die Objektklasse trägt diesen Namen. Deswegen sprechen wir von *Face*, wenn wir die elementaren Bestandteile eines *Mesh*-Objekts bezeichnen: Das *Mesh* ist ein polygones Objekt beliebiger Form, aufgebaut aus einzelnen, miteinander verbundenen Faces. Das *Face* besteht aus drei Punkten im Raum, welche seine Position und Form festlegen. Es ist bestimmt durch die Fläche, die die Kanten zwischen den Punkten einschließen.

[1] Wobei wir tatsächlich eine elegantere Methode mit „Splines“ verwenden

Gerät nun ein Ball derart in die Nähe eines Faces, dass eine Schnittfläche zwischen Ball und Facefläche entsteht, so findet eine Ball-Mesh-Kollision statt. Die Normale dieser Kollision wird mit der Funktion `iNormal()` zu jeder Zeit richtig berechnet, egal auf welche Art der Zusammenstoß geschieht. Ein Ball kann außerdem mit einem anderen Ball kollidieren. Diese Kollision kommt zwischen Billardkugeln zum Einsatz und beim Queuestoß, dessen Spitze ein Ball ist.

Die Kollision zweier Meshes wird nicht unterstützt. Der Grund ist, dass wir bewegliche Mesh-Objekte in jedem Simulationsschritt neu in die Hashtabelle des *Spatial Hashing* eintragen müssten (vgl. Abschnitt 2.1), anstatt das nur einmalig zu Beginn der Simulation zu tun. Was mit den kleinen Bällen kein Problem ist, hat sich bei den vielen Faces eines jeden Meshs als äußerst zeitintensiv erwiesen. Deshalb haben wir auf die Kollision zwischen Meshes verzichtet.

2.1 Kollisionserkennung

In einer Spielwelt mit n Bällen muss in jedem Simulationsschritt jeder Ball mit jedem anderen Ball auf Kollision getestet werden, wobei Paare nicht doppelt untersucht werden. Die Zahl der zu testenden Zusammenstöße ist quadratisch mit $N = \frac{n(n-1)}{2}$. 8-Ball besteht aus 16 Spielkugeln, es ergeben sich 120 Kollisionstests. Bei der Ball-Mesh-Kollision ist die Zahl der Tests noch viel größer: Der Billardtisch besteht aus etwa 10.000 Faces, die Zahl der Kollisionstests ist bei 16 Bällen also 160.000, nur für den Tisch. Das lässt sich mit einer Schleife über alle Objekte („brute force“) nicht mehr in vernünftiger Zeit berechnen. Glücklicherweise ist das auch nicht notwendig.

Die Kollisionserkennung hat zur Aufgabe, die Zahl der zu überprüfenden Kollisionen zu reduzieren. Dabei sollen schon im Vorfeld Kollisionen ausgeschlossen werden, die nicht stattfinden können. Die Idee des räumlichen Zusammenhangs wird angewandt. So wird etwa ein Ball im Spielfeld niemals mit den weit entfernten Tischbeinen kollidieren, genausowenig mit einem ein Meter entfernten Ball.

Zur Verwendung kommt ein Verfahren namens *Spatial Partitioning*. Dieses teilt den Raum in Zellen bestimmter Größe auf. Eine Hashfunktion versieht alle Zellen mit einer Identifikationsnummer und trägt alle Objekte, die sich in der Zelle befinden, in die Liste eines zugehörigen Tabellenplatzes ein. Dabei wird eine nicht injektive Abbildung von Zellen auf Tabelleneinträge verwendet, so dass zu jedem Tabelleneintrag mehrere Zellen gehören können. Die Größe der Tabelle lässt sich dadurch reduzieren. Eine vollständige Abbildung hätte eine kubische Tabellengröße zur Folge, was je nach Zellengröße schnell einige Gigabyte Speicher belegen würde. Die Hashtabelle sollte an die Objektzahl angepasst werden, wir verwenden etwa das Verhältnis 1:1. Die zum Objekt gehörigen Zellen werden nun ermittelt und alle Zellen, die das Objekt berührt, werden in die Hashtabelle eingetragen. Das sind kubisch viele Zellen, deshalb ist es sinnvoll, die Zellengröße geeignet groß zu wählen. Wir verwenden bei Billardkugeln

mit einem Durchmesser von 5,7 cm eine Zellengröße von 10 cm. Dieser Wert stellt einen sinnvollen Kompromiss zwischen schneller Kollisionserkennung (optimal: Zellengröße ≈ Objektgröße) und schnellem Einfügen in die Hashtabelle (besser: größere Zellen) dar.

Beim Test auf Kollision muss nun für jeden Ball geprüft werden, welche Objekte in den Zellen sind, in denen sich auch der Ball befindet. Ist die Zellengröße größer als der Ball, und befindet er sich genau in der Mitte, so muss nur für diese Zelle geprüft werden, welche Bälle sich noch in der Zelle befinden. Liegt der Ball auf der Trennfläche zwischen zwei Zellen, so müssen beide Zellen geprüft werden. Maximal kann ein Ball acht Zellen gleichzeitig berühren, wenn die Zellengröße mindestens so groß ist wie der Ball, häufig sind es aber nur ein oder zwei, was den Test sehr schnell macht. Funktionieren würde der Algorithmus für beliebige Zellengrößen, wenn sie aber zu klein gewählt wird, so müssen für jeden Ball unzählige Zellen geprüft werden, in denen sich viele Objekte befinden, da jedes Objekt in viele Zellen gehasht wird. Die gefundenen Kollisionen werden in einen Mengenvektor eingetragen, Duplikate sind nicht möglich. Jede Kollision wird also nur ein mal verarbeitet. Auf diese Weise lässt sich die Zahl der Tests auf ca. 100–300 pro Simulationsschritt reduzieren, was pro Ball etwa einem Duzend Tests auf andere Bälle und Faces entspricht.

Die Implementation ist eine Adaption des Algorithmus, der in der Vorlesung *Simulation der Computergrafik* [5] von Prof. Dr. Matthias Teschner vorgestellt wurde. Wir haben einige Verbesserungen vorgenommen, so ist das Einfügen in die Hashtabelle nun für jede Zelle in amortisiert konstanter Zeit möglich: Es wird nicht tatsächlich ein neues Element eingefügt, es wird nur ein schon vorhandener Platz als „belegt“ markiert. Desweiteren lassen sich beliebig geformte Objekte (Face und Ball) einfügen, und die Zellgröße ist beliebig wählbar, muss also nicht mindestens so groß wie das Objekt sein.

Anschließend werden zwischen Objekten, bei denen eine Kollision möglich ist, Tests durchgeführt. Der Ball-Ball-Kollisionstest ist einfach: Ist die Summe der Radien kleiner als der Abstand der Zentren, so besteht eine Kollision. Der Ball-Mesh-Kollisionstest ist etwas komplizierter und zweistufig aufgebaut, bei negativem Zwischenergebnis wird er abgebrochen. Zunächst wird ein schneller Test durchgeführt, ob der Ball die Dreiecksebene berührt. Ist das der Fall, berechnet eine kompliziertere Funktion `iNormal()` die Normale und die minimale Distanz zwischen Ballzentrum und Dreieck. Ist sie kleiner als der Ballradius, so findet eine Kollision statt. Behandelt wird ausschließlich die Kollision mit der größten Eindringtiefe. Das stellt sicher, dass in jedem Fall eine korrekte Normale berechnet werden kann.

Neben dem Spatial Partitioning mit gleichgroßen Zellen gibt es noch andere Methoden, um Kollisionen vorzeitig auszuschließen. Beim Billard sind die Kugeln ungleichmäßig im Raum verteilt. Sie konzentrieren sich auf den vergleichsweise kleinen Bereich auf der Tischplatte, wohingegen die Zellen in den Ecken des Raumes wohl nie eine Kugel zu

Gesicht bekommen. Eine andere Datenstruktur wie der „Octree“ könnte dieser Verteilung besser Rechnung tragen. Unser Ergebnis mit ca. einem Duzend Kollisionstests pro Ball und Simulationsschritt ist jedoch hinreichend gut, deswegen besteht keine Notwendigkeit, das gleichverteilte Spatial Partitioning durch ein komplizierteres Verfahren zu ersetzen.

Eine „Bounding-Volume-Hierarchie“ (BVH) könnte verwendet werden. Die Zahl der Tests zwischen Ball und relevantem, untersten Zweig des BVH-Tree ist dann logarithmisch. Verwenden wir für unseren Tisch mit 10.000 Faces sphärische Bounding Volumes und besitzt die BVH binäre Aufteilung, so hätten wir $\log_2(10.000)$ Tests, was auch ca. einem Duzend Tests auf Sphärenkollisionen pro Ball entspricht, nur für den Tisch. Auf diese Weise können wir die Zahl der Tests also kaum reduzieren. Vermutlich ist die BVH besser geeignet, wenn kompliziert geformte Objekte paarweise kollidieren.

2.2 Kollisionsbehandlung

Nachdem die Kollisionserkennung schnell nicht mögliche Kollisionen ausgeschlossen hat, und durch Tests tatsächliche Kollisionen entdeckt wurden, werden nun für jeden Ball alle Ball-Kollisionen und eine Mesh-Kollision bestimmt. Wir wählen die mit der größten Eindringtiefe in das Mesh. Dieses Verfahren stellt zwar nicht sicher, dass im nächsten Simulationsschritt alle Kollisionen aufgehoben sind, aber das ist auch ein komplexes Problem: Selbst wenn alle entdeckten Kollisionen beseitigt würden, wäre es möglich, dass die Kollisionsbehandlung zu neuen Kollisionen führt, etwa wenn ein Ball zwischen Bande und anderen Bällen eingeklemmt ist. Diese anderen Kollisionen würden wegen des Hashing-Verfahrens erst im nächsten Zeitschritt mit Sicherheit entdeckt. Mit einem iterativen Verfahren könnte man über alle Objekte in der Umgebung den Zustand der minimalen Kollision bestimmen. Der dafür notwendige Aufwand wäre aber beträchtlich. Wird nur eine Kollision pro Ball in jedem Zeitschritt behandelt, so ist es dagegen nicht notwendig, die Normale aus zwei Kollisionen zu interpolieren, sondern es kann die Normale der einzelnen Kollision verwendet werden.

Es lassen sich Situationen konstruieren, bei denen die Behandlung nur einer Kollision zu Problemen führt: Ist eine Kugel in eine Tasche gefallen, und eine andere darauf, so beginnt die untere Kugel zu oszillieren, da einerseits Druck von der oberen Kugel aus wirkt, und andererseits Gegendruck vom Boden der Tasche. Diese Schwingung ist sichtbar, gefährdet aber die Stabilität und Plausibilität der Simulation nicht. Desweiteren ist das eine Extremsituation, die im normalen Spiel nicht vorkommt, wir stapeln keine Kugeln. Die Behandlung mehrerer Kollisionen bringt uns also keine Vorteile.

2.2.1 Ball-Mesh-Kollision

Wir verwenden für den Zusammenstoß eines Balls mit einem Face eines Meshes eine impulsbasierte Kollisionsbehandlung. Dabei wird bei gleichbleibender Masse die Geschwindigkeit des Objekts direkt beeinflusst. Die Masse des Meshes wird als unendlich definiert, deshalb verwenden wir die Ball-Mesh-Kollision auch nur für den Tisch und das umgebende Zimmer. Die Auflösung entspricht im einfachsten Fall einer Spiegelung des Balls an der mit `iNormal()` berechneten Normalen $\mathbf{N}$ des Faces, die in Richtung Ball zeigt. Dazu zerlegen wir den Geschwindigkeitsvektor $\mathbf{v}$ in einen zu $\mathbf{N}$ tangentialen Anteil $\mathbf{v}_{\text{tang}}$ und einen senkrechten Anteil $\mathbf{v}_{\text{proj}}$ mit

$$\mathbf{v} = \mathbf{v}_{\text{tang}} + \mathbf{v}_{\text{proj}} = \underbrace{\mathbf{v} - \mathbf{v}_{\text{proj}}}_{\mathbf{v}_{\text{tang}}} + \mathbf{v}_{\text{proj}} = \underbrace{\mathbf{v} - \mathbf{v} \angle \mathbf{N}}_{\mathbf{v}_{\text{tang}}} + \mathbf{v} \angle \mathbf{N}.$$

Die Operation $\mathbf{a} \angle \mathbf{n}$ bezeichnet den binären Operator, bei der ein Vektor auf einen anderen projiziert wird, d.h. der Anteil von $\mathbf{a}$ der in Richtung $\mathbf{n}$ verläuft, wird zurückgegeben:

$$\mathbf{a}_{\text{proj}} = \mathbf{a} \angle \mathbf{n} = \frac{\mathbf{n}}{|\mathbf{n}|^2}(\mathbf{a} \cdot \mathbf{n})$$

Der tangentiale Teil wird nicht weiter betrachtet. Er verändert sich während der gesamten Kollision nicht. Der senkrechte Anteil wird negiert und mit Dämpfung d versehen.

Damit die Kollision im nächsten Zeitschritt nicht mehr vorhanden ist, wird die Position des Balls über die Oberfläche des Meshes, also über die Kollisionebene, zurückgesetzt. Weil während des Eindringens in das Objekt Zeit vergangen ist, stimmt der Berührpunkt genau auf der Oberfläche des Meshes nicht ganz: Während der Kollision ist der Ball mit der senkrechten Geschwindigkeit $\mathbf{v}_{\text{proj}}$ bis in eine Tiefe p eingedrungen. Dazu hat er $0 < k < h$ Zeit benötigt, in der er mit der neuen Geschwindigkeit $-d\mathbf{v}_{\text{proj}}$ (Dämpfung d) schon wieder hätte aufsteigen sollen. Es ist also

$$k = \frac{p}{\mathbf{v}_{\text{proj}}} \Rightarrow p' = -d\mathbf{v}_{\text{proj}} \cdot k = -dp$$

Deshalb ist er nach der Kollision in Wirklichkeit um dp über der Oberfläche.

Soweit der grundlegende Teil der Kollisionsauflösung für Meshes. Er macht vom Größenverhältnis her die Hauptkomponente einer korrekten Kollisionbehandlung aus, berücksichtigt aber keine Rotationsträgheit. Weil die Reibung zwischen Tisch und Kugeln beim Billard eine wichtige Rolle spielt, wird zusätzlich ein kraftbasierter Reibungsansatz angewendet, der in Abschnitt 2.4 erläutert wird.

2.2.2 Ball-Ball-Kollision

Der einfache Fall des vorhergehenden Abschnitts lässt sich ein wenig abgewandelt auch für die kompliziertere Ball-Ball-Kollision anwenden. Es gilt darauf zu achten, dass der

Gesamtimpuls $\mathbf{p}_{\text{sum}}$ des Systems erhalten bleibt, auch wenn die Bälle verschiedene Radien und Massen besitzen. In der Literatur ist der Fall für die Impulskollision zweier Sphären zwar beschrieben, aber die dreidimensionale, vektorbasierte Variante ist doch etwas schwieriger zu implementieren, als die häufig anzutreffende, zweidimensionale, skalare Darstellung es suggeriert. Ein vektorbasierter Ansatz [6] nimmt identische Massen an, wir wollen aber beliebige Massen betrachten. Daher wurde das Problem intuitiv gelöst und diese Intuition mit den Betreuern diskutiert, die immerhin die vermutete Richtigkeit bestätigten und keine Denkfehler fanden. Ein formaler Beweis wird nicht erbracht.

Zunächst einmal wurde versucht, die Reflexion des Balls von einer Ebene auch hier anzuwenden. Damit das möglich ist, muss die Kollisionsebene, also die Tangentialebene zwischen den Bällen „feststehen“, d.h. der Gesamtimpuls des Systems $\mathbf{p}_{\text{sum}}$ muss Null sein. Für diesen Fall wird angenommen, dass eine Reflexion an der Ebene korrekt sein sollte. Man stelle sich vor, die Kugel (Objekt 1) kollidiert mit einem anderen Objekt (Objekt 2), und der Gesamtimpuls des Systems ist $\mathbf{p}_{\text{sum}} = 0$. Er setzt sich zusammen aus den Einzelimpulsen beider Objekte, es gilt

$$\mathbf{p}_{\text{sum}} = m_1\mathbf{v}_1 + m_2\mathbf{v}_2 = 0.$$

Die tangentialen Anteile werden wieder außen vor gelassen: In tangentialer Richtung wirken keine Kräfte, es gibt keine Impulsänderung. Für den projektive (senkrechten) Anteil gilt daher:

$$m_1\mathbf{v}_{\text{proj}_1} + m_2\mathbf{v}_{\text{proj}_2} = 0.$$

Wir spiegeln nun den projektiven Anteil mit Dämpfung:

$$\begin{aligned}\mathbf{v}'_{\text{proj}_1} &= -d\mathbf{v}_{\text{proj}_1}\\ \mathbf{v}'_{\text{proj}_2} &= -d\mathbf{v}_{\text{proj}_2}\end{aligned}$$

also ist nach der Kollision

$$\begin{aligned}&-m_1 d\mathbf{v}_{\text{proj}_1} - m_2 d\mathbf{v}_{\text{proj}_2} &&= 0\\ \Rightarrow\quad &m_1\mathbf{v}'_{\text{proj}_1} + m_2\mathbf{v}'_{\text{proj}_2} &&= 0.\end{aligned}$$

Der Gesamtimpuls ändert sich durch die Kollision nicht und beide Objekte prallen an einer feststehenden Ebene ab. Anschließend setzen wir $\mathbf{v}$ wieder zusammen:

$$\begin{aligned}\mathbf{v}_1 &\leftarrow \mathbf{v}_{\text{tang}_1} + \mathbf{v}'_{\text{proj}_1}\\ \mathbf{v}_2 &\leftarrow \mathbf{v}_{\text{tang}_2} + \mathbf{v}'_{\text{proj}_2}\end{aligned}$$

Damit die Vorbedingung erfüllt ist und der Gesamtimpuls $\mathbf{p}_{\text{sum}} = 0$ ist, ziehen wir die Gesamt-Schwerpunktgeschwindigkeit $\mathbf{v}_{\text{sum}}$ vor dem Zusammenprall von den Objektgeschwindigkeiten ab, und addieren sie danach wieder dazu, sie ändert sich während des Vorgangs nicht. Die Schwerpunktgeschwindigkeit ist

$$\begin{aligned}\mathbf{v}_1 &\leftarrow \mathbf{v}_1 - \mathbf{v}_{\text{sum}}\\ \mathbf{v}_2 &\leftarrow \mathbf{v}_2 - \mathbf{v}_{\text{sum}}\end{aligned} \qquad \text{mit} \qquad \mathbf{v}_{\text{sum}} = \frac{m_1\mathbf{v}_1 + m_2\mathbf{v}_2}{m_1 + m_2}.$$

Auch bei der Ball-Ball-Kollision wird der Ball auf die Kollisionsebene zurückgesetzt zuzüglich der Positionsänderung, die sich durch die Eindringzeit ergeben hat. Das Resultat sieht richtig aus, das resultierende Verhalten der Kugeln erscheint vernünftig.

Auch hier wurde die Rotationsträgheit der Bälle bisher außer Acht gelassen. Wir verwenden wiederum den kräftebasierten Reibungsansatz. Er ist besonders für die Kollision zwischen dem Ball der Queuespitze und der weißen Kugel notwendig. Nur mit Hilfe der Reibung ist es möglich, der Kugel Effet mit auf den Weg zu geben, etwa einen Rückdrall durch tiefes Anstoßen der Kugel.

2.3 Integration

Die in *HapticBillard* verwendete Physik-Simulation stellt ein vereinfachtes Modell der realen Physik dar. Während in der realen Welt die Zeit ein kontinuierliches Medium ist, müssen in der Simulation diskrete Zustände zwischen zwei Zeitpunkten t und $t+h$ betrachtet werden. Der Übergang von einem Zustand in den nächsten ist abhängig von der Änderungsrate dieses Zustands. Beispielsweise ist die Änderungsrate eines Aufenthaltsortes $\mathbf{x}_t$ seine Ableitung nach der Zeit $\dot{\mathbf{x}}_t = \mathbf{v}_t$. Nach dem zweiten newtonschen Axiom ist

$$\mathbf{F}_t = \frac{d(m\mathbf{v}_t)}{dt} = m\frac{d\mathbf{v}_t}{dt}.$$

Bei konstanter Masse gilt:

$$m\frac{d\mathbf{v}_t}{dt} = m\frac{d^2\mathbf{x}_t}{dt^2}.$$

Zur Auflösung der Gleichung nach der Unbekannten $\mathbf{x}_t$ verwenden wir ein numerisches Integrationsverfahren. Die Formeln sind Folien der *Simulation der Computergrafik* [5] entnommen.

Anfänglich hatten wir das Runge-Kutta-4-Verfahren implementiert, was hohe Stabilität bei vertretbarer Rechenlast verspricht. In jedem Zeitschritt wird aber die Kraft vier mal berechnet. Durch das komplizierte Reibungssystem dauert das zu lange. Weil wir auf der anderen Seite keine stabilitätskritischen Bestandteile außer den unproblematischen Federn an den Bällen haben, können wir ein schnelleres, ungenaueres Verfahren einsetzen. Wir verwenden das Euler-Cromer-Verfahren mit einem Zeitschritt von $h = 4\,\text{ms}$.

Auch das Verlet-Verfahren wurde ausprobiert, da es kaum rechenintensiver ist als Euler-Cromer und höhere Genauigkeit verspricht. Die Position $\mathbf{x}_{t+h}$ berechnet sich aber beim Verlet aus der alten Position $\mathbf{x}_{t-h}$ und der Kraft, selbst wenn man die

Geschwindigkeit $\mathbf{v}_t$ in die Formel für die Position $\mathbf{x}_{t+h}$ mit einbaut:

$$\begin{aligned}\mathbf{x}_{t+h} &= 2\mathbf{x}_t - \mathbf{x}_{t-h} + h^2\frac{\mathbf{F}_t}{m} \\ &= \mathbf{x}_t + h\underbrace{\frac{\mathbf{x}_t - \mathbf{x}_{t-h}}{h}}_{\mathbf{v}_t} + h^2\frac{\mathbf{F}_t}{m} \\ &= \mathbf{x}_t + h\mathbf{v}_t + h^2\frac{\mathbf{F}_t}{m}\end{aligned}$$

Dann berechnet sich die Geschwindigkeit dennoch aus der alten Position:

$$\mathbf{v}_{t+h} = \frac{\mathbf{x}_{t+h} - \mathbf{x}_{t-h}}{2h} \approx \frac{\mathbf{x}_t - \mathbf{x}_{t-h}}{h}$$

Das hat zu Problemen mit der Impuls-Kollision geführt: Die Position des Balls wird bei Kollisionen direkt manipuliert. Die alte Position $\mathbf{x}_{t-h}$ wird damit ungültig, was das Verlet-Verfahren irritiert. Die Folge ist, dass der Ball hüpft. Beim verwendeten Euler-Cromer-Verfahren ist die Geschwindigkeit unabhängig von der alten Position, sie berechnet sich aus der Kraft und der alten Geschwindigkeit.

2.4 Reibungsmodell

Der maßgebliche Anteil am physikalischen Verhalten der Billardkugeln, wird durch die Impulskollisionsgesetze bestimmt. In vielen Billardsimulationen wird deshalb die Rotation der Kugel der Linearbewegung nachgeführt, das heißt, die Kugel rollt immer genau dorthin, wo sie sich hinbewegt. Doch das reicht nicht aus. Schlittern, Drall und Anschneiden der Kugeln lassen sich so nicht darstellen.

Aus diesem Grunde wurde ein kraftbasiertes Reibungsmodell implementiert. Es beruht auf der Formel für Haft- und Gleitreibungskräfte:

$$\mathbf{F}_R = -\mu|\mathbf{F}_N|\frac{\mathbf{v}_{\text{diff}}}{|\mathbf{v}_{\text{diff}}|}$$

Die Kraft $\mathbf{F}_R$ wirkt in entgegengesetzter Richtung des Differenzvektors $\mathbf{v}_{\text{diff}}$ (Rutschvektor) und bremst die Kugel ab. Die Größe der Kraft ist nur vom Reibungskoeffizient μ abhängig und von der Stärke des Anpressdrucks, der Normalkraft $\mathbf{F}_N$, nicht aber vom Geschwindigkeitsunterschied der beiden Kollisionspartner. μ bezeichnet zwei verschiedene Koeffizienten, μ_H für die Haftreibung und μ_G für die Gleitreibung. Den Differenzvektor $\mathbf{v}_{\text{diff}}$ zwischen zwei Kugeln berechnen wir mit

$$\mathbf{v}_{\text{diff}} = (\mathbf{v}_{\text{tang}_1} - \mathbf{v}_{\text{bahn}1}) - (\mathbf{v}_{\text{tang}_2} - \mathbf{v}_{\text{bahn}2})$$

wobei $\mathbf{v}_{\text{tang}_i}$ bzw. $\mathbf{v}_{\text{bahn}i}$ die Tangentialgeschwindigkeit bzw. die Bahngeschwindigkeit der Rotation ist:

$$\begin{array}{lcll} \mathbf{v}_{\text{tang}_i} & = & \mathbf{v}_i - \mathbf{v}_i \angle \mathbf{N}_i & \\ \mathbf{v}_{\text{bahn}i} & = & (\boldsymbol{\omega}_i - \boldsymbol{\omega}_i \angle \mathbf{N}_i) \times \mathbf{r}_i & i = \{1, 2\} \end{array}$$

$\mathbf{r}_i$ bezeichnet den Radialvektor von der Kugelmitte zum Berührpunkt, $\boldsymbol{\omega}_i$ die Winkelgeschwindigkeit. Der Differenzvektor zwischen einer Kugel und dem Mesh ist etwas einfacher, da hierfür nur eine Geschwindigkeit und eine Rotation betrachtet werden muss:

$$\mathbf{v}_{\text{diff}} = \mathbf{v}_{\text{tang}} - \mathbf{v}_{\text{bahn}}$$

Die Normalkraft $\mathbf{F}_N$ ist die Summe aus der Gravitation $\mathbf{F}_G$, projiziert auf die Kollisionsnormale $(-\mathbf{N})$ und der Kraft der Kollision $\mathbf{F}_C$.

$$\mathbf{F}_N = (\mathbf{F}_G \angle (-\mathbf{N})) + \mathbf{F}_C$$

Die Kollisionskraft $\mathbf{F}_C$ wirkt, wenn der Ball beim Auftreffen auf das Hindernis während eines Zeitschrittes gleichförmig von der Geschwindigkeit $\mathbf{v}_{\text{proj}}$ auf $-d\mathbf{v}_{\text{proj}}$ beschleunigt wird. Da unsere Kollision innerhalb von einem Zeitschritt der Dauer h aufgelöst wird, gehen wir von dieser Annahme aus.

$$\begin{array}{rcl} \mathbf{F}_C & = & m\mathbf{a} = m\dfrac{\Delta \mathbf{v}}{\Delta t} \\ & = & m\dfrac{\mathbf{v}_{\text{proj}} - (-d\mathbf{v}_{\text{proj}})}{h} \\ & = & m\dfrac{\mathbf{v}_{\text{proj}} + d\mathbf{v}_{\text{proj}}}{h} \\ & = & m(1+d)\dfrac{\mathbf{v}_{\text{proj}}}{h} \end{array}$$

Die Kraft $\mathbf{F}_R$ wirkt außerdem als Drehmoment auf den Ball ein mit

$$\mathbf{M}_R = \mathbf{r} \times \mathbf{F}_R$$

und beschleunigt seine Rotation. Man könnte im ersten Moment etwas verwirrt sein, dass die Kraft zweimal auf den Ball einwirkt und die Frage nach der Energieerhaltung stellen: Durch die Krafteinwirkung bewegt sich der Berührpunkt im Raum und zwar sowohl aufgrund der resultierenden Rotation als auch wegen der linearen Bewegung des Balls (die Drehachse bewegt sich im Raum und ist nicht fest). Der Weg des Berührpunktes ist dadurch länger. Die benötigte Arbeit („Kraft mal Weg“), um den Ball auf eine bestimmte Geschwindigkeit zu beschleunigen ist somit höher.

Durch das Wirken dieser Kräfte gleicht sich die relative Bewegung der Körper aneinander an. Wir nehmen für unsere Simulation an, dass nach dem vollständigen Angleichen der Bewegung keine Umkehr des Rutschens auftritt. Wenn man etwa einen Schraubenzieher in eine Kreissäge hält, wird sich ihre Rotation mit einem großen Knall umkehren. Vielleicht könnte ein Flummi-Ball auf Fliesen oder ähnlichem Untergrund mit

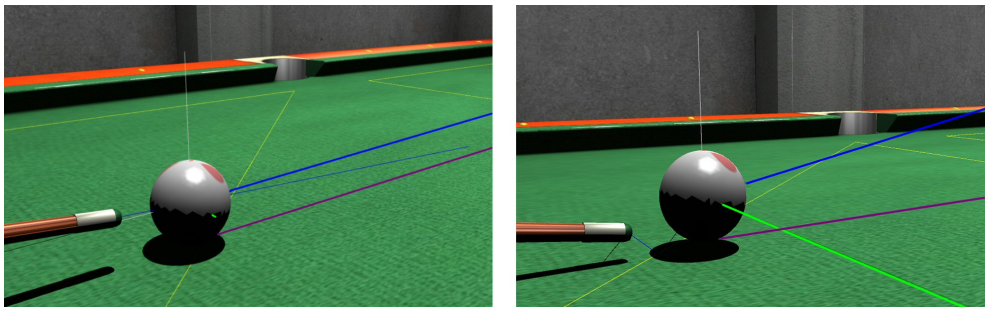


Abbildung 2.1: Im linken Bild wurde der Ball soeben hart mittig gespielt. Das Ergebnis ist eine hohe Geschwindigkeit (blau) und praktisch keine Rotation. Im rechten Bild wurde der Ball tiefer gespielt. Eine schnelle Rückwärtsrotation ist die Folge, zu erkennen an der Rotationsachse (grün). Unten am Ball ist der Differenzvektor zu sehen (violett). *(Schaubild der Software entnommen, Stärke der Linien zur Verdeutlichung nachbearbeitet.)*

hoher Reibung eine Spannung der Oberfläche aufbauen und damit seine Eigenrotation komplett umkehren. Wir denken aber, das ist nicht realistisch für unsere Simulation. Deshalb stoppt in unserem Modell der Reibungsausgleich, sobald keine Geschwindigkeitsdifferenz mehr auftritt.

Während das in der Realität wegen der kontinuierlichen Zeit einfach ist, wirkt bei uns die Kraft $\mathbf{F}_R$ einen ganzen Zeitschritt h lang, was viel zu lange sein kann. In dieser Zeit kann sich die Rotation bereits umgekehrt haben. Deshalb berechnen wir zunächst $\mathbf{F}_R$, integrieren einen Schritt in die Zukunft und testen, ob eine Rutschumkehr stattgefunden hat. Ist das der Fall, so verringern wir $\mathbf{F}_R$ gerade auf den Anteil, der das Rutschen in diesem Zeitschritt auf Null reduziert. Die verminderte Kraft $\mathbf{F}'_R$ ist

$$\mathbf{F}'_R = \frac{|\mathbf{v}_{\text{diff}}|}{|\mathbf{v}_{\text{diff}}| + |\mathbf{v}'_{\text{diff}}|}\mathbf{F}_R.$$

Dabei ist $\mathbf{v}'_{\text{diff}}$ die umgekehrte Rutschgeschwindigkeit nach der Integration.

Mit der universalen Reibungsformel und ihrer Erweiterung für diskrete Zeitschritte war es möglich, viele Verhaltensmuster von Billardkugeln realistisch nachzubauen. Egal ob es sich um den langsamen Übergang von einer rutschenden zu einer rollenden Kugel handelt, die Rotation, die umgekehrt, beinahe sogar gespiegelt wird, wenn die Kugel hart gegen die Bande prallt, all das bewirkt diese eine Formel. Auch fortgeschrittenes Spielen wird möglich: Durch versetztes Anspielen mit dem Queue lassen sich „Stopp-Bälle“ spielen: Die weiße Kugel wird etwas tiefer gespielt, damit sie einen Rückwärtsdrall erhält und nach einer (nahen) Kollision mit einer farbigen Kugel sofort anhält. Mit dieser Technik wird beispielsweise verhindert, dass die weiße Kugel einer farbigen Kugel in die Tasche folgt. Ein Beispiel ist in Abb. 2.1 zu sehen.

2.5 Herausforderungen

Mit der Kollisionserkennung und -behandlung und dem Reibungsmodell konnte eine physikalische Simulation für *HapticBillard* geschaffen werden, die den Ansprüchen an ein realitätsnahes Billardspiel in weiten Teilen genügt. Ihr Konzept wurde nach physikalischen Gesetzen entworfen. So wurde darauf geachtet, dass der Impulserhaltungssatz eingehalten wird, dass Kräfte nicht aus dem Nichts entstehen, sondern eine identische Gegenkraft existiert (oder eine sinnvolle Erklärung wie die Schwerkraft) und das Reibungsmodell kann auf eine einfache Formel zurückgeführt werden.

Eine Physik-Engine für ein Computerspiel ist dann gelungen, wenn sie am unauffälligsten ist. Wenn sich alle Objekte so bewegen wie man es erwarten würde und man die Dinge tun kann, die man auch in der Realität tun kann. Das ist bei uns im Vergleich zu anderen Billardsimulationen vielversprechend. Und doch gibt es viele Möglichkeiten, unsere Physik-Simulation zu verbessern.

Ein Problem ist das Oszillieren ruhender Kugeln. Grundsätzlich wird es verursacht, weil die Kugeln leicht auf- und ab-hüpfen: In einem Zeitschritt dringen sie wegen der Schwerkraft in den Boden ein, die Kollision wird erkannt, sie prallen ab und befinden sich im nächsten Zeitschritt in der Luft. Das geschieht in einer Größenordnung von unter 0,1 mm und so schnell dass es theoretisch nicht gesehen werden könnte. Weil aber die Physiksimulation und das Rendering asynchron laufen, ergibt sich ein zeitliches Aliasing mit dem Ergebnis, dass Kugeln im Extremfall eine zeitlang einige Pixel höher liegen, danach tiefer, sie oszilieren mit 2–10 Hz.

Verschiedene Schwellwertfunktionen wurden implementiert, um Kugeln einfach anzuhalten, wenn sie ruhen. Alle diese Methoden führten jedoch zu einer Beeinflussung der physikalischen Eigenschaften. Kugeln, die mit den Gegenmaßnahmen ausgestattet waren, rollten und sprangen anders, als welche ohne. Wir deaktivierten die Methoden und vertagten das Problem, zumal es nur für den kritischen Blick erkennbar ist.

Ein anderes Problem ist, dass sehr kleine Kugeln das Mesh durchdringen können. Bälle von unter 2 cm Durchmesser, könnten in den Tisch hineingeraten, wenn sie hart gestoßen werden. Ein schneller Ball bewegt sich mit ca. 5 m/s durch den Raum. In einem Timestep von $h = 4$ ms legt er 20 mm zurück. Ein Ball, der zu mehr als der Hälfte im Mesh steckt, wird als auf der anderen Seite angesehen. Der kleinen Billardspitze reicht das, um den Tisch zu durchdringen. Für die Flexibilität, beliebige Ebenen von beliebigen Seiten aus kollidieren zu können, ohne *innen* und *außen* definieren zu müssen, nehmen wir das in Kauf. Mit den Billardkugeln haben wir dagegen keine Probleme. Ihre Mesh-Kollision funktioniert absolut zuverlässig.

Man könnte neben den genannten Vorschlägen die impulsbasierte Kollision auf eine kraftbasierte Kollision umstellen und die Kraft so berechnen, dass die resultierende Geschwindigkeit im nächsten Zeitschritt der benötigten entspricht. Dann summieren sich bei gleichzeitigen Kollisionen zunächst die Kräfte auf, anstatt dass nur die zuletzt

betrachtete Kollision wirksam wird. Desweiteren könnte eine Sub-Simulation eingeführt werden, die kritische Momente wie Kollisionen mit kleinerem Zeitschritt rechnet um eine höhere Genauigkeit zu erreichen, gleichzeitig uninteressante Vorgänge, wie der Geradeauslauf oder die Ruhelage von Kugeln mit größerem Zeitschritt, um Rechenzeit einzusparen. Diese Vorschläge sind nur die Wichtigsten von vielen weiteren Verbesserungen, die man vornehmen könnte, um die Qualität der Simulation noch weiter zu steigern.

3 Spielsteuerung

Ein Ziel des Projekts war die Integration des PHANTOM Omni®Haptic Device in die Spielsteuerung, was der Schwerpunkt von **Benjamin Ummenhofer** war. Das Haptic Device unterscheidet sich deutlich von konventionellen Eingabegeräten wie Maus, Tastatur oder Gamepad. Es unterstützt zur Positionsbestimmung sechs Freiheitsgrade, davon drei Raumrichtungen sowie drei Rotationen und bietet Force Feedback in drei Raumrichtungen um haptische Rückmeldungen zu ermöglichen. Damit die Steuerung mit dem PHANTOM Omni Haptic Device ein möglichst realitätsnahes Spielgefühl bietet, musste sowohl die Verarbeitung der Eingabegeräte, als auch das Steuerungskonzept, speziell auf dieses Gerät zugeschnitten werden.

Im ersten Abschnitt gehen wir zunächst auf die von uns entwickelte Eingabeverwaltung ein, die eine ereignisbasierte Verarbeitung von Maus, Tastatur und haptischem Eingabegerät ermöglicht. Wir erklären die technischen Unterschiede zwischen dem haptischem Eingabegerät und anderen Eingabegeräten.

Der zweite Abschnitt soll die Ideen hinter der Benutzeroberfläche beschreiben und ihre Umsetzung erläutern.

Der letzte Abschnitt behandelt die haptische Interaktion. Wir stellen verschiedene Methoden vor mit denen haptische Rückmeldung modelliert werden kann. Danach stellen wir konkret zwei Anwendungsbeispiele für die entwickelten Methoden vor.

3.1 Eingabeverwaltung

Die Eingabeverwaltung unterstützt die drei Eingabegeräte Maus, Tastatur und das PHANTOM Omni Haptic Device (kurz Haptic Device). Maus und Tastatur werden über das *Object-Oriented Input System (OIS)* angesprochen. Das Haptic Device wird über die *OpenHaptics API* [7] angesteuert. Die Eingabeverwaltung übernimmt im Wesentlichen zwei Aufgaben. Sie ermöglicht eine einheitliche ereignisbasierte Verarbeitung für alle Eingabegeräte und sie entkoppelt die Hauptschleife des HapticRenderer, die Klasse, die die Darstellung der ForceFeedback- Effekte übernimmt, vom Rest der Simulation.

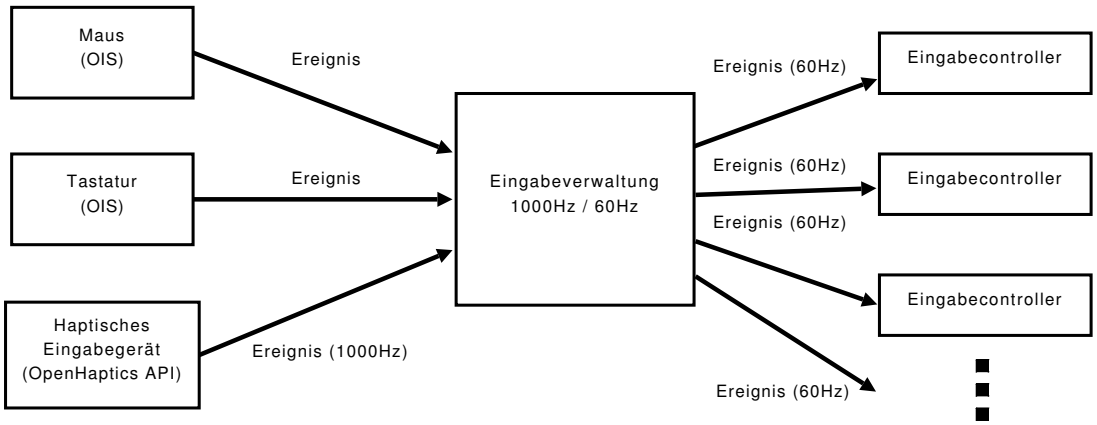


Abbildung 3.1: Ereignisbasierte Eingabeverwaltung. Die Eingabegeräte (links) generieren Ereignisse bei Zustandsänderungen. Die Eingabeverwaltung (Mitte) leitet die Ereignisse an die entsprechenden Eingabecontroller (rechts) weiter, die die eigentliche Verarbeitung übernehmen.

3.1.1 Ereignisverarbeitung

Die Verarbeitung der unterstützten Eingabegeräte erfolgt ereignisbasiert. Ändert sich der Zustand eines Eingabegeräts, zum Beispiel weil ein Knopf gedrückt wurde, dann generiert das Eingabegerät ein entsprechendes Ereignis. Das Ereignis wird durch die Eingabeverwaltung an die entsprechenden Eingabecontroller weitergeleitet, wo die eigentliche Verarbeitung stattfindet. Ob ein Ereignis weitergeleitet wird, hängt von zwei Faktoren ab. Erstens, beim Erstellen eines Eingabecontrollerobjektes wird festgelegt für welche Eingabegeräte Ereignisse empfangen werden sollen. Zweitens, das Profil, zu dem der Eingabecontroller gehört, muss aktiviert sein.

Die Profile erlauben eine flexible Steuerung der Ereignisverarbeitung. Dabei können einem Profil mehrere Eingabecontroller zugeordnet sein. Dadurch wird es möglich Eingabecontroller zu kombinieren. So kann ein Eingabecontroller die Verarbeitung des Haptic Device übernehmen, während ein Anderer auf Tastatureingaben reagiert. Gleichzeitig ist es möglich einem Eingabecontroller mehrere Profile zuzuordnen. Es gibt also eine $n : m$-Relation zwischen Eingabecontrollern und Profilen. Ein Eingabecontroller mit mehreren Profile ist dann aktiv, wenn mindestens eines seiner Profile aktiv ist. Soll ein Eingabecontroller beispielsweise immer aktiv sein, dann können ihm alle Profile zugeordnet werden. In der konkreten Implementation gibt es für diesen Fall ein spezielles Profil, das diese Aufgabe übernimmt.

3.1.2 Entkopplung des HapticRenderer

Die HapticRenderer-Klasse übernimmt die Ansteuerung des PHANTOM Omni Haptic Device. Die Berechnung der ForceFeedback-Effekte verwendet einen eigenen Thread, der asynchron zum Rest der Simulation läuft. Die Kräfte werden mit 1000 Hz aktualisiert. Ereignisse, wie Positionsänderungen oder das Drücken von Tasten, werden ebenfalls bis zu 1000−mal in der Sekunde ausgelöst. Anders als bei Maus und Tastatur werden diese Ereignisse nicht sofort verarbeitet. Da sich beliebig viele Eingabecontroller für ein Ereignis registrieren können, kann nicht gewährleistet werden, dass die Verarbeitung innerhalb einer Millisekunde abgeschlossen ist. Folglich darf die Verarbeitung nicht im Thread des HapticRenderer erfolgen sondern muss in der Hauptschleife der Simulation stattfinden.

Die Eingabeverwaltung übernimmt die Synchronisation der beiden Threads und speichert die Ereignisse des Haptic Device zur späteren Verarbeitung zwischen. Das Ereignis „Positionsänderung“ wird sehr häufig ausgelöst, da es sehr schwierig ist den Stift perfekt ruhig zu halten. Um die Anzahl der Ereignisse und damit den Verarbeitungsaufwand gering zu halten, wird nur das jeweils neueste Ereignis diesen Typs zwischengespeichert. Dadurch wird sichergestellt, dass nur das zeitlich nächste Ereignis verarbeitet werden muss. Im Gegensatz dazu müssen bei einem Knopfdruck alle Ereignisse gespeichert werden, damit kein Drücken oder Loslassen übersehen wird.

Bevor die eigentliche Weiterleitung und Verarbeitung der Ereignisse beginnt, übergibt die Ereignisverwaltung dem Scheduler der HapticRenderer-Hauptschleife eine Callbackfunktion, die die Erzeugung weiterer Ereignisse stoppt. Der Scheduler synchronisiert den Aufruf mit der HapticRenderer-Hauptschleife und gewährleistet damit den Wechselseitigen Ausschluss. Die Eingabeverwaltung hat damit alleinigen Zugriff auf die Ereignisse. Nach Verarbeitung aller Ereignisse wird die Ereignisgenerierung wieder über eine synchrone Callbackfunktion gestartet.

3.2 Benutzeroberfläche

Zur Spielsteuerung gehört auch die Benutzeroberfläche. Sie ermöglicht das Ändern von Spieleinstellungen und die Auswahl der Spielmodi. Zu unseren Designzielen gehörten zum Einen eine einfache intuitive Menüführung und zum Anderen sollte die Benutzeroberfläche vollständig mit dem Haptic Device bedienbar sein ohne Maus oder Tastatur zu Hilfe nehmen zu müssen.

3.2.1 Menüstruktur

Für die Umsetzung wurde das *CEGUI* Menüsystem [8] verwendet. Das Layout der Menüs wird über Textdateien mit XML Syntax definiert. Um die Bedienung einfach

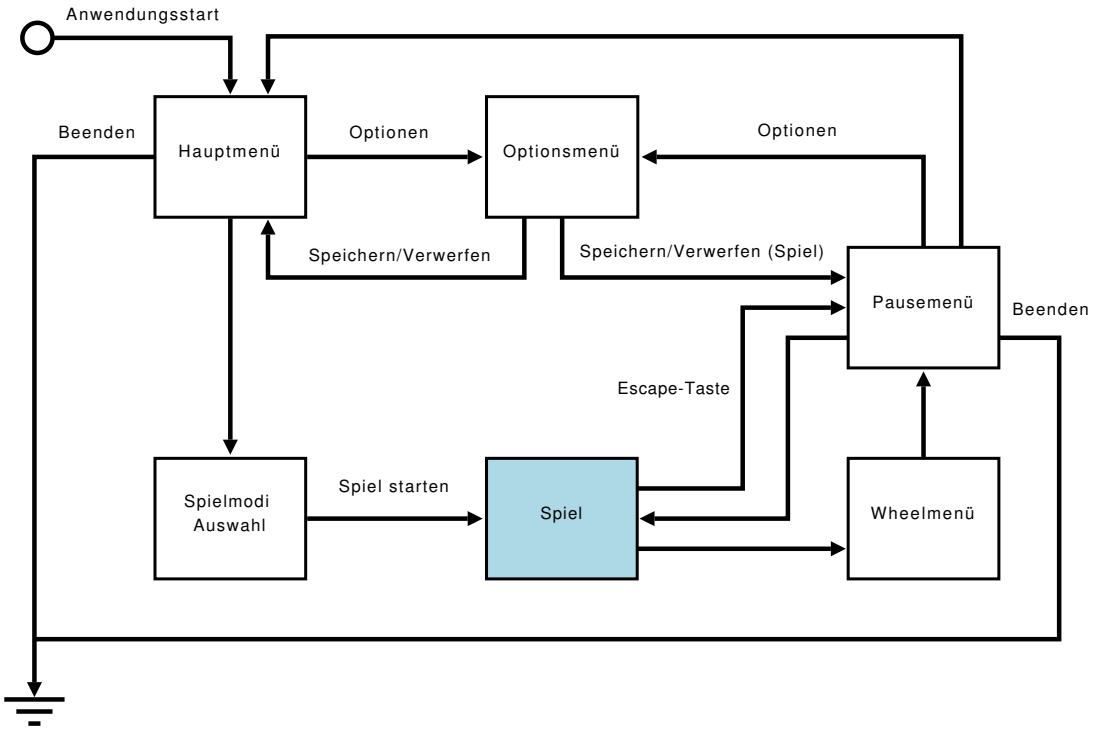


Abbildung 3.2: Menüführung. Die Anwendung startet mit dem Hauptmenü. Das eigentliche Billardspiel ist mit dem blauen Rechteck gekennzeichnet und stellt kein Menü dar.

zu halten verwenden wir nur die gebräuchlichsten Komponenten wie Knöpfe, Slider und Comboboxen.

Wir haben fünf Menüs definiert mit denen die Anwendung gesteuert werden kann. Es folgt jeweils eine kurze Beschreibung.

Hauptmenü: Das Hauptmenü ist der Ausgangspunkt. Es wird direkt nach dem Start der Anwendung angezeigt. Es ist verknüpft mit dem Optionsmenü und dem „Spielmodus Auswahl“-Menü. Außerdem kann die Anwendung von hier aus beendet werden.

Spielmodus Auswahl: In diesem Menü wird der Spieltyp und die Spieler ausgewählt. Als Spielmodi stehen 8-Ball und ein freies Spiel ohne Regeln zur Auswahl. Für die beiden Spieler kann jeweils ausgewählt werden, ob der Computer die Kontrolle übernehmen soll oder ob es sich um einen menschlichen Mitspieler handelt.

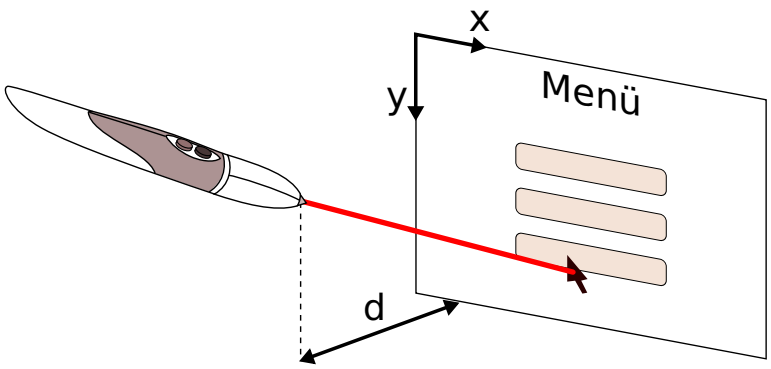


Abbildung 3.3: Abbildung der Raumkoordinaten des Haptic Device auf die Bildschirmkoordinaten. Der Schnittpunkt des verlängerten Stifts mit der virtuellen Bildschirmebene im Abstand d bestimmt die Position des Mauszeigers.

Optionsmenü: Im Optionsmenü können die Grafikeinstellungen , die Mausgeschwindigkeit, die Stärke des KI-Spielers und weitere Einstellungen verändert werden.

Pausemenü: Das Pausemenü kann aufgerufen werden, während ein Billardspiel läuft. Es kann das Optionsmenü aufgerufen werden um während des Spiels Einstellungen zu verändern. Außerdem gibt es Knöpfe um zum Hauptmenü zurückzukehren oder die Anwendung unmittelbar zu beenden.

Wheelmenu: Das Wheelmenu ist ein spezielles Menü, da es zur direkten Steuerung der Billardsimulation benutzt wird. Während des Spiels kann hier beispielsweise zwischen Queue- und Kamerasteuerung umgeschaltet werden. Es enthält auch einen Knopf mit dem das Pausemenü erreicht werden kann. Die Bedienung unterscheidet sich von der Bedienung der anderen Menüs und wird in Abschnitt 3.2.2 näher beschrieben.

Die fünf Menüs sind untereinander verknüpft, Abbildung 3.2 zeigt wie. Entscheidend für die Steuerung mit dem Haptic Device ist die Verknüpfung zwischen Pausemenü und dem Wheelmenu. Diese Verknüpfung erlaubt es das gesamte Spiel durchweg mit dem Haptic Device zu bedienen.

Die Steuerung des Wheelmenu wurde in erster Linie für das Haptic Device entworfen. Alle anderen Menüs sind für eine Maussteuerung ausgelegt. Der folgende Abschnitt erklärt wie die Steuerung der „Mausmenüs“ und die Steuerung des Wheelmenu funktionert.

3.2.2 Menüsteuerung mit dem Haptic Device

Damit die Anwendung allein mit dem Haptic Device bedient werden kann, haben wir für die Menüs eine Mausemulation entworfen. Das Haptic Device liefert eine Raumposition relativ zum Ursprung seines Arbeitsbereichs. Eine Maus hingegen liefert nur die Veränderungen einer Position in der Ebene. Um den Mauscursor mit dem Haptic Device zu steuern muss die absolute Position im Raum auf die Bildschirmkoordinaten abgebildet werden. Die Abbildung hat folgende Form:

$$(X, Y, Z) \rightarrow (x_s, y_s)$$

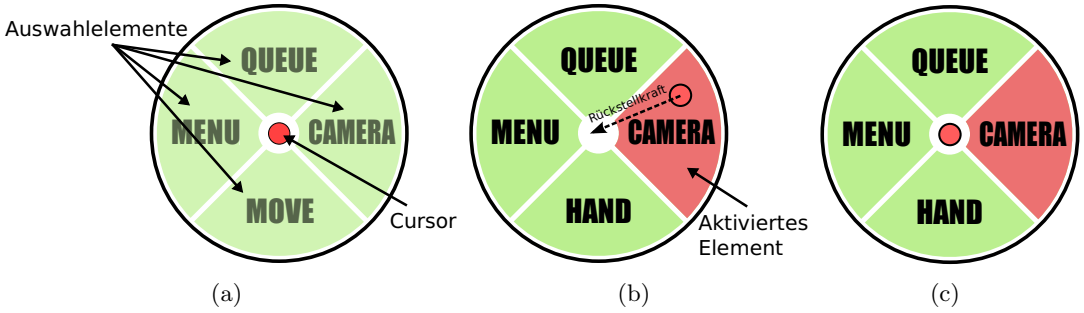
Wir wählen zunächst einen naiven Ansatz und ignorieren die Z-Koordinate, um die Abbildung zu vereinfachen. Der Ansatz funktioniert wie erwartet, berücksichtigt aber nicht die Art und Weise wie das Eingabegerät gehalten wird. Das Haptic Device wird wie ein Stift gehalten. Ein Großteil der Bewegung kommt aus dem Handgelenk, damit einher geht eine Veränderung der Orientierung des Stifts. Will man beispielsweise den Mauscursor nach links bewegen, dann zeigt man aufgrund der Bewegung des Handgelenks automatisch mit dem Stift nach links.

Wir wollen die Änderung der Orientierung in die Berechnung der Mauscursorposition einfließen lassen, um die Steuerung besser an die menschliche Hand anzupassen. Abbildung 3.3 illustriert die Idee. Ähnlich wie bei einem Laserpointer soll die Position des Mauscursors auf den Bildschirm projiziert werden. Der Abstand vom Stift zur Bildschirmebene bezeichnen wir mit d. Er wird fest gewählt. Sei $\mathbf{z}$ die lokale Z-Achse des Stifts, also die Richtung, in die der Stift zeigt, dann kann die Position des Mauscursors wie folgt berechnet werden.

$$\begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \end{pmatrix} \cdot \left[\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} + d\mathbf{z} \right] + \begin{pmatrix} t_x \\ t_y \end{pmatrix} = \begin{pmatrix} x_s \\ y_s \end{pmatrix}$$

Zusätzlich zur Projektion müssen die Koordinaten noch entsprechend skaliert und verschoben werden. Die Parameter für die Skalierung s_x und s_y werden bestimmt aus den Dimensionen des Arbeitsbereichs des Haptic Device und aus der Bildschirmauflösung. Der Parameter s_y ist negativ, weil die beiden Y-Achsen entgegengesetzte Richtungen haben. Die Vektor $\mathbf{t} = (t_x, ty)^T$ verschiebt den Ursprung des Haptic Device in den Mittelpunkt des Bildschirms.

Anstatt d fest zu wählen kann auch die Z-Koordinate des Haptic Device verwendet werden. Damit kann der Abstand des Stifts zur Bildebene beeinflusst werden. In der Praxis konnten wir dadurch keinen Vorteil erkennen, da der Abstand d in der Regel nicht bewusst zur Steuerung eingesetzt wurde.



(a) (b) (c)

Abbildung 3.4: Das Wheelmenu: (a) Menüaufbau. (b) Ein Element wurde ausgewählt und eine Rückstellkraft wirkt. (c) Der Cursor ist wieder im Ausgangszustand. Das Element bleibt ausgewählt.

Wheelmenu

Das Wheelmenu unterscheidet sich von den anderen Menüs durch seine Steuerung. Die Auswahlelemente des Menüs sind kreisförmig angeordnet. Abbildungen 3.4(a) bis (c) zeigt schematisch wie ein Element ausgewählt werden kann. Es kann während eines Billardspiels durch Gedrückthalten der rechten Maustaste oder des zweiten Knopfs des Haptic Device aufgerufen werden. Ein Element wird ausgewählt, indem der Cursor aus der Mitte über das Element bewegt wird und die rechte Maustaste losgelassen wird. Ein Element bleibt solange ausgewählt bis der Cursor ein anderes Element berührt. Eine Rückstellkraft (siehe Abbildung 3.4(b)), sorgt dafür, dass der Cursor und damit das Haptic Device wieder in die Ausgangslage zurückkehrt.

Der Vorteil dieses Methode ist, dass sich bei der Benutzung des Menüs die Raumposition des Haptic Device kaum verändert. Außerdem müssen keine großen Bewegungen durchgeführt werden um ein Element auszuwählen. Es reicht aus das Haptic Device kurz in die Richtung des auszuwählenden Elements auszulenken. Damit ist die Raumposition beim Öffnen des Menüs nahezu identisch mit der Position beim Schließen.

Wird das Menü beispielsweise aufgerufen während der Queue aktiv ist, dann kann der Spieler, unterstützt durch die Rückstellkraft, die alte Position wiederfinden, wenn er doch kein anderes Element auswählen möchte.

Eine Garantie, dass die Position des Haptic Device erhalten bleibt, kann durch die Rückstellkraft natürlich nicht erreicht werden.

3.3 Haptische Interaktion

Das PHANTOM Omni®Haptic Device unterstützt Force Feedback mit drei Freiheitsgraden. Für eine möglichst realistische Haptiksimulation, müssen diese Kräfte mit hoher Frequenz aktualisiert werden. Die HapticRenderer-Klasse übernimmt diese Aufgabe. Die Kräfte werden direkt in der Hauptschleife, die das Haptic Device ansteuert, berechnet. Die HapticRenderer-Klasse stellt eine einfache Schnittstelle zur Verfügung mit so genannten Haptikeffekten. Ein Haptikeffekt kann beispielsweise eine Feder oder eine Kugel sein. Die Kräfte dieser Effekte werden mit einer Frequenz von 1000 Hz aktualisiert.

Der folgende Abschnitt stellt die von uns implementierten Haptikeffekte vor. Die Abschnitte 3.3.2 und 3.3.3 zeigen wie diese Effekte eingesetzt werden, um entweder eine möglichst leicht zu bedienende Kamerasteuerung zu realisieren oder realistisches Force Feedback für den Billardqueue zu erreichen.

3.3.1 Haptik Effekte

Der einfachste Effekt, der „Konstante Kraft“-Effekt, lässt eine gleichbleibende Kraft wirken. Effekte können beliebig oft modifiziert werden, dadurch eignet sich dieser Effekt, um beispielsweise Kräfte aus der Physiksimulation direkt anzuwenden.

Die anderen Effekte modellieren geometrische Objekte. In diesem Fall werden auf das Haptic Device Kräfte ausgeübt, die das Verlassen des geometrischen Objekts oder das Eindringen verhindern. Die berechneten Kräfte sind Penaltykräfte, deren Beträge proportional zur Verletzung des jeweiligen geometrischen Constraints sind und in die Richtung zeigen, die die Verletzung am schnellsten reduziert.

Die folgenden grundlegenden geometrischen Haptikeffekte wurden implementiert:

Punkt: Der Punkteffekt ist eine Federkraft die das Haptic Device auf eine angegebene Raumposition zieht. Er wird beispielsweise im Wheelmenu und in der Kamerasteuerung für die Rückstellkraft eingesetzt.

Gerade: Das Haptic Device wird auf einer Gerade im Raum gehalten. Es kann sich frei auf der Gerade bewegen. Bei Verletzung des Constraints wirkt eine Kraft, die das Haptic Device auf den nächsten Punkt auf der Gerade zieht. Die Queuesteuerung benutzt diesen Effekt zur Stabilisierung.

Ebene: Es wird eine Kraft ausgeübt sobald die angegeben Ebene im Raum vom Haptic Device durchdrungen wird. Die Kraft ist proportional zur Eindringtiefe und die Richtung ist die Normale der Ebene. Die Kamerasteuerung benutzt diesen Effekt um haptisches Feedback bei Kamerabewegungen zu geben.

Kugel: Es wird eine Kugel im Raum modelliert. Dringt das Haptic Device in die Kugel ein wirkt eine radiale Kraft proportional zur Eindringtiefe. Der Effekt wird benutzt um die zu stoßende Billardkugel zu modellieren.

Es können mehrere Effekte gleichzeitig aktiviert werden. Die berechneten Kräfte werden dann überlagert. Die Aktivierung eines Haptikeffekts erfolgt über synchrone Callbacks und funktioniert ähnlich wie die in Abschnitt 3.1.2 beschriebene Unterbrechung der Ereignisgenerierung.

3.3.2 Kamerasteuerung

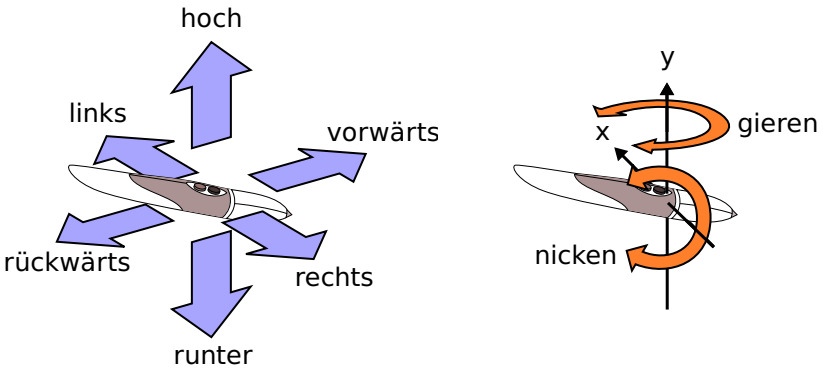
Wir haben in HapticBillard eine Kamera implementiert, die es dem Spieler erlaubt den Billardtisch aus der Ich-Perspektive zu begutachten. Die Kamera kann sowohl mit Tastatur und Maus als auch mit dem Haptic Device gesteuert werden. Für beide Steuerungsmöglichkeiten schränken wir die Kamera in einem Freiheitsgrad ein. Wir erlauben es nicht die Kamera zu rollen (Drehung um die Achse der Blickrichtung). Es verbleiben fünf Freiheitsgrade: Die Drehung um die X-Achse der Kamera (Nicken), die Drehung um die Y-Achse der Kamera (Gieren) und die Bewegung in die drei Raumrichtungen.

Für die Steuerung ohne Haptic Device verwenden wir eine herkömmliche Kamerasteuerung, die Maus und Tastatur kombiniert. Die Bewegung der Kamera wird mit der Tastatur gesteuert, die Blickrichtung wird mit der Maus verändert.

Für die Steuerung mit dem PHANTOM Omni®Haptic Device stellen wir zwei Varianten vor mit denen eine Kamerasteuerung realisiert werden kann.

Variante 1: Bei Aktivierung der Kamerasteuerung wird die letzte Position als Nullposition gespeichert. Die Kamera wird bewegt, indem das Haptic Device aus der Nullposition ausgelenkt wird. Die Richtung und Geschwindigkeit der Bewegung wird durch die Richtung und Größe der Auslenkung bestimmt. Die Bewegungen erfolgen immer im Koordinatensystem der Kamera.

Die Drehungen der Kamera wird mit der Orientierung des Haptic Device gesteuert. Wird der Stift nach links oder rechts gedreht, giert die Kamera. Die Kamera kann genickt werden, indem man den Stift nach oben oder unten zeigen lässt. Die Kamera dreht sich nicht, wenn die Richtung, in die der Stift zeigt, parallel zur Z-Achse des



(a) (b)

Abbildung 3.5: Steuerungsvariante 1: (a) Wird das Haptic Device im Arbeitsbereich in eine der Pfeilrichtungen ausgelenkt, dann beginnt sich die Kamera in die angegebenen Richtungen zu bewegen. (b) Die Blickrichtung der Kamera kann durch das Drehen des Haptic Device verändert werden. Durch kippen des Stiftes nach oben oder unten wird die Kamera um ihre X-Achse gedreht (nicken). Durch drehen des Stiftes nach links oder rechts kann die Kamera um ihre lokale Y-Achse gedreht werden (gieren).

Arbeitsbereichs des Haptic Device ist. Abbildung 3.5 zeigt wie die Stiftbewegungen in Kamerabewegungen umgesetzt werden.

Mit der bis hierhin beschriebenen Steuerung ist es praktisch unmöglich die Nullposition zu finden, um die Kamera still zu halten.

Um das Problem zu lösen, führen wir Schwellwerte ein. Erst wenn die Schwellwerte überschritten werden, bewegt oder dreht sich die Kamera. Der Schwellwert für die Translation entspricht einer Entfernung zum Beispiel $5mm$. Der Schwellwert für die Rotation ist ein Winkel.

Zusätzlich benutzen wir die oben vorgestellten Haptikeffekte, um es dem Spieler zu erleichtern den Nullpunkt wiederzufinden. Eine Federkraft zieht das Haptic Device ständig auf den Nullpunkt. Zusätzlich zur Federkraft benutzen wir sechs Ebeneneffekte, die wie die Flächen eines Würfels angeordnet werden. Der Mittelpunkt des Würfels ist die Nullposition. Diese stellen die Schwellwerte für die Raumrichtungen dar und verursachen eine zusätzliche Kraft, die die Schwellwerte erfühlbar machen sollen.

Ein Problem stellt die Orientierung des Stiftes dar. Das PHANTOM Omni®Haptic Device kann kein Moment ausüben um den Stift wieder in die Nulllage der Orientierung zu drehen oder den Widerstand für die Schwellwerte zu modellieren.

Trotzdem ist diese Steuerungsvariante brauchbar und erlaubt mit ein wenig Übung eine genaue Steuerung der Kamera.

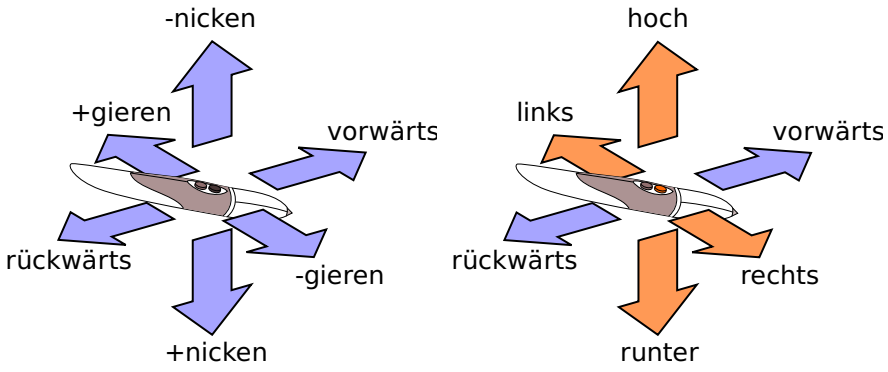


Abbildung 3.6: Steuerungsvariante 2: (a) Nur die Auslenkung nach vorne oder hinten bewegt die Kamera. Die anderen Raumrichtungen erlauben es die Kamera zu drehen. (b) Durch das Drücken der vorderen Taste wird die Funktion der vier Raumrichtungen oben, unten, links, rechts geändert. Die Steuerung funktioniert nun identisch wie in Variante 1 (vgl. Abb. 3.5(a)).

Variante 2: Die zweite Variante unterscheidet sich dadurch, dass die Rotation der Kamera nicht durch die Orientierung des Stiftes gesteuert wird, sondern durch die Auslenkung von der Nullposition. Wird der Stift nach oben, unten, links oder rechts ausgelenkt, so dreht sich die Kamera. Mit der vorderen Taste des Stiftes kann das Verhalten dieser vier Auslenkrichtungen geändert werden. Wird die Taste gedrückt, dann verhalten sich diese vier Auslenkrichtungen identisch wie in Variante 1. Abbildung 3.6 zeigt, wie sich die Funktionsbelegung der Auslenkrichtungen bei Tastendruck ändert. Der Nachteil dieser Variante ist, dass sie eine Taste benötigt um alle Funktionen zu erreichen.

Fazit

Beide Varianten sind für die Kamerasteuerung geeignet. Wir haben uns für die zweite Variante entschieden. Variante 1 kommt ohne Tasten aus, hat aber den Nachteil, dass hier die Kameradrehungen schwieriger zu steuern sind aufgrund des fehlenden haptischen Feedbacks. Bei Geräten, die Momente ausüben können, ist vermutlich Variante 1 die bessere Wahl.

3.3.3 Queuesteuerung

Eine Besonderheit der Steuerung des Billardqueues ist die direkte haptische Interaktion mit der Spielwelt. Um Kollisionen und Kräfte berechnen zu können, muss der

Arbeitsbereich des Haptic Device sinnvoll in die Spielwelt transformiert werden.

Transformation des Arbeitsbereichs

Der Arbeitsbereich des PHANTOM Omni®Haptic Device ist ein Quader mit Breite $160mm$, Höhe $120mm$ und Tiefe $70mm$. Innerhalb dieses Arbeitsbereichs können Kräfte bis zu $0{,}88N$ über einen längeren Zeitraum angewendet werden.

Damit der Spieler mit dem Queue und den Kugeln interagieren kann werden Position und Orientierung des Haptic Device in die Spielwelt transformiert und die Kräfte aus der Simulation in den Arbeitsbereich des Haptic Device abgebildet (siehe Abbildung 3.7). Eine sinnvolle Abbildung des Arbeitsbereichs in die Spielwelt muss folgende Kriterien erfüllen:

1. Die zu stoßende Kugel muss erreichbar sein, also innerhalb des Arbeitsbereichs liegen
2. Es muss möglich sein die Kugel in alle Richtungen zu stoßen

Die naive Lösung bildet den Arbeitsbereich einmalig so ab, dass die weiße Kugel im Zentrum des Arbeitsbereichs liegt. Von dort aus kann die Kugel in alle Richtungen gestoßen werden. Bei der Transformation handelt es sich um eine einfache Skalierung und Translation. Das Problem ist, dass in der Praxis mechanische und anatomische Einschränkungen es nicht erlauben die Kugel beispielsweise von hinten, also gegen die Blickrichtung, zu stoßen. Außerdem möchte man immer in die Richtung sehen, in die gestoßen wird. Eine statische Abbildung des Arbeitsbereichs kommt daher nicht in Frage.

Da der Arbeitsbereich in der Tiefe vergleichsweise klein ist positionieren wir die Kugel bei $(0, 0, -0.4 \cdot \text{Tiefe}/2)$. Die Kamera schaut auf die Kugel und ist „hinter“ dem Arbeitsbereich positioniert, so dass der Arbeitsbereich komplett eingesehen werden kann. Die Blickrichtung der Kamera auf den Arbeitsbereich ist identisch mit der Blickrichtung des Benutzers, wenn er genau vor dem Haptic Device sitzt.

Damit in alle Richtungen gestoßen werden kann, erlauben wir es dem Benutzer den Arbeitsbereich in der Spielwelt zu drehen. Das Rotationszentrum ist die Kugel, die gestoßen werden soll. Wir erlauben zwei Rotationen. Die Rotation um die Y-Achse der Spielwelt und die X-Achse des Arbeitsbereichs. Dadurch kann der Spieler in alle Richtungen schauen und den Arbeitsbereich kippen um die Kugel in einem steileren Winkel zu treffen.

Bewegt man den Stift gegen den linken oder rechten Rand des Arbeitsbereichs wird um die Y-Achse gedreht. Analog dazu wird der Arbeitsbereich gekippt, indem der Stift gegen den oberen oder unteren Rand gedrückt wird. Ähnlich wie bei der Kamerasteuerung wird der Haptikeffekt „Ebene“ für das haptische Feedback verwendet.

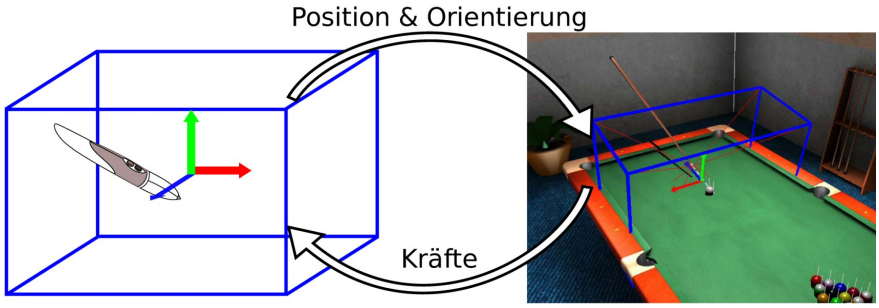


Abbildung 3.7: Transformation des Arbeitsbereichs. Für die Steuerung des Queues muss die Position und die Orientierung des Haptic Device in die Spielwelt transformiert werden. Für das haptische Feedback müssen Kräfte aus der Spielwelt (rechts) in den Arbeitsbereich des Haptic Device (links) transformiert werden.

Abbildung 3.7 (rechts) zeigt den Arbeitsbereich im Ausgangszustand. Abbildung 3.9 zeigt den Arbeitsbereich nachdem er um beide Achsen rotiert wurde. Die Kamera dreht sich immer mit dem Arbeitsbereich mit. Für die beiden Abbildungen wurde die Kamera fixiert.

Stabilisierung

In unseren Versuchen mussten wir feststellen, dass es sehr schwer ist mit dem virtuellen Queue zu stoßen. Das liegt daran, dass es in der Praxis nicht möglich ist eine perfekt gerade Bewegung mit dem Haptic Device zu machen. Außerdem ist es kaum möglich den Stift während eines Stoßes so zu halten, dass sich seine Orientierung nicht verändert. Um die Steuerung einfacher zu machen führen wir eine Kraft ein, die den Queue während eines Stoßes stabilisieren soll.

Möchte der Spieler einen Stoß ausführen, kann er mit der vorderen Taste die Stabilisierung aktivieren. Die Stabilisierung schränkt die Bewegung des virtuellen Queues ein. Der Queue kann nur noch auf einer Gerade bewegt werden. Die Gerade wird durch die Orientierung und die Position des Haptic Device zum Zeitpunkt der Aktivierung definiert. Damit die tatsächliche Position und die stabilisierte Position des Queues nicht zu weit auseinanderdriften wird der Haptikeffekt „Gerade“ angewandt. Zusätzlich zur Stabilisierung der Position wird die Orientierung des Queues eingefroren. Für die Simulation des Stoßes hat dies keine Auswirkungen, da nur die Spitze des Queues als Kugel simuliert wird. Der Spieler hingegen bekommt das Gefühl, dass sein Stoß wie gewollt verläuft.

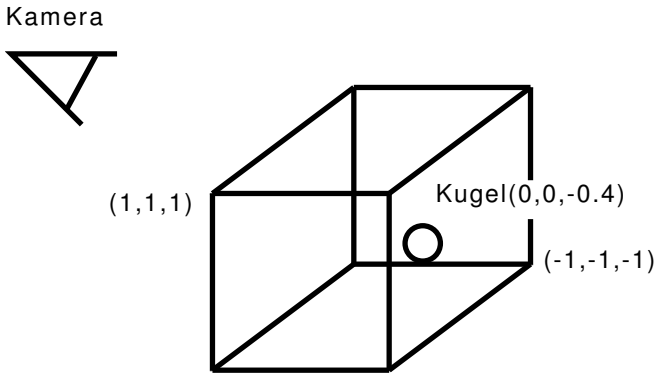


Abbildung 3.8: Der Arbeitsbereich wird so transformiert, dass sich die Kugel etwas entfernt vom Ursprung auf der lokalen negativen Z-Achse befindet. Die Kamera wird so positioniert, dass der Spieler von hinten auf den Arbeitsbereich sehen kann.

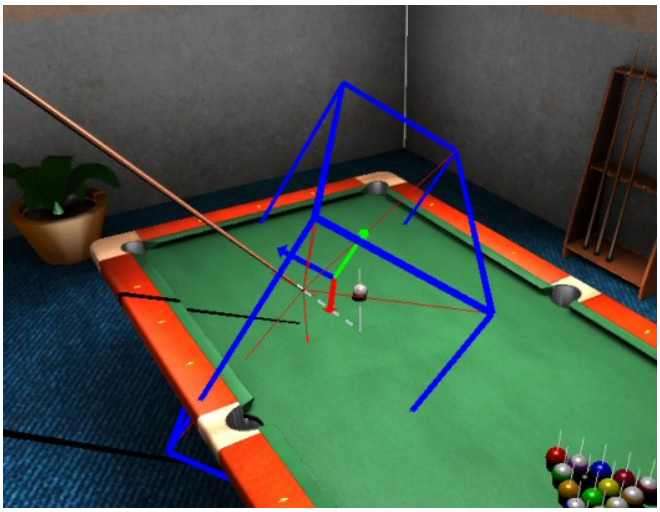


Abbildung 3.9: Arbeitsbereich nach Rotation. Die weiße Kugel kann aus einem steileren Winkel gestoßen werden.

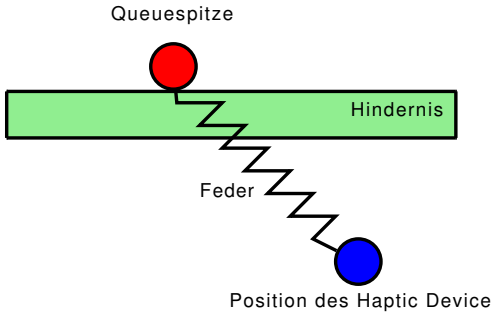


Abbildung 3.10: Die Positionen von Haptic Device und Queuespitze driften im Falle einer Kollision auseinander. Eine Feder verbindet beide Objekte. Die Kraft der Feder wird auf das Haptic Device angewendet.

Kollisionskräfte

Das Haptic Device und die physikalische Queuespitze der Simulation sind über eine Feder gekoppelt. Im kollisionsfreien Zustand sind die Positionen identisch. Tritt eine Kollision auf, weil die Queuespitze auf ein Hindernis trifft, dann werden die beiden Objekte getrennt. Das physikalische Objekt kann dem Haptic Device nicht mehr folgen und die Feder wird gespannt (Abbildung 3.10). Die Kraft, die die Feder ausübt, wird auf das Haptic Device angewendet.

Da die Kraftberechnung in der Simulationschleife erfolgt, wird die Kraft einmal pro Simulationsschritt aktualisiert. Es hat sich gezeigt, dass für länger andauernde Kollisionen, zum Beispiel zwischen Tisch und Queue, das Aktualisierungsintervall kurz genug ist.

Für die Kollision zwischen Kugel und Queue kann das Aktualisierungsintervall zu groß sein und die Kraft tritt etwas zu spät auf. Um die zeitliche Schärfe zu verbessern, wird die zu stoßende Kugel in der Hauptschleife des HapticRenderer modelliert. Die Aktualisierung der Kraft zwischen Kugel und Queue erfolgt dann mit 1000 Hz. Die Kollision kann früher erkannt und die Kraft früher aufgebaut werden. Die Dynamik der Kugel wird durch die Simulation berechnet. Das heißt die Kugelposition wird mit jedem Simulationsschritt aktualisiert.

Da die Simulation mit einer deutlich niedrigeren Frequenz arbeitet, bedeutet eine Veränderung der Kugelposition einen Sprung in der Kraft, die durch den HapticRenderer berechnet wird. Dieser Sprung wird in der Praxis nicht bemerkt, da der Kontakt zwischen Kugel und Queue sehr kurz ist.

Zusammenfassung

Neben der Queuesteuerung mit dem Haptic Device existiert auch eine Steuerung, die mit Maus und Tastatur arbeitet. Wir haben beide Steuerungen bezüglich ihrer Spielbarkeit verglichen. Die Steuerung mit dem Haptic Device steht der Maus- und Tastatursteuerung in diesem Punkt in Nichts nach.

Ein frühes Problem der Queuesteurung waren zu schwache und zu ungenaue Stöße. Die vorgestellte Stabilisierung löst diese Probleme. Die Kugel kann genauso stark und genau gestoßen werden wie mit der Maus.

Die Transformation des Arbeitsbereichs in die Spielwelt und die Positionierung der Kamera funktionieren sehr gut, lässt sich aber mit Sicherheit noch optimieren. Gerade die Positionierung der Kamera bietet viel Spielraum für weitere Experimente.

4 Meshes und Visualisierung

In diesem Kapitel erläutern wir zunächst einige Objekte, die wir in Form von 3D-Meshes modelliert haben und diskutieren ihre Eigenschaften. Anschließend stellt **Julian Bader** in den Abschnitten 4.2 bis 4.4 seine Arbeit am Grafiksystem vor.

4.1 3D-Modelle

Für unsere Billardsimulation musste eine Reihe von 3D-Meshes erstellt werden. *Ogre* verwendet ein eigenes binäres Dateiformat für 3D-Modelle. Wir verwenden dieses Dateiformat sowohl für die Visualisierung als auch für die Physiksimulation.

Wir wollen kurz auf die technischen und künstlerischen Herausforderungen eingehen, denen wir beim Design der 3D-Modelle begegnet sind.

4.1.1 Kugeln

Die ersten Kugeln bestanden aus den Meshes der `PT_SPHERE`-Entität, die im *Ogre SDK* bereits vorhanden ist. Als Textur wurden die entsprechenden Farben der Kugeln als diffuser Farbanteil ausgewählt. Um die Highlights auf der glatten Oberfläche zu simulieren, wurde den Materialeigenschaften zusätzlich Spekularanteile in weißer Farbe hinzugefügt.

In einer späteren Stufe wurde der `PT_SPHERE` durch, in Blender erstellte, *Icospheres*[1] ersetzt. Darüber hinaus erstellten wir für jeden Kugeltyp eine Textur, die zylindrisch um die Kugel gelegt wird. Diese Textur ersetzte den diffusen Anteil der Materialeigenschaften.

Der Übergang von der zweiten Entwicklungsstufe der Kugeln zum jetzigen Modell verlief fließend. Es wurden nach und nach mehr Details, wie z.B. Schatten oder zusätzliche Texturebenen, hinzugefügt. Diese und andere Designelemente werden im Abschnitt 4.3 vorgestellt und im Abschnitt 4.4 gegeneinander aufgetragen. *J.B.*

[1] In Blender entspricht ein *Icosphere* ersten Grades einem Ikosaeder. Bei jedem Übergang zu einem höheren Grad wird die Anzahl der Dreiecke vervierfacht um eine Kugel genauer zu approximieren.

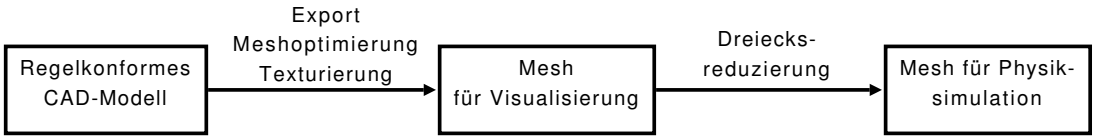


Abbildung 4.1: Designprozess für den Billardtisch: Der Tisch wird in einem CAD-Programm modelliert. Wenn alle Maße stimmen, wird der Tisch in ein Drahtgittermodell exportiert. Dieses wird noch für die Darstellung optimiert (Zum Beispiel durch Entfernen nicht sichtbarer Dreiecke). Der letzte Schritt zum Mesh für die Visualisierung ist das Texturieren. Das Mesh für die Physik basiert auf dem Mesh für die Visualisierung. Für die Physik werden irrelevante Dreiecke entfernt und Details reduziert.

4.1.2 Billardtisch

Der Billardtischs ist neben den Kugeln und dem Queue das wichtigste 3D-Modell in HapticBillard. Er muss die folgenden drei Eigenschaften erfüllen:

1. Viele Details für die Visualisierung
2. So wenig Dreiecke wie möglich, für eine hohe Performance der Physiksimulation
3. Die Dimensionen des Tischs müssen regelkonform sein

Um alle Anforderungen zu erfüllen haben wir uns für den in Abbildung 4.1 dargestellten Designprozess entschieden. Das 3D-Modell musste drei Stufen durchlaufen. In der ersten Stufe wurde der Tisch mit einer CAD-Software modelliert. Als Vorlage haben wir selbstgemachte Fotos und das Regelwerk der Deutschen Billard-Union [9] verwendet. Dem Tischmodell fehlen in dieser Stufe noch Texturen und ein Drahtgittermodell.

In der zweiten Stufe wird der Tisch als Dreiecksmesh exportiert und mit dem 3D-Modellierungsprogramm *Blender* weiterverarbeitet. Das Mesh hat noch eine sehr hohe Anzahl an Faces. Wir optimieren das Mesh, indem wir die Anzahl der Dreiecke an bestimmten Stellen mit dem *Blender PolyReducer* verringern. Außerdem entfernen wir Dreiecke, die nicht sichtbar sind, zum Beispiel die Unterseite der Tischplatte. Die fehlenden Normalen können zum großen Teil automatisch durch die Software berechnet werden. Bei den Texturkoordinaten war sehr viel Handarbeit notwendig um ein gutes Ergebnis zu erzielen. Abbildung 4.2 zeigt den fertig texturierten Tisch und sein Drahtgittermodell.

Die letzte Stufe baut aus dem Mesh der Visualisierung das Mesh für die Simulation. Um das Mesh für die Simulation schneller zu machen werden Dreiecke entfernt, die für die Physik und die haptische Interaktion irrelevant sind. Für die Physik sind nur die Dreiecke der Banden und der Taschen interessant, während für die haptische Interaktion auch Details, wie die Diamanten oder die Rundungen der Tischecken, interessant

Abbildung 4.2: Finales Tischmodell mit Texturierung. Die Drahtgitteransicht macht deutlich, dass ein Großteil der Dreiecke für die Taschen und Banden verwendet wird.

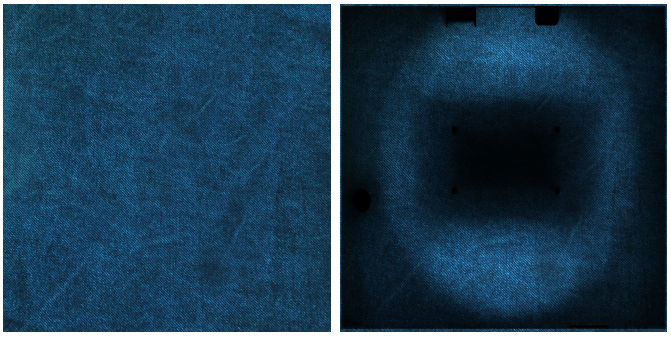
sein können. Hier gilt es also einen Kompromiss zu finden, um größtmögliche Performance zu erreichen ohne die haptische Interaktion zu beeinträchtigen. *B.U.*

4.1.3 Umgebung

Für die Umgebung des Tischs wurde ein Billardzimmer entwickelt, um das Spiel optisch noch interessanter zu machen und eine Atmosphäre aufzubauen. Unser Tisch schwebt nicht einfach in der Luft, sondern er steht einem geheimnisvollen Raum. Es gibt keine Fenster. Vermutlich ist der Raum im Kellergeschoss. Die einzigen Lichtquellen sind die Lampen über dem Billardtisch, die den Raum etwas erhellen. Für uns war es deshalb besonders wichtig diese Beleuchtung glaubhaft zu vermitteln.

Der Raum wurde komplett in *Blender* modelliert und texturiert. Praktisch alle Objekte bis auf den Queue und die Kugeln sind statisch, daher liegt es nahe die Beleuchtung vorauszuberechnen und in einer Lightmap zu speichern. Die Lightmap erlaubt es uns ein hochwertiges Beleuchtungsmodell für unsere Szene zu verwenden.

In unserem Fall teilen sich die einzelnen Modelle des Raums selten Texturen. Um Texturspeicher zu sparen, kombinieren wir deshalb unsere Texturen direkt mit der Lightmap. Die Abbildung 4.3 zeigt die Textur für den Boden. In (a) ist die unbearbeitete Textur zu sehen, (b) zeigt die Textur kombiniert mit Beleuchtungsinformationen. Es ist deutlich der Lichtkegel der Lampen und der Schatten des Tischs zu erkennen.



(a) (b)

Abbildung 4.3: (a) Unbearbeitete Textur für den Boden. (b) Textur kombiniert mit Beleuchtungsinformationen.

Abbildung 4.4 zeigt das komplette Billardzimmer einmal mit den originalen Texturen und mit den bearbeiteten Texturen. Der Unterschied zwischen Abbildung 4.4 (a) und (b) ist deutlich und zeigt wie wichtig eine realistische Beleuchtung für die Spielatmosphäre ist. *B.U.*

4.2 Grafikoptionen

Durch die Grafikoptionen ist es möglich im Spiel bestimmte Eigenschaften von grafischen Objekten zu verändern. Die Motivation ist, die Spielbarkeit der Anwendung auf leistungsschwächeren Systemen sicherzustellen und gleichzeitig zu gewährleisten, dass Besitzer leistungsstärkerer PCs optisch eindrucksvolle Effekte zu sehen bekommen.
Ein weiterer Grund für die Erstellung eines solchen Optionsmenüs war, dass dadurch Benchmarks und Vergleiche verschiedener Grafikeinstellungen einfacher zu realisieren sind, da die Änderungen im laufenden Spiel vorgenommen werden können.

Das Optionssystem besteht aus mehreren Komponenten. Die wichtigsten sind die Grafikeinstellungen und ein „Vector“, der „Structs“ enthält. Diese Struktur enthält das Grafikobjekt und die Eigenschaften, die bei der Erstellung des Objektes als Parameter übergeben wurden. Die Grafikeinstellungen sind ebenfalls durch eine Struktur (`struct`) realisiert, die für jede grafische Eigenschaft Werte speichern kann, die vorher als Konstanten in der Grafikklasse festgelegt wurden. Diese Werte repräsentieren eine bestimmte Detailstufe (z.B. Texturdetail) der Eigenschaft.
Neben diesen Strukturen gibt es noch drei Methoden, die diese Datenstrukturen bzw. deren Elemente verändern können:

(a) Texturen ohne Beleuchtungsinformation.

(b) Texturen mit Beleuchtungsinformation.

Abbildung 4.4: Das Billardzimmer mit zwei verschiedenen Textursätzen. Für beide Aufnahmen wurden die Schattenalgorithmen deaktiviert.

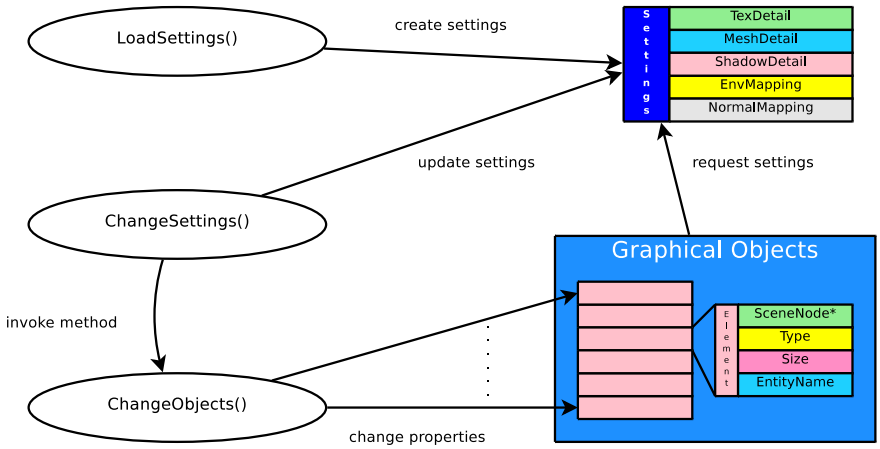


Abbildung 4.5: Grafisches Optionssystem

`LoadSettings()`
Diese Methode lädt alle Grafikoptionen und erstellt daraus die benötigte Struktur, die diese Optionen für die Anwendung bereitstellt. Sie wird nur einmal zu Beginn der Anwendung zur Initialisierung aufgerufen.

`ChangeSettings()`
Sobald die Grafikeinstellungen durch den Benutzer geändert wurden, wird diese Methode aufgerufen. Sie manipuliert die Struktur, in der die Grafikeinstellungen vorgehalten werden. Außerdem ruft sie die Methode `ChangeObjects()` auf.

`ChangeObjects()`
Beim Aufruf dieser Methode werden die Eigenschaften der Grafikobjekte entsprechend den Grafikeinstellungen geändert. Dabei werden werden diese Objekte zuerst zerstört und danach anhand der gespeicherten Eigenschaften mit den neuen Einstellungen neu erzeugt.

4.3 Verwendete Techniken

Eines der grundlegenden Probleme bei der Visualisierung von Computerspielen ist die Balance zwischen Detailreichtum und Performanz. Prinzipiell erfreuen sich die Spieler immer an atemberaubenden Effekten und hochauflösenden Meshes, was allerdings dazu führen kann, dass aufgrund rechenaufwändiger Effekte das Spiel nur mäßig schnell oder gar unspielbar langsam läuft. Lässt sich dieses Problem teilweise durch variable Grafikeinstellungen (vgl. Abschnitt 4.2) beheben, so ist man als Entwickler doch bestrebt auch Besitzern leistungsschwächerer Rechner ohne hochauflösende Meshes bzw. Texturen eine brillante Optik zu bieten.

In diesem Abschnitt wird beschrieben welche Möglichkeiten sich bieten ein 3D-Spiel optisch aufzubereiten. Im darauffolgenden Abschnitt wird diskutiert, wie sich die in folgenden Abschnitt beschriebenen Techniken einsetzen lassen um mit möglichst geringer Performance zufriedenstellende optische Effekte zu erzeugen.

4.3.1 Objektdetailstufen

Die Objektdetailstufe ist direkt mit der Anzahl der Polygone eines Objekts verknüpft. Dabei gilt, je höher die Objektdetailstufe, desto größer ist die Anzahl der Polygone eines Objektes. Um verschiedene Detailstufen eines Objekts zu erhalten, generiert man üblicherweise das Objekt mit unterschiedlicher Anzahl von Dreiecken.
Um die Detailstufe der Billardkugeln zu erzeugen, wurden die Kugeln mit unterschiedlicher Polygonzahl direkt aus „Blender“heraus erstellt. Hierbei wurden drei verschiedene Kugelobjekte mit 5120, 1280 und 320 Dreiecken generiert.

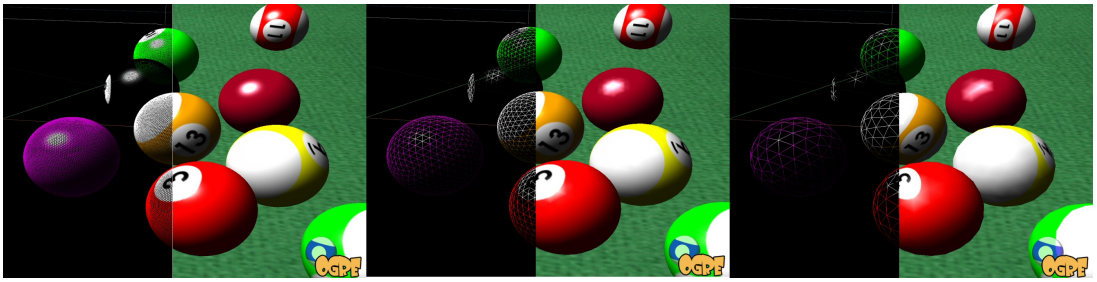


Abbildung 4.6: Kugeln mit hoher, mittlerer und geringer Objektdetailstufe (v.l.n.r.). Während man in der „Wire-Frame“-Ansicht deutlich erkennt, dass sich die Anzahl der Dreiecke unterscheidet, ist dies in der voll gerenderten Ansicht nur anhand des Highlights auf den Kugeln erkennbar.

4.3.2 Texturauflösung

Unter Texturauflösung versteht man die Anzahl der Bildpunkte, die in einer Textur gespeichert werden. Eine höhere Auflösung führt demnach zu einer höheren Anzahl erkennbarer Details. Bei Texturen ist die prinzipielle Vorgehensweise zur Erstellung von Detailstufen, erst eine hochauflösende Textur zu erstellen, dessen Auflösung dann (meist durch bilineare oder trilineare Filterung) auf entsprechende Stufen reduziert wird.
Um die Texturen für die Billardkugeln zu generieren, wurden die ursprünglichen Texturen mit einer Auflösung von 1024 x 1024 gezeichnet. Für eine zweite und dritte niedrigere Detailstufe wurde die Auflösung danach auf einen Wert von 256 x 256 und 64 x 64 reduziert.

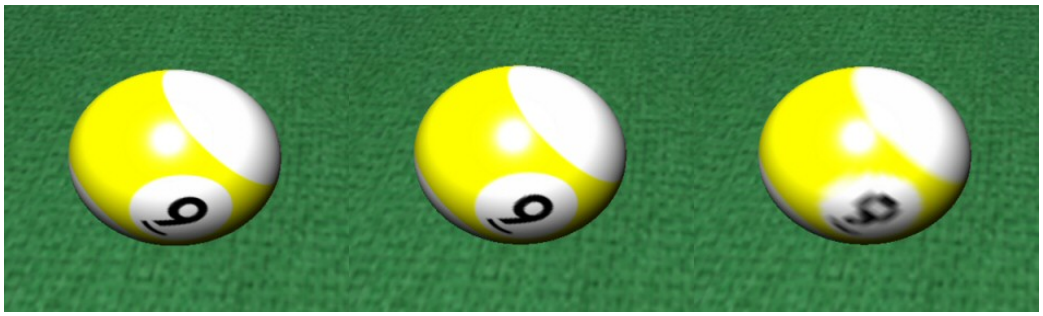


Abbildung 4.7: Kugeln mit hoher, mittlerer und geringer Texturauflösung (v.l.n.r.). Zu erkennen ist, dass die Zahl auf der rechten Kugel verschwommen erscheint. Bei den beiden anderen Kugeln ist kaum ein Unterschied zu erkennen.

4.3.3 Grafische Effekte

Höhere Polygonanzahl bei Objekten bzw. höhere Auflösung bei Texturen sind recht einfache Methoden um den Detailreichtum einer Szene zu erhöhen. Daneben gibt es noch einige „Tricks“, die weitere Details zur Umgebung hinzufügen bzw. vortäuschen. In diesem Unterabschnitt werden die Effekte vorgestellt, die verwendet wurden um die Optik des Spiels zu verbessern.

Schatten

Schatten sind ein wesentliches Element um Spielumgebungen ein realistisches Aussehen zu verleihen, so ist es z.B. möglich eine Tiefeneinschätzung von Objekten zu erhalten. In vielen Spielen (z.B. „Half Life 2“) werden sie auch, in Verbindung mit einer adäquaten Beleuchtung, dazu verwendet eine düstere Atmosphäre zu erzeugen.
Es gibt zwei prinzipiell unterschiedliche Methoden um dynamische Schatten in einer Szene zu generieren (wobei es bei jeder dieser Methoden verschiedene Erweiterungen/Anpassungen gibt):

Shadow Maps: Bei dieser Methode wird die Szene zuerst von der Lichtquelle aus gerendert. Die Tiefeninformationen der gezeichneten Objekte werden dadurch in einem speziellen Puffer gespeichert („Shadow Map“). Im einem zweiten Durchlauf wird die Szene von der Kameraposition aus gezeichnet. Dabei werden die Werte im Z-Buffer transformiert um sie mit denen in der „Shadow Map“zu vergleichen. Ist der Wert im Z-Buffer größer, liegt der Bildpunkt im Schatten und es wird nur mit ambientem Licht gezeichnet. Ansonsten wird der Punkt mit voller Beleuchtung gezeichnet. Diese Methode benötigt demnach zwei Renderdurchläufe.
Der Vorteil dieser Methode ist vor allem die Schnelligkeit. Da es sich hierbei um eine

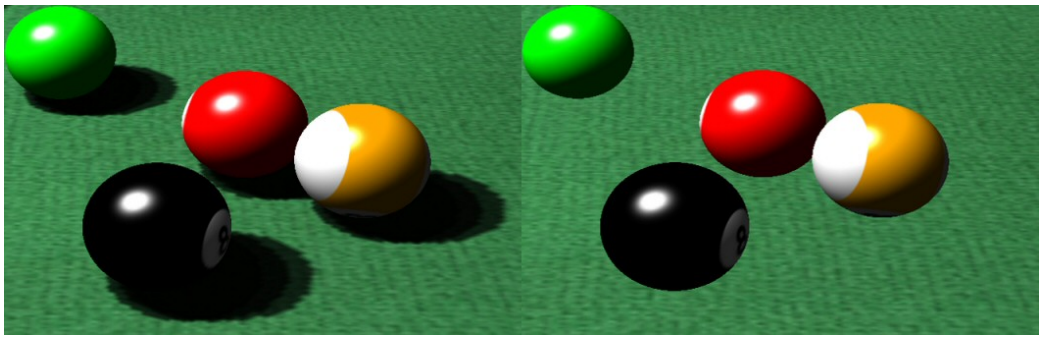


Abbildung 4.8: Bei der Szene, die mit „Shadow Maps“ gerendert wurde (links), sieht man unscharfe Schattenumrisse. Eine Abstufung der Intensität der Schatten ist nur zu erahnen. Die Kugeln in der unschattierten Szene (rechst) scheinen über der Oberfläche zu schweben.

„Image Space“-Methode handelt, ist kein Wissen über die Geometrie der Szene erforderlich. Allerdings können Alias-Effekte (engl. „Aliasing“) auftreten. Diese Methode eignet sich nur gut für weit entfernte Punktlichtquellen.
Durch entsprechende Erweiterungen ist es möglich Alias-Effekte abzumildern bzw. ganz zu vermeiden. Darüber hinaus gibt es Methoden um weiche Schatten („Soft Shadows“) zu erzeugen, indem man die Lichtquelle in Abschnitte aufteilt und für jeden Abschnitt eine eigene „Shadow Map“ erzeugt.
Die Billardanwendung verwendet in der niedrigsten Schattendetailstufe „Shadow Mapping“.

Shadow Volumes: Bei dieser Methode wird für jedes Objekt in der Szene zuerst eine Silhouette berechnet, die die beleuchteten Dreiecke und jene, die im Schatten liegen, trennt. Anhand dessen wird eine Art kegelförmiges Volumen gebildet, indem Strahlen von der Lichtquelle aus geschickt werden, die die Silhouette schneiden. Alles innerhalb dieses Volumens liegt im Schatten.
In einem ersten Renderdurchlauf wird nun die Szene mit ambientem Licht gezeichnet. In den folgenden zwei Durchläufen werden nun Vorder- und Rückseite der „Shadow Volumes“ von der Kameraposition aus gezeichnet, wobei der Stencil-Buffer für jede Vorderseite inkrementiert und für jede Rückseite dekrementiert wird[1] (ZPass). Alle Bildschirmbereiche, bei denen der Stencil-Buffer den Wert 0 enthält liegen im Licht. Die anderen Bereiche liegen im Schatten. Im letzten Renderdurchlauf werden mit Hilfe des Stencil-Buffers alle Bereiche mit voller Beleuchtung neu gezeichnet, die im Licht liegen. Hierbei werden demnacht vier Renderdurchläufe und einige komplexere Berechnungen (zur Ermittlung der Silhouetten) benötigt.

[1] Alternativ kann auch zuerst Rückseite und danach Vorderseite gezeichnet werden, wobei dementsprechen bei jeder Rückseite inkrementiert bzw Vorderseite dekrementiert wird. (ZFail)

Diese Methode erzeugt Schatten ohne störende Alias-Effekte. Auch Selbstschattierung wird dadurch ermöglicht. Sie eignet sich in der Regel für alle Arten von Lichtquellen. Allerdings erfordert diese Methode ein performanteres System um alles flüssig darzustellen, da mehr Renderdurchläufe und komplexe Berechnungen der Silhoutten vorgenommen werden müssen.
Die Billardanwendung verwendet für die beiden höchsten Schattendetailstufe „Shadow Volumes“.

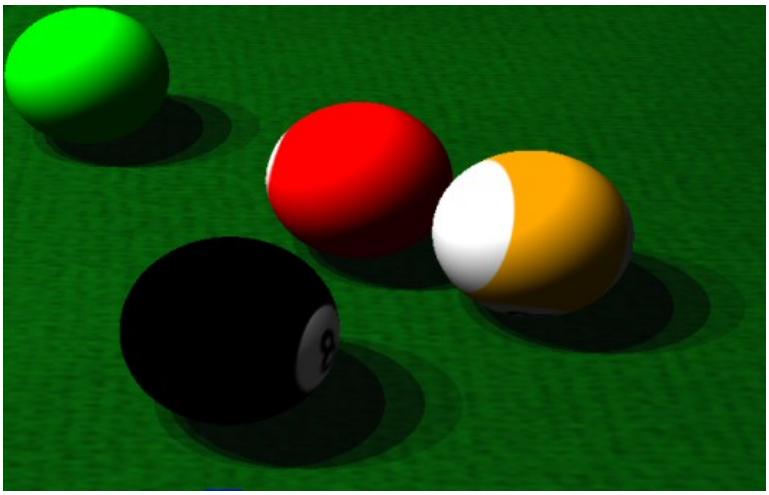


Abbildung 4.9: Shadow Volumes: Die Kugeln werfen Schatten mit einer scharfen Kontur auf die Oberfläche. Durch den Einsatz mehrerer Lichtquellen ist gut erkennbar, dass die Helligkeit der Schattenbereiche variiert, jenachdem wieviele Lichter verdeckt werden.

Environment Mapping

„Environment Mapping“ ist eine Methode um Spiegelungen der Umgebung in glatten Oberflächen vorzutäuschen. Anders als bei „Ray Tracing“-Ansätzen werden nicht die umliegenden Objekte sondern eine zuvor erstellte „Sky Box“ zur Berechnung der Spiegelung herangezogen.
Für die Billardsimulation wurde eine „Sky Box“ des Raumes (inkl. Tisch) erstellt. Diese wird für die Spiegelung der Billardkugeln und der schimmernden Teile des Tisches verwendet.

Normal Mapping

„Normal Mapping“ ist eine Methode einem Objekt mehr Details hinzuzufügen, als sich aus dem eigentlichen Mesh ergeben. Hierzu werden die Oberflächennormalen eines

Abbildung 4.10: Environment Mapping: In diesem Bild ist gut zu erkennen, wie sich der Raum in den Kugeln zu spiegeln scheint.

Objekts anhand einer zuvor berechneten Textur verändert. Da sich die Reflexionswinkel des diffusen und spekularen Lichts dadurch ändern, erscheint das Objekt strukturierter als es in Wirklichkeit ist.
In der Simulation wird dieser Effekt dazu verwendet Kratzer auf der Oberfläche der Kugeln zu simulieren.

4.3.4 Level of Detail

„Level of Detail“ (LOD) ist eine Methode in einer 3D-Welt verschiedene Detailstufen einzuführen. In größeren Spieleumgebungen mit weiter Sicht, ist es möglich Objekte zu sehen, die sehr weit vom eigenen Standpunkt entfernt sind. Diese Objekte mit derselben Detailstufe zu rendern, wie Objekte, die nah sind, führt dazu, dass sehr viele Dreiecke gezeichnet werden und deshalb zu Performanz-Problemen. Das ist unnötig denn die hohen Details sind in großer Entfernung gar nicht erkennbar. Aus diesem Grund geht man dazu über ab einem bestimmten Abstand zur Kamera die Detailstufe von Objekten zu verringern.

Meshes

LOD bei Meshes bedeutet, dass mit zunehmendem Abstand die Komplexität des Meshes verringert wird. Die zu verwendenden Meshes und die Abstände, bei denen die Details verringert werden, werden festgelegt bevor die Szene geladen wird.

In der Billardsimulation wird LOD für Meshes dazu verwendet, die Komplexität der Kugeln zu verkürzen, wenn diese einen höheren Abstand zur Kamera haben.

Texturen

LOD bei Texturen bedeutet, dass die Texturdetails bei zunehmendem Abstand verringert werden. Dies kann zum einen durch „Mip Mapping“ passieren. Zum anderen hat man auch die Möglichkeit bestimmte Texturebenen zu entfernen um so zusätzliche Renderdurchläufe zu vermeiden.
In der Billardanwendung wird LOD in Bezug auf Texturen dazu verwendet, die Detailtextur des Filzes bei größerem Abstand auszublenden. Außerdem wird das „Environment Mapping“ und das „Normal Mapping“ deaktiviert um bei weiter Entfernung zusätzliche Renderdurchläufe einzusparen.

4.4 Effekte vs. Performance

Im letzten Abschnitt wurden Methoden vorgestellt die Spielewelt der Billardsimulation mit Detailreichtum zu versehen. In diesem Abschnitt werden verschiedene Kombinationen dieser Methoden vorgestellt und unter den Aspekten „Spielbarkeit“ und „Optik“ beurteilt.

4.4.1 Benchmarks

Um die Frameraten bei verschiedenen Grafikeinstellungen zu messen wurde das Spiel im Modus „KI gegen KI“ gestartet. Die Kamera wurde so Positioniert, dass der gesamte Tisch sichtbar und das Spielfeld gut zu erkennen war. Änderungen der Grafikeinstellungen bezogen sich ausnahmslos nur auf die Kugeln.
Sämtliche Tests fanden bei einer Bildschirmauflösung von 1440x900 und einer Farbtiefe von 32 Bit statt. VSync und Anti-Aliasing wurden deaktiviert. Als Testsystem diente ein Rechner mit folgender Ausstattung:

- AMD Athlon X2 3800+
- 1,5 GB RAM (DDR 400 / PC3200)
- 1x GeForce 8600GT Grafikadapter (PCI-Express x16) mit 512 MB GDDR3
- Windows XP®Professional

Objektdetails

Die Objektdetails der Kugeln wurden in drei Stufen gemessen. Die Kugelmeshes enthielten 5120, 1280 und 320 Dreiecke. Als Textureinstellung wurde eine mittlere Auflösung von 256x256 gewählt. Gemessen wurde die Framerate pro Sekunde sowohl mit „Shadow Maps“ als auch mit „Shadow Volumes“.

	Frames pro Sekunde	
Dreiecke	Shadow Maps	Shadow Volumes
320	62	55
1280	59	50
5120	56	4

Texturauflösung

Die Auflösung der Kugeltexturen wurden in drei Stufen gemessen. Die Texturauflösung betrug 1024x1204, 256x256 und 64x64. Die Anzahl der Dreiecke für die Kugelmeshes wurde auf 1280 festgelegt. Gemessen wurde die Framerate pro Sekunde ohne Schatten.

Auflösung	Ohne Schatten (FPS)
1024x1024	65
256x256	68
64x64	74

Effekte

Gemessen wurde die Framerate pro Sekunde nach dem Aktivieren bzw. Deaktivieren von „Environment Mapping“ und „Normal Mapping“. Die Objektdetails wurden auf das mittlere und die Texturauflösung auf das hohe Niveau gestellt. Als Schatteneffekte kamen sowohl „Shadow Mapping“ als auch „Shadow Volumes“ zum Einsatz.

	Shadow Maps	Shadow Volumes
Environment Mapping	56	48
Normal Mapping	54	47

4.4.2 Schlussfolgerungen

Auf dem getesteten System gab es wenige Unterschiede bei den verschiedenen Einstellungen bzgl. der Framerate. Generell schneiden sämtliche Einstellungen, die einen erhöhten Rechenaufwand erfordern schlechter ab, als solche, die weniger Rechenleistung benötigen. So resultieren höhere Objektdetails und Texturauflösung in einem leichten Rückgang der Framerate.
Interessant ist der deutliche Leistungsabfall bei der höchsten Objektdetailstufe in Verbindung mit der „Shadow Volume“-Methode. Dies könnte sowohl an der hohen Anzahl der zu zeichnenden Dreiecke liegen (ca. 82.000 bei 16 Kugeln), als auch an der höheren Komplexität bei der Bestimmung der Schattenvolumen.
Die Anzahl der Bilder pro Sekunde ist bei den „Shadow Maps“ leicht höher als bei den „Shadow Volumes“. Dies lässt sich durch die größere Anzahl von Renderdurchläufen und die erhöhte Berechnungskomplexität der Schattenvolumen erklären.
Die niedrigeren Frameraten bei aktiviertem „Environment Mapping“ bzw. „Normal Mapping“ sind vor allem das Resultat von mehr Renderdurchläufen, die für diese Effekte notwendig sind. Desweiteren wird bei beiden Methoden auf den Texturspeicher zugegriffen, was die Darstellungsgeschwindigkeit ebenfalls verlangsamen kann.

Fazit

In der Grafikprogrammierung vor allem in der 3D-Visualisierung bedeutet mehr Dreiecke und Texturdetails nicht notwendigerweise mehr Gewinn bzw. Optik. Zwar zeigen die Messungen, dass mit steigenden Objektdetails oder höherer Texturauflösung die Performance kaum abnimmt. Allerdings ist auch deutlich zu erkennen (vgl. Abb. 4.6 und 4.7), dass ab einer bestimmten Stufe kaum noch Verbesserungen wahrnehmbar sind. Aufgrund dieser Beobachtungen scheint es sinnvoller die Einstellungen auf einem niedrigeren Niveau zu belassen und die so gesparte Rechenleistung für andere Effekte zu verwenden.
Vergleicht man die Ergebnisse von „Shadow Volumes“ und „Shadow Maps“ (Abb. 4.9 und 4.8), so erkennt man deutliche Unterschiede. Während die eine Methode deutlich unscharfe Schatten und Aliaseffekte hervorbringt, produziert die andere gestochen scharfe, abgestufte Schattierungen.
Bei schwächeren Systemen besteht bei den Schatten ein großes Potential einen Performance-Gewinn zu erzielen. Vergleicht man die Frameraten von Volumenschatten und deaktivierten Schatten, ergibt sich eine Differenz von fast 20 FPS.
Verfügt das System über genügend Rechenleistung empfiehlt es sich in jedem Fall Effekte wie „Environment Mapping“ oder „Normal Mapping“ zu aktivieren. Die Optik der Simulation kann dadurch wesentlich verbessert werden (vgl. Abb. 4.10).

5 Periphere Komponenten

Abgesehen von den zentralen Themen Steuerung, Physik und Grafik gehören zu unserer Simulation *HapticBillard* mehrere Komponenten die sich nicht in die genannten einreihen lassen, aber erwähnenswert und elementar für die Funktion sind. Dabei handelt es sich um das Eventsystem zum Austausch von Nachrichten, die Implementation der Spielregeln, das Soundsystem und den KI-Gegner. Sie wurden von **Johannes Wendeberg** entwickelt, wenn nicht anders angegeben, und werden im Folgenden vorgestellt.

5.1 Eventsystem

Während des Spielablaufs ist es notwendig, Informationen über das aktuelle Spielgeschehen bekanntzumachen. Wir haben ein postähnliches System, den *EventManager*, entworfen, welcher eingehende Nachrichten sammelt und regelmäßig an alle eingeschriebenen Empfänger weiterleitet, was auch künstlich verzögert passieren kann. Die Kollision eines Balles mit einem anderen etwa, ist sowohl für die Spielregel-Verwaltung, als auch für die Sound-Engine interessant. Sie werden durch das Eventsystem über solche Vorgänge benachrichtigt. Auch der KI-Spieler erhält so Informationen, um in seinem Status voranzuschreiten.

Die Nachrichten besitzen einen Stempel, welcher Art sie sind. Angehängt werden können verschiedene Parameter beliebiger Anzahl, z.B. Bälle, Vektoren, Textstrings und Fließkommazahlen. Eine Nachricht könnte etwa so aussehen:

(`WHITE_HIT_FULLBALL`, Ball* a, Ball* b, Vector3 position, float strength)

Sie überträgt die Nachricht, dass die weiße Kugel eine volle Kugel getroffen hat, die beiden involvierten Bälle, den Ort der Kollision und ihre Stärke.

Die Registrierung als Empfänger von Nachrichten geschieht automatisch durch Ableitung einer Subklasse von der Klasse `AbstractEventListener`. Anschließend wird die Funktion `collectEvent(const Event& event)` redefiniert, wo der eingehende Event weiterverarbeitet werden kann. *J. W.*

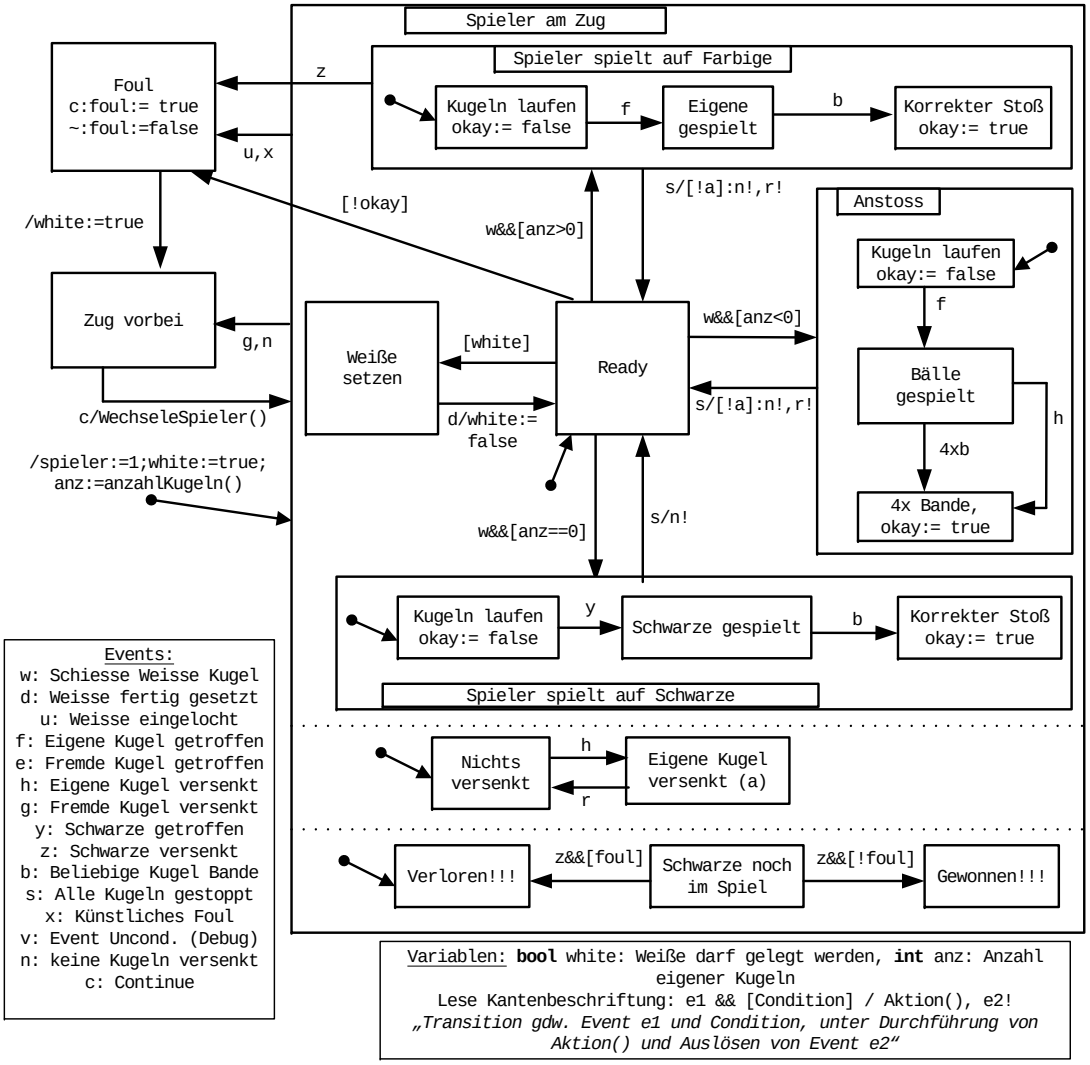


Abbildung 5.1: Statecharts-Diagramm der Spielregeln, die wir mittels *Boost Statechart* in C++-Code umsetzten. Auffällig in State `Spieler am Zug` sind die parallelen AND-States gleichzeitiger Überwachungen (getrennt durch gestrichelte Linien) und die sequentielle Substates, wenn ein Spieler am Zug ist.

5.2 Spielregeln

Für die weit bekannte Spielvariante *Acht-Ball* bzw. *8-Ball* entwarfen wir eine Implementation des Regelwerks. Wir versuchen allerdings nicht, das komplette Regelwerk des offiziellen Billard abzubilden. So gibt es in *HapticBillard* weder die Möglichkeit, Taschen anzusagen, noch einen „Safety Shot“ (nach dem Stoß ist Zug in jedem Fall beendet). Vielmehr verwenden wir einen Auszug, der sinnvolles Spielen ermöglicht.

Acht-Ball wird von zwei Spielern mit einer weißen Kugel, 14 farbigen Spielkugeln und einer schwarzen Kugel gespielt. Die farbigen Kugeln werden in 7 „volle“ und 7 „halbe“ Kugeln aufgeteilt. Ein Spieler übernimmt nach dem ersten Einlochen einer farbigen Kugel nach dem Anstoß diese Farbe. Ziel des Spiels ist, die schwarze Kugel zu versenken, was erst erfolgen darf, nachdem alle Kugeln der eigenen Farbe versenkt worden sind. Jeder Stoß muss dabei korrekt ausgeführt werden, andernfalls wird ein „Foul“ gewertet und der andere Spieler ist an der Reihe. Auch wenn keine eigene Kugel versenkt werden konnte, ist der Zug beendet. Die kompletten Regeln können in vielen Quellen [10] nachgelesen werden.

Die Implementation dieser Spielregeln wurde von uns im Vorfeld heiß diskutiert. Wir waren uns einig, dass wir nicht wie viele andere Billard-Simulationen [3] vorgehen wollen und Kolonnen von `if`-Statements in einer Funktion mit gigantischem Ausmaß zu schachteln. Eine Alternative wäre die Verwendung einer Scripting-Engine gewesen, z.B. *Lua*[1]. Das hielten wir aber für übertrieben. Einer der Autoren hatte Erfahrung mit *Statecharts* [11], welche 1984 von David Harel im Rahmen seiner Dissertation entworfen wurden. Diese eventbasierte Erweiterung von hierarchischen Zustandsautomaten kann sowohl mit Variablen umgehen, als auch mit parallelen Sub-Zuständen eines Zustands. Wir sahen grundlegende Parallelen zwischen den Zuständen dieser Statecharts und unseren verschiedenen Modi, etwa Anstoß, Spiel auf eine Farbe, Spiel auf die Schwarze, oder im zeitlichen Ablauf eines einzelnen Spielzuges. Wir waren der Ansicht, dies ließe sich gut abbilden und beschlossen, Statecharts eine Chance zu geben.

Die fundamentale Idee war, ein graphisches Statechart mit der kompletten Abbildung des Regelwerks zu entwerfen und anschließend in C++-Code einer existierenden Statecharts-Library zu transformieren. Die Kommunikation, die in Statecharts eventbasiert abläuft, wollten auch wir durch Events darstellen. Aus diesem Grund hatten wir den *EventManager* entworfen, der Spielevents 1:1 in Statecharts-Events übersetzt und umgekehrt Events auslöst, sobald bestimmte States betreten werden. Ein Diagramm unserer Statecharts ist in Abb. 5.1 zu sehen.

Auf dem Markt existieren verschiedene Statecharts-Libraries die allesamt Open-Source und damit frei verfügbar sind. Wir betrachteten zunächst mehrere, um herauszufinden, welche geeignet sind.

[1] Lua, `http://www.lua.org/`

- **A Lightweight Implementation of UML Statecharts in C++:** Diese kleine, arraybasierte Stateklasse war einfach zu bedienen und funktionierte in Tests. Leider unterstützt sie keine parallelen AND-States, eine Voraussetzung für unser Projekt. [12]
- **CHSM:** Die *Concurrent Hierarchical Finite State Machine* [13] sah vielversprechend aus. Doch der Autor war nicht in der Lage, den Quellcode unter Windows zu kompilieren. Eine Vielzahl von portierten Unix-Libraries war die abschreckende Systemvoraussetzung.
- **Boost Statecharts:** Die Statecharts-Implementation der bekannten C++-Library *Boost* [14] besitzt eine massiv `template`-basierte Struktur. Die States werden durch Typisierung einer `template`-Basisklasse erzeugt. Trotz der daraus resultierenden, äußerst unbequemen Syntax einigten wir uns auf *Boost Statecharts*, nicht zuletzt, weil *Boost* in Programmiererkreisen einen guten Ruf geniest. Wir verwenden die Version 1.36.0.

Die Umsetzung des graphischen Charts in C++-States gelang nach kurzer Einarbeitung in die exotische Syntax überraschend einfach und zügig. Zunächst entwarfen wir eine von der Billardsoftware unabhängige Konsolenapplikation, die einzelne Events durch Tastaturbefehl entgegennehmen konnte und sogar ganze Eventsequenzen per Copy & Paste aus einer Vorlagendatei verarbeitete. Dadurch wurden uns Testläufe mit der Dauer von mehreren Runden und beliebige künstliche Szenarien möglich, um die Korrektheit der Implementation zu untersuchen.

Doch zunächst konnten wir ständig nur die Nicht-Korrektheit nachweisen. Trotz mehrfacher Überarbeitung des ursprünglichen Diagramms, kristallisierte sich nach und nach heraus, wie schwierig es war, die Charts so zu gestalten, dass das Billard-Regelwerk tatsächlich mit allen Spezialfällen abgebildet war. Hinzu kam eine Schwierigkeit mit der Boost-Library, bei der auftretende Events nur an eine anstehende Transition übermittelt wurden, wenn mehrere Transitionen gleichzeitig auf dasselbe Event warteten. Wir konnten leider nicht feststellen, ob das so gedacht war, oder ob es sich um eine Konzeptschwäche handelt. Der Workaround, nur je eine Transition auf einen Event warten zu lassen und die Events der anderen Transitionen umzubenennen und während der ersten Transition künstlich zu erzeugen, verschaffte Abhilfe, erhöhte aber die Zahl der Events. Auch die exotische Syntax forderte ihren Tribut. Der State *StateFoul* wird definiert durch

```
class StateFoul : public sc::state<StateFoul, StateGlobalState>
```

Er ist also eine Ableitung von einer Templateklasse mit dem Namen als Template-Argument. Wird der State zur Laufzeit betreten, so wird eine Instanz erzeugt und der Konstruktor ausgeführt. Wird er wieder verlassen, so wird die Instanz zerstört und der Destruktor ausgeführt. Dazu muss man wissen, dass Template-Code zur Kompilierzeit

in Maschinencode umgesetzt wird. Jeder State ist also eine eigene Klasse mit eigenem Code, selbst wenn 100 States genau dasselbe tun. Die Kompilierzeit und der benötigte Speicherverbrauch wachsen dadurch mit jedem State. Die Größe unserer ausführbaren `exe`-Datei wächst von ca. 2 MB auf über 4 MB, obwohl die Stateklassen weniger als 5 % des Quellcodes ausmachen.

Letztendlich war es möglich, die C++-Version des Statecharts soweit anzupassen, dass korrektes 8-Ball spielbar ist. Von der definitionsgetreuen, puristischen Auslegung von Statecharts ist leider nicht viel übrig geblieben. Wir verwenden Hilfsvariablen und bedingte Transitionen, falls wir in bestimmten States sind. Das ist grafisch wiederum schwer darstellbar, somit haben wir unser ursprüngliches Ziel, ein übersichtliches Spielregelkonzept zu implementieren, verfehlt. Die Spielregeln mittels Statecharts sind das einzige Softwaremodul in *HapticBillard*, bei dem wir froh sind, dass es funktioniert, aber es nicht noch einmal so machen würden. *J. W.*

5.3 Soundsystem

Was wäre ein Billardspiel ohne das vertraute Klicken der Kugeln und dem satten Geräusch, wenn eine Kugel in eine Tasche fällt? Basierend auf der Sound-Library *FMOD* [15] haben wir eine Sound-Engine implementiert, mit der positionale Geräusche abhängig vom Spielgeschehen und der Position des Zuhörers erzeugt werden. Vom *EventManager* erhält die Engine eine Nachricht, welcher Vorgang passiert ist, z.B. *Weiße trifft Ball*, und wo er stattgefunden hat. Die Position des Vorgangs, der zugehörige Soundclip und die Position und Ausrichtung des Zuhörers, welche der Kameraposition und -ausrichtung entspricht, werden FMOD übergeben. Dieses berechnet aus der dreidimensionalen Darstellung eine Stereoausgabe, die in räumlich korrekter Position und Entfernung den Klang wiedergibt.

Die Sounds haben wir mit einem Soundrecorder an einem realen Billardtisch aufgenommen und anschließend nachbearbeitet. Für jede Soundklasse haben wir mehrere verschiedene Klänge erstellt, und wählen sie zufällig aus, so dass zwei gleiche Vorgänge immer anders klingen. An dieser Stelle gilt unser Dank dem Billardcenter *Shooters* in Gundelfingen für die freundliche Genehmigung zur Aufnahme von Geräuschen, Bildern und Videos. *J. W.*

5.4 KI-Spieler

Ein Element der Software, das über den gesteckten Rahmen etwas hinausgeht, ist der simulierte Gegner, auch *Bot* genannt, der im Spielmodus *Eightball* gegen den menschlichen Spieler antritt. Die Ziele eines künstlichen Gegners für Billard sind vergleichsweise einfach zu formulieren:

- Versuche, alle eigenen Kugeln zu versenken.
- Versuche, dabei kein Foul zu begehen.

Mit diesen beiden Paradigmen lässt sich bereits die Spielstruktur der meisten menschlichen Spieler nachbilden. Erst fortgeschrittene Spieler versuchen, Kugeln so zu spielen, dass die Weiße nach dem Stoß an einem bestimmten Ort zur Ruhe kommt, damit sie für den anschließenden Stoß gut platziert ist. Auch Techniken wie Vorwärts-, Rückwärts- oder seitlicher Drall kommt nur bei ihnen zum Einsatz. Um die Ergebnisse solcher Planungen zu berechnen, ist eine große Spielerfahrung notwendig. Man muss wissen, wie sich die Kugeln bewegen werden.

Das könnte in der Implementation durch Vorausrechnen der Spielsimulation geschehen, was auch leicht als „schummeln“ interpretiert werden kann, der Computergegner könnte so in die Zukunft sehen. Das soll hier nicht geschehen. Die „Intelligenz“ des Gegners kann ausschließlich das Spielfeld beobachten, sie erhält 2D-Koordinaten des laufenden Spiels und kann Vektormathematik darauf ausführen.

Ein Statussystem lässt den Spieler im Laufe eines Spielzugs voranschreiten:

$$IDLE \rightarrow LAZING \rightarrow AIMING \rightarrow SHOOTING \rightarrow ANALYZING \rightarrow DONE$$

Bevor ein Zug ausgeführt wird, startet der Spieler mit einer kurzen Phase des Nichtstuns, was ein Überlegen andeutet und etwas Hektik aus dem Spiel nimmt. Der Zielvorgang wird einmal berechnet und dann kontinuierlich ausgeführt, und dabei der Queue bewegt. Auch das Schießen wird kontinuierlich ausgeführt, indem der Queue nach vorne bewegt wird, bis die weiße Kugel getroffen wurde. Danach zieht der Spieler den Queue wieder zurück und geht in den Abschlußzustand. Die interessante Phase ist dabei der Zielvorgang *AIMING* in dem die Wegplanung stattfindet und das Ziel berechnet wird.

5.4.1 Wegplanung

Was dem menschlichen Spieler einfach gelingt, nämlich zu untersuchen, welche Kugeln anspielbar sind, und welche durch andere Kugeln blockiert sind, war hier etwas schwerer zu implementieren. Gelöst wurde es durch eine Funktion, die für die Strecke zwischen zwei Kugeln A und B überprüft, ob eine beliebige andere Kugel P in der Nähe liegt und den Weg blockiert. Wir rechnen das Skalarprodukt der Differenzvektoren:

$$k = (\mathbf{p} - \mathbf{a}) \cdot \frac{(\mathbf{b} - \mathbf{a})}{|(\mathbf{b} - \mathbf{a})|}$$

Wir bekommen die Aussage, wenn $0 < k < |\mathbf{b} - \mathbf{a}|$, dann liegt die Kugel P, auf die Strecke $\overline{AB}$ projiziert, zwischen A und B. Wenn die Länge der Linearkombination $|\mathbf{a} + k\frac{(\mathbf{b}-\mathbf{a})}{|(\mathbf{b}-\mathbf{a})|} - \mathbf{p}|$ kleiner als die Summe der Radien von A und P ist, dann liegt

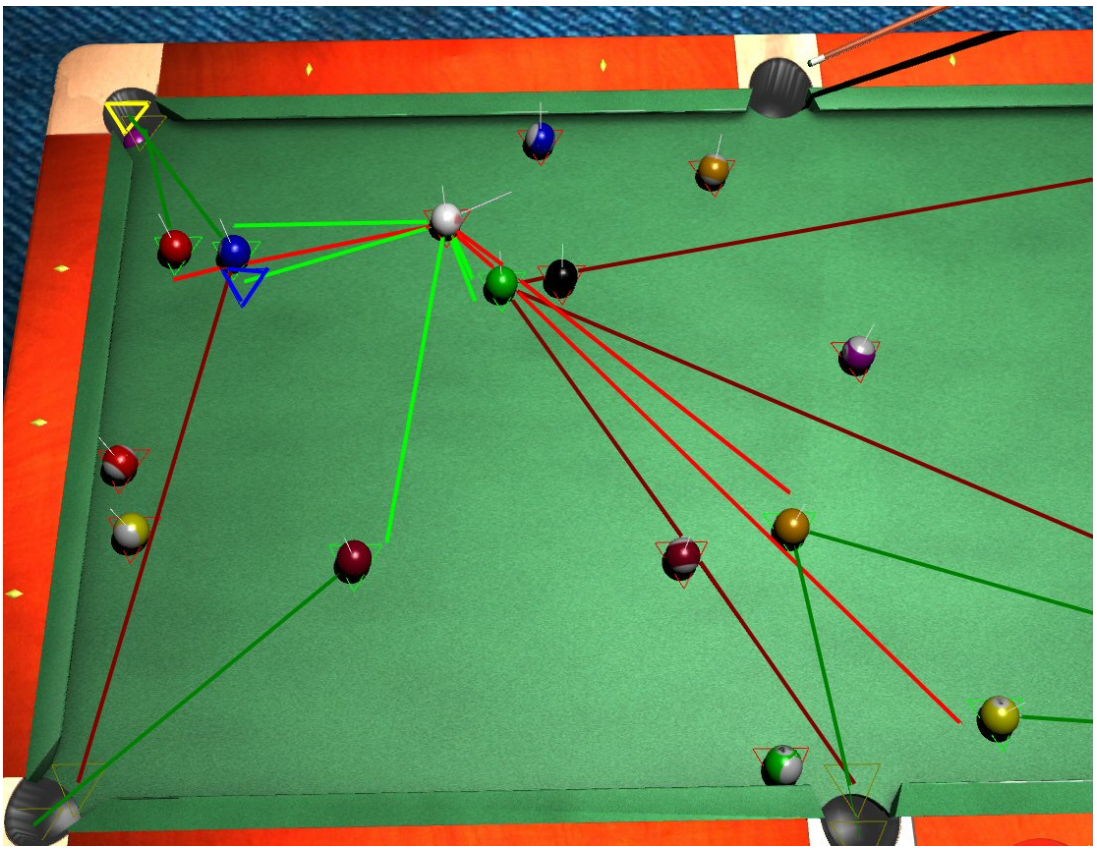


Abbildung 5.2: Beispiel für eine Wegplanung des KI-Spielers. Die vollen Kugeln gehören ihm, sie sind mit grünen Dreiecken markiert. Freie Pfade zu den Kugeln sind grün, blockierte sind rot. Die Pfade von den Kugeln zur Tasche sind dunkler dargestellt. Der Spieler hat sich für die blaue Kugel entschieden (blaues Dreieck). *(Schaubild der Software entnommen, Stärke der Linien zur Verdeutlichung nachbearbeitet.)*

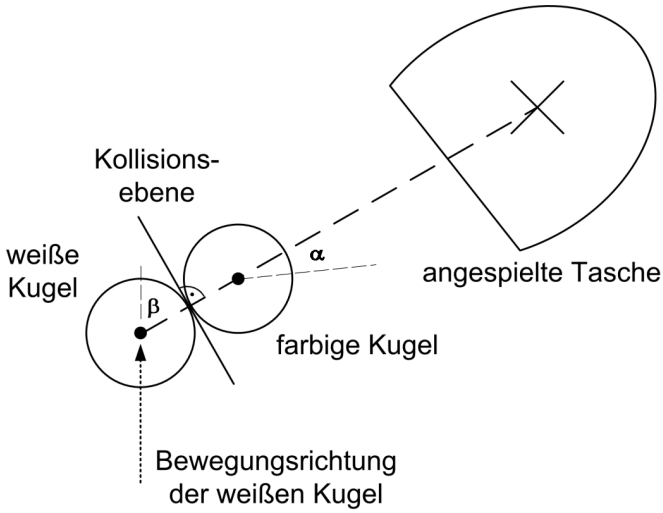


Abbildung 5.3: Versetztes Anspielen einer farbigen Kugel, so dass sie versenkt wird. Die weiße Kugel, die farbige Kugel und die Tasche bilden eine Linie. β ist der Schusswinkel zwischen der Bewegungsrichtung der weißen Kugel und der Zielrichtung, α der Winkelfehler zur korrekten Richtung der farbigen Kugel. Es gilt $\beta \approx f\alpha$.

P im Weg und die Strecke $\overline{AB}$ ist tatsächlich blockiert. Dieser Test wird für alle eigenen Kugeln von der weißen Kugel bis zur entsprechenden Kugel und von dort zu jeder spielbaren Tasche vorgenommen. Aus dieser Menge spielbarer Bälle wird nun der günstigste Ball berechnet. Ein Scoringsystem gewichtet mehrere Faktoren: die Distanz, den Schusswinkel, den Einfallswinkel zur Tasche, eine Penalty-Punktzahl, wenn die weiße Kugel zu nahe an der farbigen liegt und eine Penalty-Punktzahl falls der Winkel zu gering ist (weil dann die Weiße der farbigen Kugel in die Tasche folgen könnte). Die Kugel-Tasche-Kombination mit der kleinsten Punktzahl wird ausgewählt. In Abb. 5.2 ist ein Beispiel für eine fertige Wegplanung zu sehen.

5.4.2 Zielen

Das Versenken einer farbigen Kugel geschieht durch versetztes Anspielen mit der weißen Kugel. Dabei wird von der Faustregel ausgegangen, dass die weiße Kugel, die farbige Kugel und die angespielte Tasche zum Zeitpunkt der Kollision eine Linie bilden (vgl. Abb. 5.3). Das stimmt in der Praxis nicht ganz. Zum einen muss winkelabhängig um einen Faktor f steiler gezielt werden, um die Rotationsdifferenz auszugleichen, zum anderen musste anfangs aus unerklärlichen Gründen wieder gerader gezielt werden. Die Gründe hierfür sind inzwischen gefunden. Bei der Lösung des Problems war der Lernalgorithmus des KI-Spielers eine große Hilfe (vgl. Abschnitt 5.4.3).

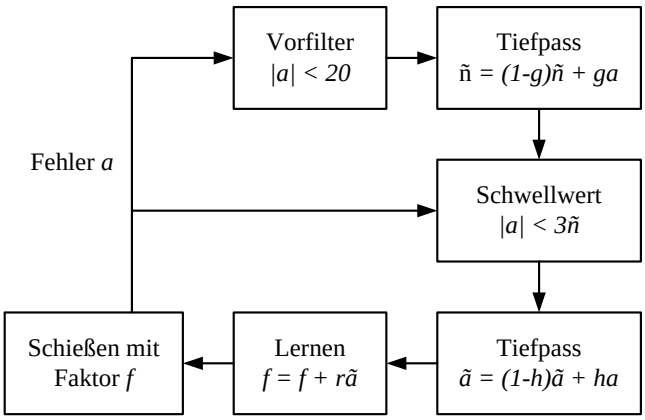


Abbildung 5.4: Der aus Schüssen resultierende Winkelfehler α wird nach Vorfilterung zur Schwellwertbestimmung verwendet. Dieser Schwellwert entscheidet, ob α in den gemittelten Fehler $\tilde{\alpha}$ aufgenommen wird. Aus $\tilde{\alpha}$ und der Lernrate r wird der Rotationsfaktor f bestimmt. Geht $\tilde{\alpha}$ gegen Null, so stabilisiert sich f.

Bei einfachen Bällen, deren Winkel nicht zu steil ist und die nicht an einer Bande liegen, konnte eine Trefferquote von ca. 60–70 % für gezielt geschossene Bälle (gemittelt über 10 Schüsse) erreicht werden. Mit gezielten Bällen sind alle Schüsse auf eine Tasche gemeint, auch Bälle, die direkt an der Bande liegen, oder Schüsse, bei denen die weiße oder eine farbige Kugel durch die Bande blockiert wird. Die genannten 60–70 % sind im Vergleich zu einem menschlichen Spieler ziemlich stark.

5.4.3 Lernalgorithmus

Das unter Billardspielern weit bekannte Paradigma, den Anspielpunkt, die farbige Kugel und die angespielte Tasche auf einer Linie zu vereinen, ist in der Praxis nicht ganz korrekt. Der Spieler muss steiler zielen, denn bei der Kollision wird eine Rotationsdifferenz ausgeglichen, die die Laufrichtung der farbigen Kugel verfälscht. Die Verfälschung ist zudem von der Schussstärke abhängig. Offenbar gilt $\beta \approx f\alpha$ mit konstantem Faktor f (vgl. Abb. 5.3). Diesem systematischen Fehler war erst durch den KI-Spieler, der einen Ball exakt anspielen kann, auf die Spur zu kommen. Es stellte sich die Frage, wie stark der KI-Spieler tatsächlich steiler zielen muss, und ob er den systematischen Fehler ausgleichen kann. Aus diesem Grund wurde ein Lernalgorithmus implementiert, der den Faktor f automatisch bestimmt. Der gespielte Winkelfehler α wird kurz nach dem Stoß abgegriffen und fließt als Feedback in die zukünftigen Stöße ein. Ist f zu niedrig, spielt der Spieler zu gerade, ist f zu hoch, spielt er einen zu steilen Winkel. Mit diesem Wissen lässt sich der Faktor optimieren. Das Schema des Lernalgorithmus ist in Abb. 5.4 dargestellt.

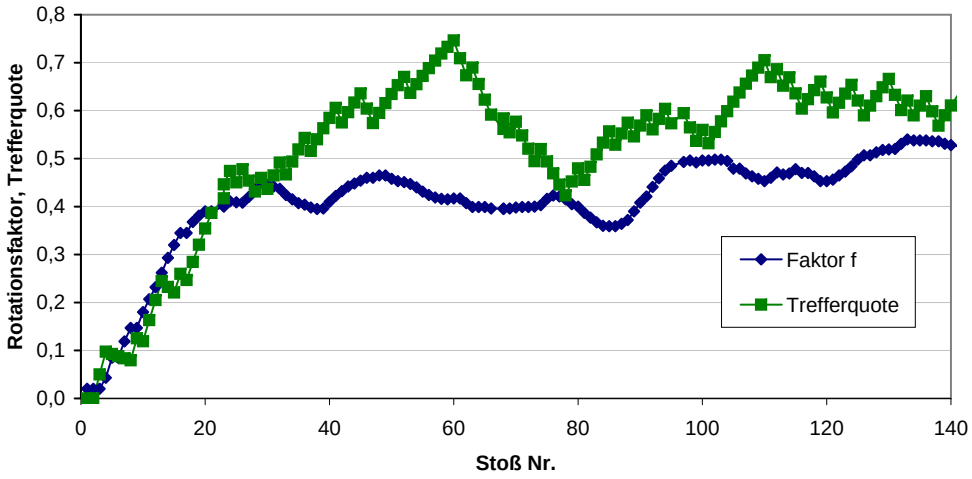


Abbildung 5.5: Bereits nach 20 Zügen stabilisiert sich f von einem Startwert von 0,0 aus bei 0,45 (mit Fehler von $\pm 0{,}1$). Sobald der Vorfaktor stimmt, steigt auch die Trefferquote (wegen Mittelung etwas verzögert) auf Werte von bis zu 70 % an.

Das Abgreifen der resultierenden Ballrichtung nach dem Stoß ist fehlerbehaftet. Liegt der Ball etwa an der Bande, so bewegt er sich in eine für den Lernalgorithmus unerwartete Richtung und ein massiver Fehler wird aufgezeichnet. Deshalb war eine ausgeklügelte Vorverarbeitung der Messwerte notwendig. Wurden diese Fehlerwerte aussortiert, so zeigt der Algorithmus eine gute Tendenz gegen den Zielwert nach spätestens 20 Spielzügen, selbst mit einem schlechten Startwert von $f = 0{,}0$. Der Algorithmus stabilisiert sich bei einem Wert von ca. $f = 0{,}45$, was einiges steiler ist als nach der Faustregel von Abschnitt 5.4.2.

Mit dem KI-Spieler wurde eine weitere Verfälschung festgestellt, indem die Reibung zwischen Bällen zwischenzeitlich deaktiviert wurde. Nachdem das zu gerade Spiel weggefallen war, spielte er zu steil, der Ball musste also gerader angespielt werden als nach der Faustregel. Der Grund für das zu steile Reflektieren der farbigen Kugel wurde bald gefunden: Die diskrete Simulation lässt ein Eindringen der Kugeln ineinander zu. Wird die Kollision erkannt, so haben sich die Kugeln bereits illegalerweise ineinander bewegt. Falls die Kollision exzentrisch war, so ist die Kollisionsnormale zum Zeitpunkt der Detektion verschoben und zwar verschieden stark, je nachdem wieviel Zeit $0 < \Delta t < h$ seit der Kollision vergangen ist. Nachdem wir den exakten Ort der Kollision anhand der Geschwindigkeitsvektoren $\mathbf{v}$ zurückgerechnet hatten und von dort aus die Kollisionsnormale berechneten, verschwand das Problem. Der KI-Spieler näherte sich nun $f = 0{,}0$. Der Mittelwert des bereinigten Fehlers E_c, also nach Ausschluss aller Fehlerwerte die über dem jeweiligen Schwellwert lagen, sank nun von 1,4 auf 0,6, nach wie vor ohne Reibungsmodell. Das bedeutet, dass nicht nur der systematische Fehler abnahm (indem der KI-Spieler f lernt), sondern sich auch der zufällige Fehler, den die unkal-

kulierbare Normalenverschiebung verursachte, verringerte. Anschließend erreichte der KI-Spieler Trefferquoten von traumhaften 80 %.

Leider mussten wir aus Gründen des Realismus das Reibungsmodell wieder aktivieren, alles andere wäre physikalisch nicht korrekt. Es muss also um den Faktor $f = 0{,}45$ steiler gespielt werden und der bereinigte Fehler ist nun höher mit $E_c = 2{,}1$, weil der Winkelfehler von der Schussstärke abhängig ist. Der KI-Spieler spielt nun etwas schlechter und erreicht nur noch 60–70 % Trefferquote, weil er die variierende Schussstärke nicht kompensieren kann, vgl. Abb. 5.5.

In der jetzigen Version wird der Lernalgorithmus nicht mehr benötigt, da wir den Faktor f fest vorgeben können und er sich im Laufe des Spiels nicht mehr verändert. Als Technikdemonstration war er aber sehr interessant zu entwickeln und bei der Fehleranalyse bewies er seine Daseinsberechtigung.

5.4.4 Resultat

Mit dem implementierten Verhaltensmuster konnte ein fähiger Computergegner entworfen werden, ohne die Simulation ressoucenintensiv und spieltechnisch fragwürdig in die Zukunft zu rechnen. Der Bot spielt die Kugeln, die man als Mensch spielen würde, und versenkt sie häufig. Die variable Schussstärke verstärkt den realistischen Eindruck weiter. Seine Spielstärke siedelt sich irgendwo zwischen dem präzisen Laienspieler und dem fortgeschrittenen, planenden Spieler an. Bei Testspielen des Programmierers gegen ihn gewinnt mal der Mensch, mal der Computer.

Interessant war, dass die für den Menschen einfache Aufgabe der Wegplanung komplizierter zu entwerfen war, wogegen der für den Menschen schwierige Vorgang des exakten Zielens und Schießens leicht zu implementieren war. Das Zielen ist, wie das Rechnen mit einem Taschenrechner, eine für den Menschen komplexe Aufgabe, die einem Computer sehr leicht fällt. *J. W.*

6 Zusammenfassung

Das Projekt *HapticBillard* war für alle Beteiligten über einen Zeitraum von sechs Monaten gleichzeitig ein Hochschulprojekt, als auch ein herausforderndes Freizeitprojekt. Die Software umfasst zum Stand vom April 2009 einen Umfang von ca. 22.000 Zeilen Code. Es wurden mehr als 48.000 Zeilen Code in dieser Zeit verändert. Zusätzlich enthält *HapticBillard* Grafiken und 3D-Objekte, die allesamt selbst entworfen worden sind.

Das Ergebnis unserer Arbeit kann sich sehen lassen. Die drei Komponenten Physik, Inputsystem und Grafik sind trotz mancher zwischenzeitlicher Schwierigkeiten nicht nur soweit gediehen, dass sie die gestellten Anforderungen erfüllen, sondern sie können in manchen Punkten sogar über bestehende Ansätze herausragen.

Die physikalische Simulation ermöglicht ein Spiel nach einem glaubwürdigen Modell der Wirklichkeit. Grundlegende Formeln der newtonschen Physik wurden so implementiert, dass das Verhalten der Billardkugeln in vielen Punkten authentisch erscheint. Die schnelle Meshkollision führt zu korrekten Kollisionen mit der Umwelt, auch wenn Kugeln in Taschen fallen. Der Queue interagiert, besonders in Verbindung mit dem Haptic Device, wie ein echter Queue: Wird die Kugel versetzt und hart angespielt, so springt sie und erhält einen Rückdrall. Dieses Springen konnten wir auch bei populären Open-Source-Projekten [3, 4] nicht beobachten, obwohl es ein reales Phänomen ist. Ein kommerzieller Anbieter liefert ein Billardspiel [16], bei dem Bälle vom Tisch springen, wenn sie im Tisch verlegte Minen (sic!) berühren. Wir können mit unserer Simulation „Jump Shots“ durchführen, wozu nur wenige andere Billardsimulationen in der Lage sind.

Ein zentrales Thema des Spiels war auch die Bedienung mit dem Haptic Device. Mit diesem lässt sich der Queue in sechs Freiheitsgraden (3 Dimensionen, 3 Rotationsachsen) bewegen. Die Queuespitze interagiert dabei mit der Spielwelt, die Kugeln, der Tisch, sogar die Diamanten, die in der Bande als Zielhilfe eingelassen sind, lassen sich erfühlen. Darüber hinaus haben wir uns erfolgreich mit dem Problem befasst, mit dem Haptic Queue einen gezielten Schuss auf eine Kugel abzugeben, was De Paolis et al. [1] nicht lösen konnten: Aufgrund der Kürze und des begrenzten Bewegungsspielraums des Controller-Stiftes kann man ihn nicht wie einen Billardqueue mit beiden Händen fassen. Wir fixieren durch Tastendruck alle Freiheitsgrade und lassen nur noch eine gerade Bewegung in Richtung der Queueachse zu. Eine Kraft sorgt dafür, dass das Haptic

Device ebenfalls nur noch auf dieser Gerade bewegt werden kann. Die Steuerung ist auf diese Weise genauso präzise wie mit der Maus, aber bedeutend realistischer.

Man merkt unserer Software jedoch den jungen Entwicklungsstand an. Die Menüführung hakt noch zuweilen, die Spielregel-Steuerung zeigt in extremen Situationen überraschende Verhaltensweisen, die Kompatibilität auf verschiedenen Betriebssystemen ist noch nicht perfekt und viele weitere Kleinigkeiten sind noch zu korrigieren. In der Hinsicht besitzen die etablierten Billardsimulationen einen großen Vorsprung, welcher monatelange Weiterentwicklung und Fehlerkorrektur voraussetzt. Wir haben einen *SourceForge*-Account[1] angelegt in dessen Kontext wir *HapticBillard* weiterentwickeln werden. Wir hoffen, dass wir uns mit ein wenig Zeit und Mühe irgendwann in den Rängen der großen Open-Source-Billardspiele etablieren können.

[1] HapticBillard - `http://sourceforge.net/projects/hapticbillard/`

Literaturverzeichnis

[1] Lucio T. De Paolis, Marco Pulimeno, and Giovanni Aloisio. Different simulations of a game of billiards based on force feedback and marker detection. *CGI 2008 Conference Proceedings*, pages 338–341, 2008.

[2] OGRE - Object-Oriented Graphics Rendering Engine, 2009. `http://www.ogre3d.org/`. Recv.: March 2009.

[3] Stefan Disch, Tobias Nopper, and Martina Welte. BillardGL, 2001. `http://www.billardgl.de/`. Recv.: March 2009.

[4] Florian Berger. FooBillard, 2004. `http://foobillard.sunsite.dk/`. Recv.: April 2009.

[5] M. Teschner. Simulation der Computergrafik, 2007. `http://cg.informatik.uni-freiburg.de/teaching.htm`. Recv.: Jan 2009.

[6] Brian Townsend. illiPool - An Attempt at a Real-Life Simulation of the Mechanics of Billiard Balls, May 2003. `http://archive.ncsa.uiuc.edu/Classes/MATH198/townsend/math.html`. Recv.: March 2009.

[7] Brandon Itkowitz, Josh Handley, and Weihang Zhu. The OpenHaptics Toolkit: A Library for Adding 3D Touch Navigation and Haptics to Graphics Applications. *World Haptics Conference*, 0:590–591, 2005.

[8] CEGUI - Crazy Eddie's GUI System, 2009. `http://www.cegui.org.uk/`. Recv.: March 2009.

[9] Deutsche Billard-Union e. V. Materialnormen Pool, 2007. `http://www.billard-union.de/downloads/2.8.3materialnormenpool.pdf`. Recv.: March 2009.

[10] Wikipedia, Die freie Enzyklopädie: 8-Ball. `http://de.wikipedia.org/wiki/8-Ball`. Recv.: Dez. 2008.

[11] David Harel. Statecharts: A Visual Approach to Complex Systems. *Science of Computer Programming*, pages 231–274, August 1987.

[12] G.D. Schultz. A Lightweight Implementation of UML Statecharts in C++, 2005. `http://www.codeproject.com/KB/cpp/Statechart.aspx`. Recv.: March 2009.

[13] Paul J. Lucas and Fabio Riccardi. CHSM: Concurrent Hierarchical Finite State Machine, 2007. `http://chsm.sourceforge.net/`. Recv.: March 2009.

[14] The Boost Statechart Library, April 2007. `http://www.boost.org/`. Recv.: March 2009.

[15] FMOD, music & sound effects system, 2009. `http://www.fmod.org/`. Recv.: March 2009.

[16] Blade Interactive. World Championship Pool 2004, 2004. `http://www.jaleco.com/`. Recv.: April 2009.

[17] Lucio T. De Paolis, Giovanni Aloisio, and Marco Pulimeno. A Simulation of a Billiards Game Based on Marker Detection. *Advances in Computer-Human Interactions, 2009. ACHI '09*, pages 148–151, Feb. 2009.