Billard

Dokumentation

für das Projektseminar
im Studiengang Medieninformatik
der Hochschule Furtwangen

Prof. Dr. Dirk Eisenbiegler

Stefan Fischer
Matrikelnummer: 22 90 63

Maurizio Camagna
Matrikelnummer: 22 91 55

Markus Wellmann
Matrikelnummer: 22 93 99

Alexander All
Matrikelnummer: 22 83 22

Inhaltsverzeichnis

1 Einleitung

Problemstellung

In Deutschland mangelt es an Menschen, die sich für die Themen Mathematik, Ingenieurwesen, Naturwissenschaften und Technik auseinandersetzen. Dabei werden gerade Fachkräfte mit diesen Qualifikationen in der Deutschen Wirtschaft dringend gebraucht. Hierzu wurde 2008 eine Initiative mit dem Namen MINT(Mathematik, Ingenieurwesen, Naturwissenschaft und Technik) gegründet.

Aufgrund des Fachkräfte-Mangels in dem Mint-Fächern soll das Projekt Billard dazu beitragen, dass sich Menschen in Deutschland mehr für diese Themen interessieren. Wir haben in unserem Projekt dabei die Schwerpunkte auf die Mechanik, die Mathematik und der Informatik gelegt. Der Nutzer unseres Programmes setzt sich spielerisch mit diesen Schwerpunkten auseinander, mit dem Ziel ein erhöhtes Interesse in dem Bereich zu bekommen. Das Billardspiel, sollte so aufgebaut sein, dass der Nutzer Freude an dem Programm hat. Dabei wird aber der Lerneffekt nicht vernachlässigt.

Ziel des Projekts

Ziel des Projekts war es ein Billard-Lernspiel zu entwickeln. Der Benutzer sieht am Anfang ein gewöhnliches Billardspiel, dieses Spiel verbirgt allerdings viel mehr als es anmuten lässt. Das besondere ist, dass es einen Lerneffekt beim Benutzer bewirken soll. Das Spiel soll den Benutzer in einen regelrechten Spielflow bringen und sich mit den Eigenheiten dieses Billardspiel beschäftigen. Je tiefer der User sich dann mit dem Billardspiel und seinen Besonderheiten auseinandersetzt, um so tiefer drängt der Benutzer in die Bereiche der Mechanik, Mathematik bzw. Informatik ein.

Denn das Billardspiel sollte nicht nur gut aussehen, sondern während dem Spiel sollten die Physikalischen Elemente der Realität beachtet werden. Eigenschaften wie Masse der Kugeln, Beschleunigung, Impuls, Impulserhaltung, Energieerhaltung und Rollreibung haben einen erheblichen Einfluss auf den Spielverlauf. Wenn der Benutzer sich mit diesen Parametern auseinandersetzt, kann er anhand von Berechnungen den perfekten Billardstoß ausführen.

Zusätzlich kann der User nicht nur spielen, sondern auch spielen lassen. Der Nutzer kann sich nämlich mit dem Schreiben eines Billardroboters auseinandersetzen. Und vertieft sich dadurch mehr mit der Informatik. Der Nutzer kann Algorithmen für einen Billardroboter schreiben und diesen Roboter im Spiel gegen andere Spieler oder Roboter antreten lassen. Ziel ist es dabei einen Billardroboter zu entwerfen, der möglichst viele Bälle mit einem Stoß versenkt und wenig Rechenaufwand sowie Speicherplatz

benötigt. Um seine Fähigkeiten in Informatik mit anderen vergleichen zu können, kann er seinen Billardroboter zu einem Turnier schicken. Hier sollte eine Turnierplattform entwickelt werden, welches die Qualität des Algorithmus bewertet, und Sie mit Billardrobotern von anderen Benutzern vergleicht. Eine Möglichkeit wäre, dass ein Turnierportal zentral über eine Webanwendung läuft, in der regelmäßig Turniere veranstaltet werden. Bevor ein Turnier beginnt, kann der Nutzer seinen Billardroboter auf der Turnierplattform hochladen. Die Turnierplattform sammelt dann zum Turnierstart alle Billardroboter ein und lässt diese unter bestimmten Kugelpositionen und sonstigen Einstellungen gegeneinander spielen. Die Qualität des Stoßes, der Rechenaufwand und der Speicherbedarf führen zu einer Gesamtnote. Die Billardroboter mit den besten Resultaten kommen eine Runde weiter, während die anderen aus dem Turnier ausscheiden. Dies wird so lange fortgesetzt, bis es einen Gewinner gibt der als Belohnung in der Bestenliste auftritt.

Bei der Physik wäre der Drall bzw. das Effet-Spiel eine mögliche Erweiterung. Eine 2. mögliche Erweiterung bestand darin in der Physik das Planeten-Billard zu integrieren. Beim Planeten-Billard ist die Besonderheit darin begründet, dass sich Kugeln gegenseitig anziehen können.

Bei den Zielen konnte die Projektgruppe eigene Prioritäten setzen, und somit über das Endprodukt des Projekts bestimmen.

2 Grundlagen

MINTlets Begriffserklärung und Aufgabe

MINTlet ist eine Wortneuschöpfung bestehend aus dem Initialwort *MINT* und der Endung *‚-let'*.

Die Abkürzung MINT setzt sich aus den Anfangsbuchstaben der Schulfächer Mathematik, Informatik, Naturwissenschaften und Technik zusammen. MINT ist eine Initiative zur Förderung dieser Fächer und mit ihnen verbundenen Berufen zur Verringerung des Fachkräftemangels in Deutschland.

Die Wortendung ‚-let' kommt aus dem Englischen und ist mit den deutschen Endungen ‚-lein' und ‚-chen' vergleichbar. Beispiele für die Verwendung dieser Endung in der Informatik sind Applets, Servlets oder Midlets. Sie stehen für JAVA-Anwendungen, die für bestimmte Umgebungen und Aufgabengebiete erstellt wurden: Applets für Webbrowser, Servlets für Server und Midlets für Mobiltelefone. Überträgt man nun diese Bedeutung auf den Begriff MINTlets, so würde dies eine Softwareanwendung im Kontext von MINT bezeichnen.

MINTlets sollen ihre Nutzer anregen sich mehr für ‚MINT-Fächer' zu begeistern und ihnen idealerweise auch Grundlagen oder fortgeschrittenes Wissen in den definierten Fachgebieten beibringen.

Billard

Billard ist ein Spiel für zwei Personen. Zentrale Bestandteile des Spiels sind die Billardkugel (auch Bälle genannt), das Queue und der Billardtisch.

Billard mit den Bestandteilen Tisch, Kugeln und Queue wurde wohl im späten 15. Jahrhundert zum ersten Mal gespielt und hat sich aller Wahrscheinlichkeit nach aus anderen verwandten Freiluftballspielen wie Cricket und Croquet entwickelt, um dem Spiel auch bei schlechtem Wetter in geschlossenen Räumen nachgehen zu können.

Heute differenziert man im Billard grob zwischen den drei großen Überarten Carambolage, Poolbillard und Snooker. Sie unterscheiden sich vor allem durch die Art des Tisches und der verwendeten Kugeln. Die meist gespielte und bekannteste Billardversion in Deutschland ist 8-Ball, welches zu Poolbillard gehört.

8-Ball Regeln

Beim 8-Ball wird mit der weißen Kugel, dem Spielball und 15 nummerierten Objektbällen gespielt. Die Kugeln mit den Nummern 1-7 sind einfarbig und werden ‚Volle' genannt. Die Kugeln 9-15 haben jeweils nur einen farbigen Streifen. Diese werden als ‚Halbe' bezeichnet. Der Ball mit der Nummer 8 ist auch einfarbig schwarz, hat jedoch eine Sonderrolle.

Im Laufe des Spiels werden die Halben und die Vollen auf die beiden Spieler aufgeteilt. Sind alle Kugeln einer Art in den Löchern versenkt worden, hat der Spieler, dem diese zugeordnet wurden, die Möglichkeit das Spiel durch das Einlochen der schwarzen Kugel zu beenden.

Offiziell ist 8-Ball ein Ansagespiel. Das bedeutet, dass man vorher ansagen muss, welcher Ball in welche Tasche gespielt wird. Ein Spieler darf solange weiterspielen, bis er es ihm misslingt, die angesagte Kugel einzulochen ohne dabei ein Foul zu begehen. Wenn der Gegner ein Foul begeht, hat man grundsätzlich ‚Ball in Hand', das heißt man kann die weiße Kugel an eine beliebige Stelle des Tisches legen und von dort aus in eine beliebige Richtung weiterspielen. Ausnahme bildet der Anstoß, für welchen Sonderregeln gelten. Das offizielle Regelwerk ist natürlich um einiges umfangreicher und es gibt eine Vielzahl von Regeln und Sonderfällen.

In der breiten Öffentlichkeit wird 8-Ball fast immer ohne Ansagen gespielt und auch oft mit weiteren Änderungen zu den offiziellen Regeln. Es gibt viele unterschiedlichen Spielweisen, die sich teilweise sogar schon von Freundeskreis zu Freundeskreis unterscheiden können.

Physikalische Grundlagen

Die Physik berechnet im Voraus die gesamte Simulation. Dazu wird die Anfangsgeschwindigkeit der weißen Kugel übergeben. Die Anfangspositionen der Kugeln erhält die Simulation aus den einzelnen Kugelobjekten.

Die Simulation selbst wird in dynamischen Zeitintervallen berechnet wobei in jedem folgende Schritte abgearbeitet werden:

- Ermittlung der Gesamtdauer des Stoßes
- Dynamische Ermittlung des Zeitintervalls
- Neue Position der Kugeln berechnen
- Kollisionserkennung
- Kollision zwischen Kugeln
- Kollision zwischen Kugeln und Banden

Ermittlung der Gesamtdauer des Stoßes

Aus der Anfangsgeschwindigkeit die an die weiße Kugel übergeben wird, kann die Dauer des gesamten Stoßvorgangs ermittelt werden. Geht man davon aus, dass die gesamte **kinetische Energie** (5) zusammen mit der **Rotationsenergie** (2) unter Berücksichtigung der **Winkelgeschwindigkeit** (3) und dem **Trägheitsmoment einer Kugel** (4) für die **Reibarbeit** (1) verwendet wird so gilt (6):

$$W(Reibung) = \mu * r * m * g * s \quad (1)$$

$$E(Rotation) = \frac{1}{2} * J * w^v \quad (2)$$

$$w = \frac{v}{r} \quad (3)$$

$$J = \frac{2}{5} m * r * v^2 \quad (4)$$

$$E(kin) = \frac{1}{2} * m * v^2 \quad (5)$$

$$s = \frac{E(kin)+E(Rotation\ )}{\mu * m * g} \quad (6)$$

$$v = \frac{s}{t} \quad (7)$$

$$t = \frac{s}{v} \quad (8)$$

Nachdem die Strecke in (6) ermittelt worden ist kann die Dauer (8) des Stoßvorgangs nach Umformung des **Weg-Zeit-Gesetzes** (7) errechnet werden. Bei der Berechnung wird die Annahme gemacht, dass die weiße Kugel ohne Kollision den Weg zurücklegt. Diese Überlegung ist richtig da nach dem **Impulserhaltungssatz** die Summe aller Impulse vor dem Stoß gleich der Summe der Impulse nach dem Stoß ist. Daher gilt für den **elastischen Stoß** (zwischen 2 Körpern) die Impulserhaltung (9) und die Erhaltung der kinetischen Energie (10):

$$\overrightarrow{p1} + \overrightarrow{p2} = \overrightarrow{p1}' + \overrightarrow{p2}' \quad (9)$$

$$\frac{1}{2} m1 * v1^2 + \frac{1}{2} m2 * v2^2 = \frac{1}{2} m1 * (v1)'^2 + \frac{1}{2} m2 * (v2)'^2 \quad (10)$$

Dynamische Ermittlung des Zeitintervalls

Auf der Suche nach einer Alternative zu der statischen Methode, bei der die Betrachtung des Tisches und dessen Objekten in konstanten Zeitabständen erfolgt, kam eine dynamische Lösung in Frage. Dieser Ansatz sollte durch die variable Schrittgröße Effizienz und Sicherheit bietet, bei der keine Kollision unbemerkt bleibt.

Folgende Überlegung bildet hierfür die Grundlage: im ungünstigsten Fall bewegen sich die zwei schnellsten Kugeln (K_1, K_2) (14) frontal aufeinander wobei der Abstand s > Radius r beträgt (Abb. 2.1).

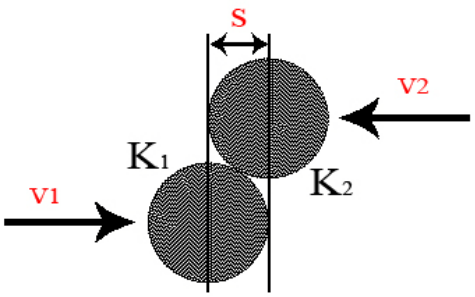

Abb. 2.1

Aus diesem Szenario lässt sich durch Umformung (12) der Gleichung für **gleichmäßige beschleunigte Bewegung** (11) das Intervall mit Hilfe der **p-q-Formel** (13) berechnen. Dabei erhält man zwei Werte wobei nur der positive für uns relevant ist.

$$s = \frac{1}{2}a * t^2 + v * t \tag{11}$$

$$0 = t^2 + \frac{v}{2a} * t - \frac{s}{2a} \tag{12}$$

$$x1{,}2 = -\frac{p}{2} \pm \sqrt{\frac{p^2}{2} - q} \tag{13}$$

$$v = v1 + v2 \tag{14}$$

Neue Position der Kugeln berechnen

Die Verschiebung der Kugeln um die Strecke $\vec{s}(t)$ aus dem zuvor errechneten Intervall t (13) erfolgt nach dem **Weg-Zeit-Gesetz** (15):

$$\vec{s}(t) = \frac{1}{2} * \vec{a} * t^2 + \overrightarrow{v0} \tag{15}$$

Alternativ zu (15) steht das **klassischen Runge-Kutta-Verfahrens** (16) der Physikbibliothek zur Verfügung. Diese 4-stufige Methode dient zur numerischen Lösung von Anfangswertproblemen. Sie ist effizienter als die explizite Verfeinerung des Intervalls. Die Anwendung dieses Verfahrens ist simpel dabei wird $f(x, y)$ auf die erste Ableitung von (15) $s'(t) = a * t + v$ abgebildet.

$$k1 = f(xn, yn)$$

$$k2 = f\left(xn + \frac{h}{2}, yn + \frac{k1}{2}\right)$$

$$k3 = f\left(xn + \frac{h}{2}, yn + \frac{k2}{2}\right)$$

$$k4 = f(xn + h, yn + k3)$$

$$s(t) = yn + \frac{h}{6} * (k1 + 2 * k2 + 2 * k3 + k4) \tag{16}$$

Nach dem $s(t)$ bestimmt und der Winkel bekannt ist, können sofort mit Hilfe des Pythagoras die neuen Koordinaten in das Kugelobjekt geschrieben werden.

Fehlt nur noch die neue Geschwindigkeit. Diese lässt sich nach dem **Geschwindigkeits-Zeit-Gesetz** (17) bestimmen wobei $\vec{a}$ ein negatives Vorzeichen erhält.

$$v(t) = \vec{a} * t + v0 \tag{17}$$

Kollisionserkennung

Die Erkennung eines Zusammenpralls zwischen zwei Kugeln ist keine Herausforderung, dieser kommt bei einem Abstand $s \leq$ Radius r zustande. Die Schwierigkeit liegt hier vielmehr bei der Ermittlung des genauen Zeitpunktes. Im Idealfall (vollkommen elastischer Stoß) wäre dieser gleich dem Radius (18).

$$s = r \tag{18}$$

Bei der dynamischen Festlegung des Intervalls t (siehe oben) wurde sichergestellt, dass keine Kollision unbemerkt bleibt. Unabhängig davon können die Elemente zum Zeitpunkt der Betrachtung einander durchdringen.

Mit Hilfe der Rekursion und dem Halbschrittverfahren (Abb. 2.2) konnte das Problem elegant gelöst werden. Dabei wird in jedem Intervall der Abstand zwischen den Objekten geprüft. Beträgt dieser dabei $s > r$ so hat keine Kollision stattgefunden und es kann zum nächsten Zeitpunkt t_{n+1} gesprungen werden. Trifft jedoch die Bedingung $s < r - Toleranz$ zu werden folgende Schritte abgearbeitet:

1. Rücksprung zum Zeitpunkt t_{n-1}
2. Verschiebung der Objekte um $\frac{t_{n-1}}{2}$
3. Erneute Prüfung ab (19) gilt

   $$[r - Toleranz \leq s \leq r] \tag{19}$$

   - Wenn erfüllt wird zum nächsten Zeitpunkt t_{n+1} gesprungen
   - Wenn nicht zutreffend wird mit Punkt 1. wieder begonnen und in Punkt 2. erfolgt die Verschiebung um $\frac{t_{n-1}}{4}$

Wenn der Zeitpunkt sowie Distanz, die innerhalb der Toleranzgrenze liegt ermittelt worden sind kann mit der Berechnung des Abprallwinkels begonnen werden. Der Toleranzwert kann mit dem Argument des nicht vollkommen elastischen Stoßes gerechtfertigt werden da der vollkommen elastische Stoß nur in der Theorie existiert und nicht in der Praxis.

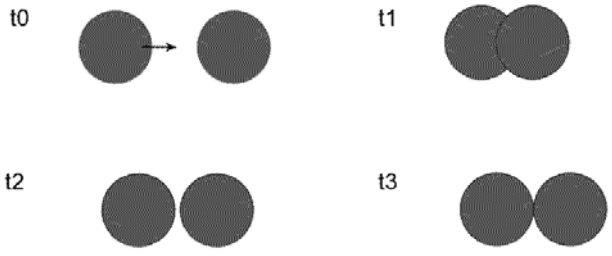

Abb. 2.2

Kollision zwischen Kugeln

In dieser Simulation bildet der **elastische Stoß mit gleichen Massen** die Grundlage für die Winkel und Geschwindigkeitsberechnung nach einer Kollision. Als Quelle diente [4] außerdem wird in folgenden Fällen die Rotation vernachlässigt.

Beim Billard wird zwischen zwei Arten von Stößen unterschieden:

- Gerader Stoß: Die Bahnen beider Schwerpunkte liegen auf einer Geraden

- Schiefer Stoß: Die Schwerpunkte der Stoßpartner liegen auf der Normalen zur Berührungsebene durch den Berührungspunkt (Stoßnormale)

Zentraler elastischer Stoß

Die Berechnung des geraden elastischen Stoßes erfolgt nach den gleichen physikalischen Grundlagen wie der dezentrale elastische Stoß (siehe unten).

Dezentraler elastischer Stoß

Bezugspunkt wird so gewählt, dass der zweite Körper ruht. Um den Stoß zu berechnen wird der Stoßparameter b benötigt um auf den Winkel α zu kommen (Abb. 2.3).

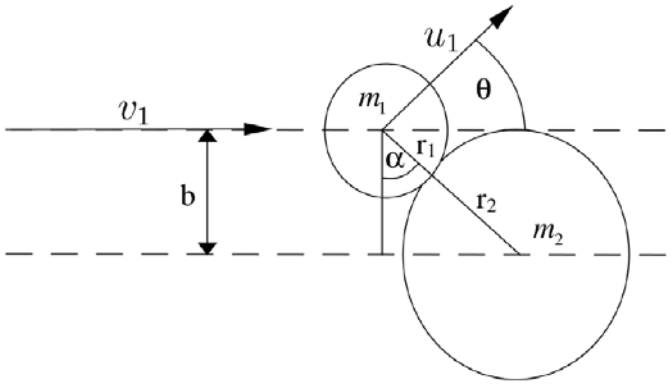

Abb. 2.3

Daraus lässt sich die Formeln für den Stoßparameter b bzw. des Winkels α kommen:

$$b = 2r * \cos(\alpha) \tag{21}$$

$$\alpha = \cos^{-1}(\frac{b}{2r}) \tag{22}$$

Nun wird die Geschwindigkeit des ersten Körpers *v1* zum Zeitpunkt des Stoßes in zwei Teilgeschwindigkeiten aufgeteilt. Zum einen in den Radialanteil *vr1*, zum anderen in den Anteil der Normalen dazu, dem Tangentialanteil *vt1*:

$$vr1 = v1 * \sin(\alpha) \tag{23}$$

$$vt1 = v1 * \cos(\alpha) \tag{24}$$

Lediglich der Radialanteil wirkt auf dem zweiten Körper, der Tangentialanteil steht senkrecht und hat so keine Wirkung auf den zweiten Körper. Für die Berechnung der neuen Geschwindigkeiten dienen daher folgende Formeln:

$$u1 = \sqrt{{v_{r1}}^2 * {v_{t2}}^2} \tag{25}$$

$$u2 = \sqrt{{v_{r2}}^2 * {v_{t1}}^2} \tag{26}$$

Um nun den Winkel θ, in welchem der zweite Körper abgestoßen wird, zu berechnen, genügt ein Blick auf die Abbildung 2.3. Hier ist unschwer zu erkennen, dass von einem rechten Winkel einfach α abgezogen wird:

$$\theta_2 = 90℃ - \alpha = 90℃ - \cos^{-1}(\frac{b}{2r})$$

Der Winkel des ersten Körpers setzt sich aus den Winkeln α und β zusammen:

$$\theta_1 = \alpha + \beta = \cos^{-1}(\frac{v_{r1}}{v_{t2}})$$

Kollision zwischen Kugeln und Banden

Dies entspricht einer Reflektion und wird nach dem Prinzip Einfallswinkel = Ausfallswinkel durchgeführt. Für die Umsetzung der „komplexen“ Bande basiert auf mathematischen Grundlagen die im folgendem diskutiert werden. Im Allgemeinen wird der Geschwindigkeitsverlust sowie die Rotation die durch die Kollision mit der Bande entstehen nicht berücksichtigt.

Prinzip der „komplexen“ Bande

Die komplexe Bande wurde aus sechs Segmenten (Abb. 2.4), die jeweils aus drei Geraden bestehen modelliert. Dabei wird in jedem Durchlauf auf einen Schnittpunkt der einzelnen Kugelobjekte mit den Bandengeraden geprüft.

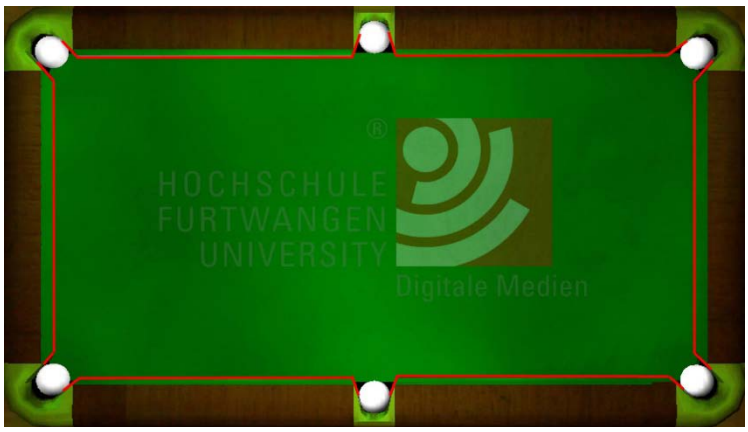

Abb. 2.4

Formel zur Berechnung der Schnittpunkte

Für die Schnittpunktermittlung werden die Kreisgleichung (27), wobei $(xM|yM)$ die Koordinaten des Mittelpunktes sind und die Geradengleichung (28) benötigt.

$$(x - xM)^2 + (y - yM)^2 = r^2 \tag{27}$$

$$y = mx + c \tag{28}$$

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{29}$$

Verfahren

Geradengleichung (28) anhand der Koordinaten aufstellen, in Kreisgleichung (27) einsetzen und nach Parameter auflösen. Mit Hilfe der Mitternachtsformel (29) können sich je nach Geradenlage null, ein oder zwei Werte für den Parameter ergeben.

3 Projektmanagement

Eingesetzte Techniken und Kommunikation

Obwohl schon ein Grundgerüst auf der Schwarzwälder VM vorhanden war, wurde zu Beginn sehr viel Zeit gebraucht, um das Projektmanagementtool Trac zusammen mit einem SVN Repository zu implementieren. Es stellte sich jedoch schnell heraus, dass der Funktionsumfang von Trac für dieses Projekt überdimensioniert war. Die Wiki und das Bug Tracking System wurden von den Projektmitgliedern nicht angenommen und als zu komplex und unnötig angesehen. Lediglich der RSS-Feed der Timeline, welcher alle Changesets im SVN Repository übermittelt, war bis zum Schluss für die Übersicht über das Projekt sehr hilfreich. Des Weiteren wurde noch das Projektmanagementtool Planner eingesetzt, welches sich aber auch bald als zu unflexibel erwies. Da der Projektmanager auch sehr stark als Programmierer eingebunden war, wurde das Projektmanagement vor allem in der zweiten Hälfte des Projektes stark zurückgefahren und nur noch mit GoogleGroups und GoogleDocs, sowie mit handschriftlichen Aufzeichnungen und mündlicher Kommunikation durchgeführt.

Probleme und Leson Lerand

Es wurde bis zum Schluss kein einheitliches Bug Tracking System verwendet, was sich als Fehler herausgestellt hat. Eine Mischung aus Projektmanager und Teammitglied bringt viele Probleme mit sich und sollte wenn möglich vermieden werden. Ein ausführliches internes Kick-Off-Meeting ist von unschätzbarer Bedeutung. Regeln wie das Festlegen einer einheitlichen Programmiersyntax, eines einheitlichen Bug Tracking Systems und auch das Bestimmen von regelmäßigen Terminen und viele weitere Aufgaben sollten zu Beginn Projektes fest geregelt sein.

Das Projektmanagement war in diesem Team eine große Herausforderung. Dies lag auch daran, dass vor allem zu Beginn des Projektes, das Projektmanagement als nicht ernst zu nehmende Tätigkeit angesehen wurde, was auch für Spannungen zwischen dem Team und dem Manager gesorgt hat.

Zusammenfassend lässt sich sagen, dass für das Managen eines Softwareprojektes bei Weitem nicht nur gute Programmierkenntnisse ausreichen und es auch bei kleinen Projekten mit viel Aufwand verbunden ist, der nicht unterschätzt werden sollte.

4 Billard MINlet

Ziele/Vorgaben

Motivationsinteresse an PhysikerIn

Aufgrund dessen, dass das Billardspiel auf wahren physikalischen Elementen beruht, hat der Benutzer, der sich mit der Physik des Spiels auseinandersetzt, einen erheblichen Vorteil gegenüber dem Spieler ohne dieses Hintergrundwissen. Dabei bietet das Programm anhand einer Hilfe dem User an, sich mit den Gesetzen der Physik auseinanderzusetzen. Der User weiß anhand eines Hilfe-Fensters, welche physikalischen Elemente das Programm berücksichtigt. Beim auseinandersetzen mit den Physikalischen Elementen kann der Benutzer anhand von Berechnungen den optimalsten Stoß ermitteln. Dieses neu erlernte Wissen lässt den Nutzer mit ganz anderen Augen auf das Billardspiel blicken. Die neu erlernten Gesetze kann der Nutzer auch auf dem richtigen Billardspiel übertragen. Bei einer Runde Billard auf dem richtigen Tisch kann man, durch das erlernte physikalische Wissen, sich einen Vorteil gegen Freunde und Bekannte verschaffen.

Informatik wecken

Wie in dem Projektspiel beschrieben, liegt beim Billardroboter der Schwerpunkt in der Informatik. Nachdem der Benutzer sich mit den Physikalischen Elementen auseinandergesetzt hat, hat er die Möglichkeit mit dem Billardroboter automatisieren zu entwickeln, die in jeder Situation das Ziel haben den perfekten Stoß durchzuführen. Dabei ist das Ziel des Nutzers einen Billardroboter zu entwickeln, das einen möglichst guten Stoß macht, wenig Rechenleistung benötigt und einen geringen Speicherbedarf hat. Anhand der Beschreibung zum Thema Billardroboter und dem Beispiel-Code kann er seinen eigenen Roboter bauen. Durch den regelmäßigen Wettbewerb versucht er seinen Billardroboter zu perfektionieren um sich evtl. als Gewinner auf der Bestenliste wiederzufinden.

Motivationswirkung von Computerspielen nutzen

Computerspiele können den Nutzer in einen Flow versetzen, in der der Nutzer so stark vom Spiel gefesselt ist, dass er alles um sich herum vergisst und gar nicht merkt wie die Zeit vergeht. Gerade junge Menschen verbringen gerne ihre Freizeit mit Computerspielen. Diesen Spielflow, den die Benutzer bei gewöhnlichen Computerspielen haben, wollen wir auch in dem Lernprogramm nutzen. Das Ziel ist es auf spielerische Weise dem Benutzer was beizubringen, so dass er sich mit Spaß an den Themen Mathematik, Mechanik und Informatik nähert und dabei nicht das Lernen sondern das Spielen im

Mittelpunkt steht. Das auseinandersetzen mit den vorher genannten Themen bringen ihn sozusagen einen Level weiter und er kann sich gegen andere Benutzer messen, wer das Spiel am besten beherrscht.

Eigene Ziele

Neben den Zielen und Vorgaben die das Projekt durch seine Aufgabenstellung vorbestimmt hat, brachten die Mitglieder des Projektteams auch eigene Wünsche vor, welche das Projekt maßgeblich im Verlauf beeinflusst haben. Die wichtigsten sollen im folgenden Abschnitt erläutert werden.

3D-Grafik

Das implementieren von 3D-Visualisierungen in Softwareprojekten ist in den letzten Jahren deutlich durch ein wachsendes Angebot an 3D-API's, Bibliotheken und Frameworks vereinfacht worden. Dies ist z.B. auch der Grund weshalb 3D-Grafik inzwischen Einzug in die Welt der Webseiten gefunden haben, und speziell 3D-Grafik in Flash-Seiten zu einem boomenden Sektor herangewachsen ist.

Während jedoch der Einsatz von 3D im Webbereich aufgrund der darunter leidenden Übersicht zu Hinterfragen ist und bei einigen Spielen, wie dem Schachspiel, durch eine 3D-Ansicht kein großer Mehrwert entsteht, war das Team sich darüber einig dass es sich im Rahmen eines Billardspieles durchaus als sinnvoll erweisen sollte. Der Grund hierfür war, dass der Blickwinkel auf den Tisch im Billard durch die Stoßhaltung selbst fest vorgegeben ist. Somit ist eine zweidimensionale Vogelperspektive zwar Übersichtlicher, jedoch wesentlich unnatürlicher. Deshalb wurde beschlossen eine 3D-Sicht zu erstellen, aus welcher jederzeit in eine Vogelperspektive gewechselt werden kann.

Computer gegeneinander antreten lassen

Eine Vorgabe des Projektes war die Anbindung an ein Testframework zur Ermittlung der Leistungsfähigkeit der Billardroboter. Dies wurde jedoch von Teilen des Teams ob der Aussagekraft dieser Messungen in Frage gestellt. Grund hierfür war, dass die beiden Mitglieder die diese Einwände brachten, in ihrer Freizeit sehr gerne Schach spielen und deshalb wussten, dass im Schach zur Messung der Kraft eines Schachprogramms, dieses gegen andere Schachprogramme in Turnieren antreten muss. Grund hierfür ist, dass die Leistung allein, also beispielsweise die Anzahl der Berechnungen in der Sekunde, keine Aussagekraft über die Qualität des Zuges gibt. Ein schlechtes Programm wird einen taktisch, also kurzfristig, guten Zug bevorzugen, auch wenn dieser strategisch und somit auf den gesamten Verlauf des Spieles unklug ist.

Dies lässt sich auch auf das Billardspiel anwenden. Wenn ein Spieler in einer Situation ist, in welcher ein guter Stoß schlicht nicht möglich ist, so hat auch die Geschwindigkeit mit welcher der Roboter zu dieser Einsicht kommt keinen Nutzen. Stattdessen müsste in dieser Situation ein Stoß gemacht werden, welcher den Gegenspieler in eine mindestens ebenso ungünstige Position versetzt.

Aus diesem Grund wurde beschlossen, dass es eine höhere Priorität hat einen Modus zu erstellen in welchem die Roboter gegeneinander antreten können und diesen möglicherweise zu einer Turnierplattform auszubauen.

Weltraumphysik

Die Idee des Spieles ist, das Interesse für die physikalischen Vorgänge auf dem Tisch zu wecken und sich dadurch spielerisch mit der Physik auseinander zu setzen. Das Problem hierbei ist jedoch, dass die Physik auf dem Billardtisch denselben Regeln folgt wie alle anderen Vorgänge unseres Lebens. Es ist dadurch also schwer die Neugierde an diesem Thema zu wecken, da sich dieses Thema aus Vorgängen zusammensetzt, die uns seit unserer Geburt bestens geläufig sind.

Aus diesem Grund war es unser Wille eine Weltraumphysik zu programmieren, welche sich komplett anders Verhält als die uns aus dem Alltag bekannte. Durch diesen Überraschungseffekt ist es wahrscheinlicher die Neugierde und den Spieltrieb eines Benutzers zu wecken und ihn dadurch dazu bewegen sich auch mit den bereits bekannten physikalischen Effekten auseinander zu setzen.

Vielseitige Verwendung

Die Billardspiele, welche es in Massen im Internet zu finden gibt, sind für gewöhnlich zum Zwecke der Unterhaltung entwickelt und dafür optimiert worden. Dies ist nicht verwerflich wenn man sich über Einsatzort und Nische sicher ist, in welcher das Programm später platziert sein soll. In unserem Falle war diese Bedingung jedoch nicht erfüllt, da der Entwicklungsprozess sehr dynamisch war, und man sich über das Resultat anfangs keine genauen Aussagen treffen konnte. Aus diesem Grunde haben wir uns das Ziel gesetzt das Programm sehr offen zu halten um es zu einem späteren Zeitpunkt möglicherweise für ein Einsatzgebiet optimieren zu können. Dies hat zur Folge dass das Programm seine Nische mit der Zeit hoffentlich selbst findet, indem jemand das Programm seinen Bedürfnissen nach abändert um es beispielsweise ausschließlich, wie oben erwähnt, als Turnierplattform für Billardroboter zu nutzen.

5 Umsetzung

Programmaufbau

Zu Beginn des Projektes waren die Art und Anzahl der zu implementierenden Features und Techniken schwer zu planen, da es nicht möglich war realistisch einzuschätzen wie hoch der Grad der Komplexität der Physik sein würde, welche Daten ein Billardroboter braucht um gute Schläge auszuführen, welche Technik eingesetzt werden wird für die 3D Darstellung des Tisches etc. Da das Projekt also nicht im Vorfeld strukturiert voraus geplant werden konnte, musste sich das Team in kleinen Schritten an die gesetzten Ziele heran tasten.

Eine solche Arbeitsweise ist jedoch mit der Gefahr behaftet dass die Infrastruktur des Programms auf Programmteile aufsetzt, welche zu einem späteren Zeitpunkt verändert werden müssen und dadurch zu Inkompatibilitäten der Programmteile untereinander und der Infrastruktur führen können. Um im späteren Verlauf das Projektes problemlos um Features erweitern zu können, ohne diese in den Quellcode unsauber einhacken zu müssen, wurde Wert darauf gelegt ein möglichst flexibles Grundgerüst zu Programmieren, welches den kommenden Anforderungen gerecht wird.

Um dies zu realisieren wurde das Programm zuerst in 5 logische Module aufgeteilt, welche essentiell für das Billardspiel sind. Diese wären: ein Grafikmodul, ein Modul zur Interaktion, eines für die physikalischen und mechanischen Berechnungen, das Regelwerk, sowie das Kernmodul. Letzteres beschreibt weniger ein Modul per se, als eine Sammlung von Klassen welche die Kommunikation der Module untereinander steuert und zur Laufzeit Komponenten austauschen kann.

Änderungen oder Erweiterungen des Projektes sollten in den meisten Fällen einem dieser Module zugeordnet werden können, was also zur Folge hat dass Änderungen auch nur im jeweiligen Modul zu erfolgen haben. Wenn nun sichergestellt wird, dass die Schnittstelle des Moduls nach außen unverändert bleibt, sind die Module voneinander unabhängig, und können modifiziert werden.

Um dies zu Gewährleisten wurden die Schnittstellen der einzelnen Module in Form von abstrakten Klassen definiert und die konkreten Implementierungen als von diesen Schnittstellen erbende Klassen. Dadurch Kommuniziert das Kernmodul beispielsweise immer auf dieselbe Art mit der abstrakten Klasse Physik, ganz gleich wie die physikalische Berechnung aussehen mag.

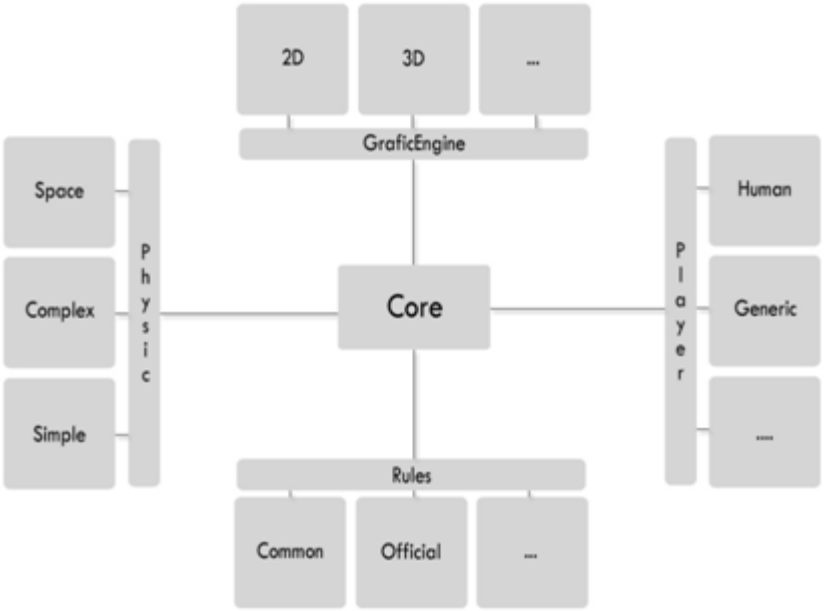

Abb. 5.1

Spiele-GUI

Entscheidungen

Bei der Umsetzung der Graphischen Oberfläche wurde entschieden, keinen WYSIWYG-Editor zu verwenden. Und zwar deshalb weil in verschiedenen Quellen im Internet erwähnt wurde, dass es danach schwieriger sei, sich in dem vom WYSIWYG erstelltem Code einzufinden. Außerdem könne es auch zu Fehlern kommen, wenn man danach Änderungen mit dem WYSIWYG-Editor vornimmt. Zudem seien die Freiheiten von WYSIWYG-Editoren recht begrenzt. Vorteil hingegen wäre die schnelle Positionierung von Elementen.

Nach Abstimmung mit den Gruppenmitgliedern wurde entschieden, bei den Layoutmanagern keine weiteren Layouts zu verwenden. Ausgenommen hiervon ist die GamepanelView-Klasse, die den BorderLayout-Manager enthält. Dabei wurde der Vorschlag aufgegriffen, die Methode setLayout() mit Paramater Null zu übergeben. D.h. es wird kein Layout verwendet. Danach verwendet man die Methode setBounds(), um eine exakte Positionierung der Elemente vorzunehmen. Es wurde folglich die genauere Positionierung der einzelnen Komponenten bevorzugt. Bei der Verwendung von Layoutmanager würden die einzelnen Komponenten größer, kleiner, bzw. würden sich verschieben, je nach Fenstergröße und Auflösung. Auch mit der Methode setPreferedSize() würde dem LayoutManager lediglich eine Größe vorgeschlagen. Außerdem könnte man, wie zuvor beschrieben, mit LayoutManagern nur ungefähr die Positionierung der einzelnen Elemente bestimmen. Die Aufteilung geschah dabei über Layouts wie GridLayout, BorderLayout, FlowLayout...

Erste Versuche

Für die Umsetzung der Graphischen Oberfläche für das Billardspiel wurde nachfolgender Aufbau überlegt.

Die Oberfläche sollte in 2 Bereiche gegliedert werden. Auf der linken Seite sollte ein 3D-Fenster erscheinen. Um das 3D-Fenster sollte ein Rahmen gelegt werden, wobei auf der rechten Seite 3 Eingabemöglichkeiten zur Verfügung stehen (siehe Grafik unten). Zum einen konnte man bei dem obersten Objekt den Stoßpunkt bestimmen (Effet-Canvas), an der die Kugel angestoßen werden sollte. In der Mitte war ein senkrechter Schieberegler für die Angabe der Kraft vorgesehen,sowie ein zusätzliches Textfeld in dem die Kraft per Eingabe exakt bestimmt werden konnte. Eine zusätzliche Überlegung war, die Anzeige der Kraft mittels eines Dreiecks (Force-Canvas) graphisch darzustellen. Schliesslich wurde beim untersten Objekt die Anzeige des Winkels in Form eines kreisförmigen Radars dargestellt. Auf dem Gradmesser des Radars sollte man die genaue Gradzahl bestimmen können.

Abb. 5.2

Bei den beiden Canvas Anzeigen von Effet, Force und Angel sollte man mit der Maus drüber fahren können.

Wenn man die Maus über den Effet-Canvas bewegt und eine Stelle anklickt, sollte der rote Punkt an dieser Stelle auf dem Ball erscheinen. Ein entsprechender Effet-Stoß wird dann später ausgeführt. Dabei entsprach der Kreismittelpunkt dem Koordinatenursprung.

Bei der Force-Anzeige sollte man den Schieberegler hoch und runterziehen, dementsprechend würde dann ein Dreieck nach oben gespannt, je intensiver das Rot am Ende des Dreiecks ist, desto mehr Kraft wurde übertragen. Als Alternative gab es noch die Möglichkeit, den Wert numerisch in einem Textfeld einzugeben.

Die Winkelanzeige liegt in einem gedachten Koordinatensystem, wobei der Kreismittelpunkt den Koordinatenursprung darstellt. Bei der Winkelanzeige unten sollte man mit der Maus an die Stelle fahren, die den Winkel bestimmt. Zwischen der Maus und dem Kreismittelpunkt wird dann eine Linie gezogen, die über den Kreis und die Gradmesser

hinausgeht. Dabei spielt die Distanz zum Mittelpunkt des Radars keine Rolle, sondern nur der Winkel. Wenn man z.B. oben rechts in der Ecke anklickt, so entspricht das einem Winkel von 45°.

Umsetzung

Während der Umsetzung war dann doch die Überlegung, ob die Force Anzeige nicht doch waagrecht erscheinen sollte. Bzgl. der Umsetzung wurde am Anfang eine Extra Klasse verwendet, in der alle Canvases enthalten waren. Ob Effet, Angle oder Force erzeugt werden sollte, wurde in einem Konstruktor über eine If-Schleife nachgefragt. In dem Konstruktor vom Gamepanel wurden alle Befehle und Methoden geschrieben, die zur Erstellung von Swing-Komponenten nötig waren. Dabei wurde zunächst versucht, die Funktionsweise ohne Modell-View zu schreiben. Dies, um zu prüfen, ob die Umsetzung wie geplant für Java-Swing möglich wäre. Grundzüge von Modell-View waren zwar bekannt, jedoch nicht, welche Methoden bzw. Befehle in der Modell-Klasse bzw. in der View-Klasse erstellt werden sollten. Bei der Methode setBounds war klar, dass es zu der View-Klasse gehören sollte. Allerdings war die Frage, ob das Erstellen von Komponenten (z.B. `new Button()`) und die Einstellungen (z.B. `setMaximum()`) nicht in die Modell-Klasse kommen sollten. Außerdem stellte sich die Frage, in welcher Klasse die Komponente gespeichert werden sollte und wie die andere Klasse die Komponente verwenden könnte, wenn die Komponenten von 2 Klassen bearbeitet würden.

Danach galt es noch, sich mit dem Thema `ActionListener` auseinander zu setzen. In dem Buch wurden 3 Methoden dargestellt. Die erste Möglichkeit bestand darin, die Methode in den jeweiligen Listener zu schreiben. Die zweite Möglichkeit war, die Methoden außerhalb des Listeners als eigenständige Methode zu schreiben und im Listener die Funktion aufzurufen. Die 3. Möglichkeit bestand darin, dem Listener ein Objekt zu übergeben. Als Beispiel wurde im Buch beschrieben, wie man dem ActionListener ein Objekt von einer Klasse Maus übergibt. Verwirrend war, dass diese Klasse direkt unterhalb der eigentlichen Klasse war. Außerdem war am Anfang nicht ganz klar, welche Listener für diese Aufgabe benötigt würden. Es stellte sich die Frage, ob eine Klasse die Aktionen mehrerer Klassen verarbeiten könnte. Falls ja, wie konnte die Klasse wissen, welche Methode zu verwenden war und welche Komponenten gerade bearbeitet wurden?

Modell-View

Bei unserer Teambesprechung wurde vereinbart, dass die Zeichnen-Klasse in mehrere Klassen aufgeteilt werden sollte die jede für sich steht. Zur Umsetzung von dem Modell View Ansatz, gab es dann ein Beispiel-Skript eines Teammitglieds, welches bei der Umsetzung hilfreich war. In dem Beispiel war auch eine Möglichkeit beschrieben, wie die Listener zu verwenden waren. Zunächst erzeugt man von der View Klasse das Modell

und übergibt dem Modell als Parameter die View Klasse selbst. Während der Erstellung der View Klasse sollte dann die Modell-Klasse als Listener übergeben werden. Es entstanden somit die Methoden `GamesPanelView`, `ShootPanelView` und `ShootPanelModel`. Dabei wurde den Modells immer das Objekt vom View übergeben, so das sich beide Klassen immer kannten. Im Konstruktur wurden keine Befehle mehr geschrieben, sondern nur noch Methodenaufrufe, wie `initComponents()`, `setComponents()`, `addComponents()` usw. Außerdem erbten alle Klassen von der Klasse JPanel. Somit musste vorher kein JPanel erzeugt werden, der die einzelnen Klassen unter sich vereinte.

ShootPanelView

Im `ShootPanelView` wurden die Komponenten relativ zueinander gesetzt, da oftmals die neue Positionierung einer Komponente eine Neupositionierung aller nachfolgenden Komponenten zur Folge hatte. Dies hatte anfangs häufig unnötigen, zusätzlichen Aufwand zur Folge, da man jedesmal alle Komponenten neu positionieren musste. Darum wurden die Positionen, sowie Breite und Höhe in Variablen gespeichert. Die Variablen wurden dann in der Methode `setBounds` verwendet. Folglich wurden vor jedem Methodenaufruf `setBounds` von einer Komponente die 4 Variablen genutzt und verändert. Alle nachfolgenden Komponenten konnten somit relativ zur vorherigen Komponente positioniert werden, indem man der Variablen einfach einen Wert hinzufügte. Danach war es auch möglich die Komponenten an anderen Stellen zu verschieben oder zu löschen, ohne alles wieder neu positionieren zu mussste.

Canvases

Außer Ballsleft hatten alle Canvases einen schwarzen Hintergrund.

Ballsleft

Ballsleft ist ein neu entstandenes Canvas, da in dem vorherigen Entwurf noch eine Anzeige zur Darstellung der eingelochten Bälle fehlte. Dieses Canvas verrät dem Spieler, welche Kugeln er versenkt hat, und ob er die halben oder vollen Kugeln einlochen muss. Es war vorgesehen 2 Canvases zu erstellen, eine für Spieler 1 und Spieler 2. Somit konnten beide sehen, welche Kugeln Sie bereits eingelocht haben, ob sie in Führung liegen und welche Kugeln sie einlochen sollten.

Bei Ballsleft wurden die Bälle mit der Zeichnenmethode erstellt. Dabei wurde bei vollen Kugeln jeder Ball durch einen Kreis dargestellt, und zwar mit der jeweiligen Farbe des Balles und der Ballnummer. Bei den halben Kugeln wurden die Kreise jeweils weiß gefüllt. Daraufhin wurde ein farbiges Reckteck über den Kreis gelegt. Um die Rundungen des Balles zu beachten, wurde links und rechts vom Ball wieder ein Kreis gezeichnet, um diese wie richtige halbe Kugeln aussehen zu lassen.

Angle

Das Angle-Canvas wurde wie oben beschrieben erstellt. Bei einem Mouse-Over konnte man bereits die Winkelanzahl einsehen. Die Anzeige wurde in Grün dargestellt. Mit einem Klick konnte zusätzlich an der Position ein weißer Strich erzeugt werden, wobei die Winkelanzeige ebenfalls in Weiß dargestellt wurde.

Bei Angle musste anhand der Positionierung der Maus der richtige Winkel bestimmt werden, da bei der Nutzung von sin/cos die Winkelanzeige nicht zwischen negativen und positiven Gradangaben unterscheidet.

Effet

Wenn man die Maus über dem Canvas bewegt, so wurde hinter der Maus ein Fadenkreuz zur Unterstützung der Mausinteraktion erstellt. Dann wurde anhand der DrawString Methode die Position bestimmt, in der sich die Maus gerade befindet. Bei einem Klick bekam die Stelle dann einen dickeren roten Punkt.

Force

Bei Force wurde das Polygon beim Zeichnen verwendet. Es handelte sich dabei um ein rot gefülltes Dreieck. Wenn man mit der Maus an einer Stelle im Canvas drückte, so wurde überprüft, ob der Klick außerhalb des möglichen Bereichs des Dreiecks lag. Wenn dem so war, dann wurde überprüft, ob der Klick links vom Dreieck oder rechts vom Dreieck war. Links bedeutet, dass keine Kraft übertragen werden sollte, während rechts die maximale Kraft übertragen wurde. Lag der Klick irgendwo innerhalb des Dreiecks, so wurde der X-Wert genommen und das Dreieck dementsprechend gezeichnet. Hier wurde allgemein die Farbe Rot für die Kraft verwendet, ohne die Stärke der Kraft zu messen.

Neue Graphik

Der bisherige Entwurf war graphisch nicht befriedigend. Deshalb wurde beschlossen, die Darstellung der Graphischen Oberfläche zunächst über Photoshop zu entwickeln. Die Einteilung des Layouts entsprach ungefähr dem des vorherigen Entwurfs.

Zunächst wurde mit der Darstellung der Kugeln begonnen. Zu diesem Zweck wurde nach Tutorials zur Darstellung von Kugeln gesucht. Schließlich wurde die folgende Seite verwendet [4]. Nach dem Tutorial wurden dann die Bälle erstmals erstellt.

Abb. 5.3 *Verwendete Kugeln für die graphischen Oberfläche*

Für die Darstellung des Queues konnte kein Tutorial gefunden werden. Daher wurde dieser selbst gezeichnet anhand von entsprechenden Bildern. Im `AngleCanvas` wurde dann nur ein Teil vom Queue dargestellt.

Der erste Entwurf sah wie folgt aus. Wie beim vorherigen Entwurf sollte auf der rechten Seite der Schuss definiert werden. Auf der linken Seite sollten die Eigenschaften der Physik dargestellt werden. In der Mitte, wo die schwarze Fläche ist, war das 3D-Billard vorgesehen. Darunter kam dann zusätzlich eine Regelanzeige, die Fouls und sonstige wichtige Informationen bereitstellt. Für die Aufteilung der einzelnen Abschnitte wurde der „Goldenen Schnitt“ beachtet. Dies sollte zu einer gleichmäßigen und angenehmen Aufteilung führen. Die Farbe Grün wurde verwendet, um dem „Look&Feel“ von einem Billardtisch zu geben. Als Schriftfarbe wurde Rot gewählt, um einen guten Kontrast zum Hintergrund zu schaffen.

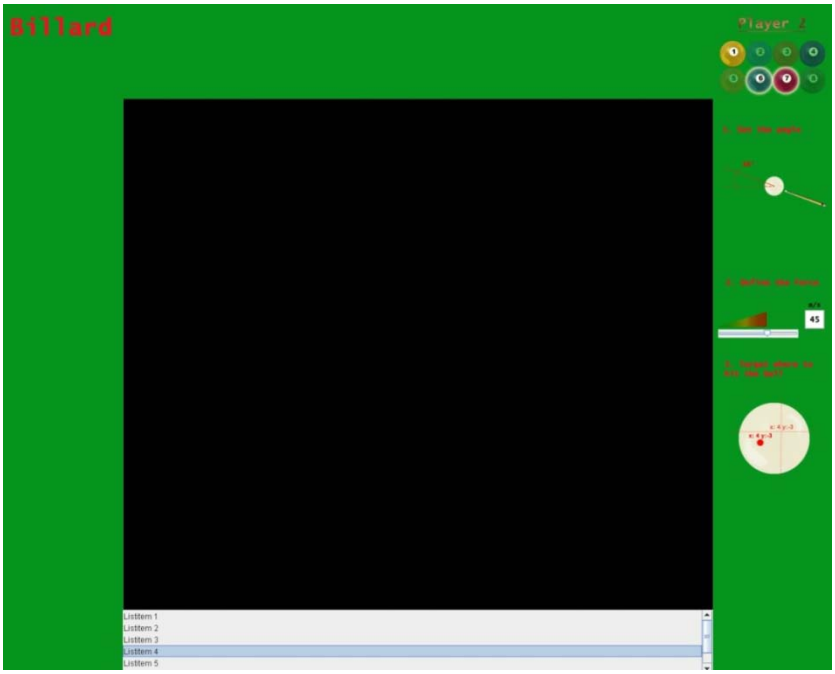

Abb. 5.4

Die Aufteilung an sich wurde als gut empfunden allerdings waren sowohl das Größenverhältnis der Kugel als auch die Tatsache, dass die Eingabe zur Bestimmung von Winkel und Effet nicht als Eingabe verstanden werden konnte. Schlussendlich kam die folgende finale Version zustande:

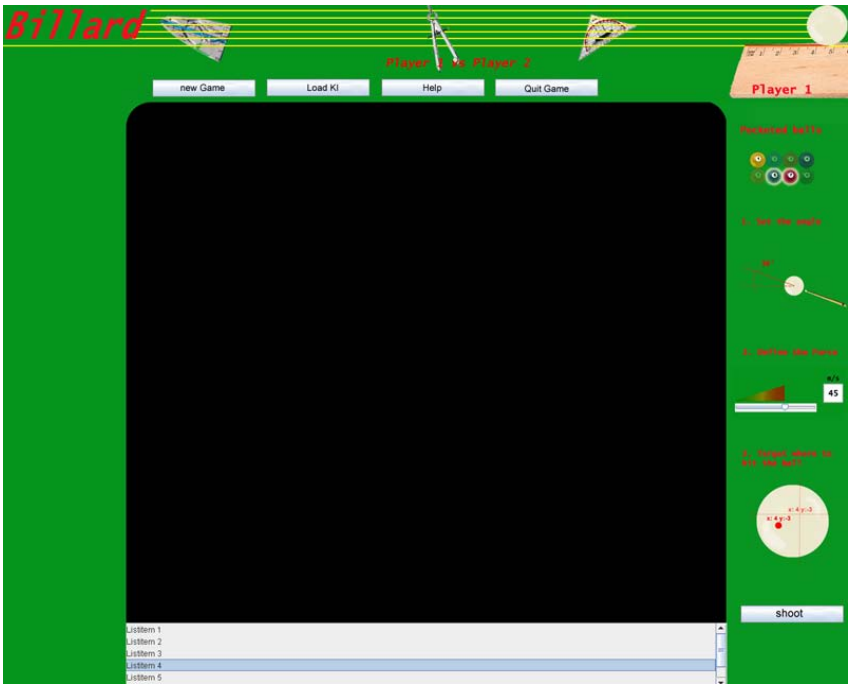

Abb. 5.5

Lineal, rote Strich, Zirkel und Geodreieck wurden nicht verwendet, da es der Gruppe ohne diese Elemente besser gefiel. Leider kann man an dem obigen Bild nicht erkennen, dass an den Seiten ein Rand vorhanden ist. Dieser soll einen etwas moderneren, abgehobenen Effekt erzielen und dem Ganzen eine etwas realistischere Fläche geben. Der Schriftzug Billard sollte mehr Aufmerksamkeit bringen. Deswegen wurde die Schrift kopiert, in den Hintergrund gestellt und etwas transparent gemacht. Die Schrift wurde dann noch schräg gestellt, als ob sie von der schnellen Kugel rechts zur Seite geschoben wurde. Die gelben Linien symbolisieren Luftströme, die durch die rasche Bewegung der Kugel entstanden sind.

Bei dem Winkel gab es noch die Idee, wieder eine Radar-Form zu verwenden. Dabei wurde nach passenden Tutorials zu Erstellung des Radars gesucht. Man entschied sich jedoch schließlich dagegen, da diese Art der Bedienung nicht unbedingt jedem verständlich ist. Stattdessen wurde eine Kugel verwendet, welche die weiße Kugel auf dem Tisch darstellen soll. Durch das kreisförmige Bewegen des Queues kann dann die Richtung des Schusses bestimmt werden. Dabei wird angezeigt, wie viel Grad gerade eingegeben wurden. Hier kam dann Java2D zum Einsatz, bei dem das Rotieren von Objekten ermöglicht wurde. Dabei wurde immer die komplette Graphik zum Koordinatenursprung gedreht.

Die Anzeige der Kraft, sowie der Effet wurden aus der vorherigen Version mehr oder weniger übernommen, wobei diese graphisch etwas aufgewertet wurden.

Bei der Schrift ("Player 1") im Lineal, sollte dann der Spieler geschrieben werden, der

gerade dran ist. Die Bälle die auf dem Tisch waren, sollten leicht transparent dargestellt sein. Wenn neue Bälle eingelocht wurden, tauchte hinter den eingelochten Bällen ein Schein auf und die Kugel war nicht mehr transparent. Wenn der nächste Spieler dran war, verschwand der Schein wieder.

Anstatt einer Menüleiste war vorgesehen, Buttons zu erstellen. Je nach Größe der Menüleiste konnten einzelne Elemente hinter der OpenGL Darstellung verschwinden. Denn OpenGL programmiert direkt auf der Graphik. Allerdings beachtet es ein vor sich stehendes Fenster.

Die Rundungen der Ecken am 3D-Fenster wurden bevorzugt, da es insgesamt doch einen angenehmeren Eindruck vermittelte. Die Rundungen an dem OpenGL Fenster konnte dann aber nicht umgesetzt werden aus den oben genannten Gründen.

Als Button war der Standard-Java Button vorgesehen. Dieser war sowohl für den "Shoot"-Button, den Button "New Game", "Load KI", "Help" und den Button "Quit Game" vorgesehen.

Zusätzlich kam dann die Regelanzeige, welche zur Darstellung von Regeln und sonstigen wichtigen Informationen fungieren soll. Dies ist eine Alternative zum nervenden Dialogfenster, welches bei einem Foul erscheint.

Umsetzung in Java

Der Aufbau anhand des Borderlayouts im GamePanelView war folgendermaßen geplant:

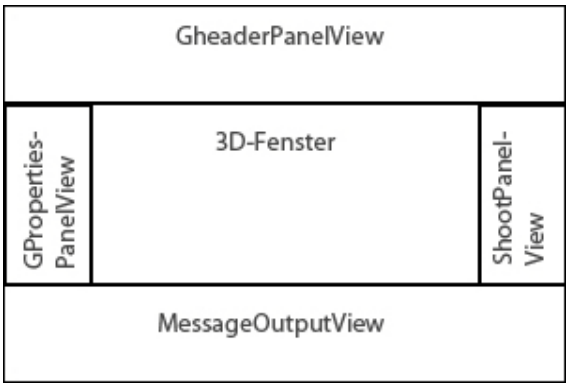

Abb. 5.6

Zur Umsetzung in Java wurde noch eine Library Datei verwendet, den sogenannten `ImageLoader`. Diese Library sollte dabei helfen, Bilder problemlos einzubinden, die später von der Jar Datei benötigt werden.

Die Bilder wurden nun zur Umsetzung der Oberfläche geschnitten und eingebunden. Hier war die genaue Position von Breite und Höhe zu bestimmen.

Sie wurden anfangs in `GamePanelView` erstellt. Allerdings wurden diese später teils in den Unterklassen "`ShootPanelView`", "`GPropertiesPanelView`", "`Gheader-panelView`" etc. erzeugt. Für die Angaben der genauen Positionen wurde eine zentrale Klasse verwendet, mit deren Hilfe man die Position von Klassen und Bildern bestimmen konnte. Auf diese wird später noch genauer eingegangen.

Somit wurde die erste Version um einige Befehle erweitert oder umgeändert.

Die Umsetzung der physikalischen Gesetze bei Vorläufer und Rückläufer war schwieriger, als es zunächst erschien. Deshalb wird bei dem Effet nur die X-Achse verwendet. Die Graphik wurde deshalb umgeändert, so dass es nur noch eine horizontal verlaufende, unterteilte X-Achse gibt. Die Bewegung der Maus wird nur noch auf dieser X-Achse durch eine vertikale Linie dargestellt. Diese beschreibt, wo die Kugel auf der X-Achse angestoßen wird.

Zu den Canvases für die Bestimmung des Winkels und des Effets sollten noch Textfelder erstellt werden. Damit ist eine genaue Eingabe anhand von Zahlen möglich. Um Änderungen in den einzelnen Canvas-Klassen auch in den Textfeldern zu übernehmen, werden zusätzliche Methoden benötigt. Diese Methoden heißen `setAngle()`, `set-Force()` und `setEffet()` und werden in dem `ShootPanelModel` realisiert. Zur Kommunikation zwischen der 3DEngine und der graphischen Oberfläche musste eine Schnittstelle zwischen den beiden Klassen geschaffen werden. Beim Eingeben über die graphische Oberfläche musste `ShootPanelModel` erkennen, um was für Eingaben es sich handelte. Nach dem Erkennen der Eingabe wurde in der 3DEngine.GuiAdapter entweder die Methode `changeForce()`, `changeAngle()` oder `changeEffet()` aufgerufen.

Diese Klasse sollte die für die 3DEngine wichtigen Befehle aufrufen und dann wieder zum Adapter der graphischen Oberfläche springen. Hierbei wird wieder die entsprechende Methode aufgerufen, welche die einzelnen Komponenten der graphischen Oberfläche ändert.

Bei dem Canvas-Ballsleft wurde über 2 Möglichkeiten gesprochen. Die erste Möglichkeit war 2 `Canvases` zu erzeugen, in der zu jedem Spieler die Bälle angezeigt werden, die er eingelocht hat. Bei der 2. Variante sollen alle Bälle angezeigt werden die auf dem Tisch sind. Die eingelochten Kugeln werden dabei transparent dargestellt, während die Bälle auf dem Tisch normal dargestellt werden. Zusätzlich soll neben der Anzeige des Spielers auch graphisch verdeutlicht werden, ob dieser die halben oder vollen Kugeln zu treffen hat. Wir hatten uns als Gruppe für die 2. Variante entschieden.

Bei der Klasse für die Regelausgabe sollte es möglich sein, die Farbe und die Schriftart zu ändern. Technisch wurde dies mit einer `JList`, dem `ListCellRenderer` und

einem Vektor realisiert. Hierzu musste eine eigene Klasse erstellt werden. Der `ListCellRenderer` war nötig, da es ansonsten nicht möglich wäre, Schriftfarbe, Schriftart usw. zu verändern. In der `ListCellRender` konnte man seine eigene `JList` erstellen. Hier gäbe es weitere Möglichkeiten, wie das Hinzufügen von Bildern z.B. mit Icons!

Weiter bedurfte es eine Klasse `TexttoGuiAdapter`, um die Regelanzeige in der Liste auszugeben. Hierfür wurden Methoden bereitgestellt, die von außen aufgerufen werden konnten und zu einer Ausgabe in der Liste führten.

Zur zentralen Steuerung und dem Bereitstellen von gemeinsamen Methoden wurde die Klasse GamePanelController erstellt. Der Vorteil war, dass man jetzt direkt auf wichtige Klassen zugreifen kann.

Methoden, die in dieser zentralen Klasse definiert wurden, konnten von mehreren Klassen verwendet werden. Z.B. Methoden, welche Breite und Höhe des Fensters zurückgaben. Somit wurde sichergestellt, dass bei Breiten- oder Höhenangaben nur ein Befehl geändert werden musste. Außerdem wurde der GamePanelController nach dem `GamePanelView` erstellt. Der `GamePanelController` hat dann alle weiteren Klassen erzeugt, die im `BorderLayout` vorkamen. Und zwar sowohl die Modell als auch die View-Klassen. Somit konnte später immer auf die Controllerklasse zugegriffen werden, um dann auf andere Klassen zu gelangen. Vorher war das eher umständlicher, da man immer über den `GamePanelView` in mehrere Klassen springen musste, um dann letztendlich die gewünschte Klasse zu erreichen.

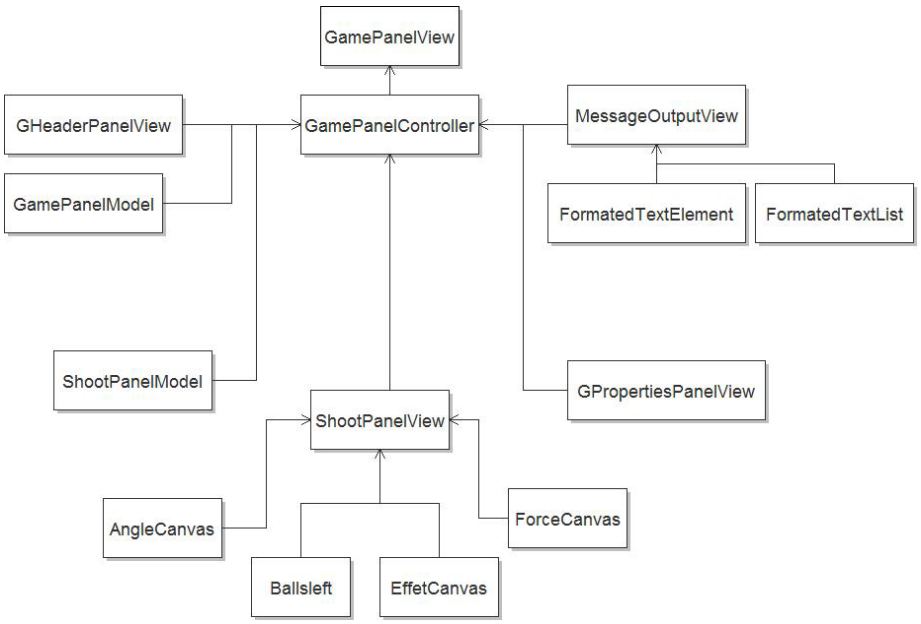

Abb. 5.7 *Das neue Klassendiagramm*

Durch die Regelliste verschob sich der untere Bereich nach oben. Ein Problem war, dass die beiden Klassen links und rechts wegen der Aufteilung eines `BorderLayout` nicht bis an das obere bzw. untere Ende des Fensters ragten. Um den Platz trotzdem noch auszunutzen, wurden die Elemente in dem `ShootPanelView` etwas zusammengedrückt. Allerdings war der rechte Bereich im `GHeader` und `GRulesPanelView` leer, während im `ShootPanelView` die Komponenten eher zusammengedrückt waren (siehe Grafik zum Aufbau vom BorderLayout oben). Von der Logik wurde dann gesagt, dass der `CanvasBallsleft` auch in die obere Klasse durfte, da diese keine Eigenschaft von `ShootPanelView` war. Allerdings wurde auch der Button unten in `GRulesPanelView` gestellt. Dies ließ sich allerdings nicht mit der Logik vereinbaren. Deshalb wurde das Layout geändert. In dem neuen Layout enthielt das `BorderLayout` keine `SouthKomponente` mehr. Stattdessen wurde die Regelanzeige in der Mitte vom Borderlayout unterhalb des 3D-Fensters erzeugt. Somit ragte der `ShootPanel` und `GPropertiesPanel` ganz nach unten bis zum Ende des Fensters.

Ein weiteres Problem war der Header der graphischen Oberfläche. Hier wurde die Paint-Methode und das Erstellen von Swing-Komponenten gleichzeitig verwendet. Dies führte jedoch insofern zu Problemen, als merkwürdigerweise die Buttons verschwanden. Diese erschienen erst wieder, wenn man mit der Maus über den Bereich fuhr, wo ein Button sein sollte. Vermutet wurde, dass Swing zum Zeichnen seiner Komponenten ebenfalls die Paint-Methode verwendet. Dies hatte zur Folge, dass beim Zeichnen mit der Paint-Methode über die Swing-Komponenten gezeichnet wurde. Ein Versuch die `PaintMethode` vor den anderen Methoden zu verschieben hatte änderte nichts, da festgestellt wurde, dass die Swing-Komponenten im Konstruktor erstellt waren. Dabei nutzte die Paint-Methode den Bereich nicht, in dem die Buttons erstellt waren. Beim Mausover erschien der Button wieder. Dies lässt sich damit erklären, dass beim Überfahren eines Buttons mit der Maus der gelbe Innenrahmen erschien. Somit musste dieser Button nochmals gezeichnet werden. Eine Möglichkeit wäre gewesen, die Methoden im Konstruktor nochmals aufzurufen. Allerdings wäre dies sicherlich nicht die ideale Lösung. Hilfreich war es, den Befehl `super.paint(g)` zu verwenden. Dies behob zumindest die Probleme bei den anderen Klassen. So z.B. in der Klasse `ShootPanel`, wo die erzeugten Ränder über den Canvases und den Buttons gezeichnet wurden. Eine weitere Lösung hätte darin bestanden, die Zeichnungen im GHeader in einem Extra Canvas zu zeichnen.

Weiter hatte die Methode `getWindowwidth()` und `getWindowheight()` nicht die richtigen Werte zurückgegeben. Die Werte stimmten nicht mit denen des Monitors überein. Später wurde vermutet, dass beim Programmieren der Monitor an einem Laptop angeschlossen war. Dies hatte vermutlich zur Folge, dass 2 verschiedene Auflösungen vorhanden waren. Zur besseren Positionierung mussten bei der Breite 5 Pixel und

bei der Höhe 48 Pixel abgezogen werden. Die Pixelangaben kamen durch Ausprobieren zustande, da wir wegen der Präsentation unter Zeitdruck waren.

Ein größeres Problem war noch, dass sowohl Bilder, Canvases und alle anderen Swing-Komponenten eine feste Größe hatten. Allerdings war die Aufteilung des Layouts relativ zur Auflösung des Bildschirms. Aufgrund der eingestellten Auflösung des Benutzers wurde der goldene Schnitt berechnet, und dieser zur Aufteilung des Layouts verwendet. Dahinter stand der Gedanke, zu jeder x-beliebigen Auflösung ein angenehmes Layout zu erstellen. Zumal bei der Erstellung der Graphischen Oberfläche zuerst mit einem alten Monitor (Auflösung 1024x768) gearbeitet wurde. Allerdings wurde dieser dann defekt und wurde schließlich durch einen neuen Monitor (Auflösung 1900x1080) ersetzt. Dies führte dann aber insofern zu Problemen, als kleinere Auflösungen nicht beachtet wurden. Folglich ragten einzelne Komponenten aufgrund ihrer Größe, über den Monitor hinaus.

Bei der Schrift im ShootPanel wurde dann später doch auf Schwarz umgestellt, da die Farbe nicht gefiel und man Schwarz auch bei den unterschiedlichsten Farben ohne Probleme verwenden konnte.

Bei den Buttons hatte man später entschieden, doch noch eine Menüleiste zu erstellen, da oben noch genügend Platz war und die Menüleiste nicht so viele Items enthielt. Somit konnte oben das Menü ausgeklappt werden, ohne dass diese unter dem OpenGL Fenster gerieten.

Auf der linken Seite fiel der Bereich mit den Physikparametern weg, da diese über das Fenster "new Game" bestimmt wurden. Außerdem war man der Meinung, dass während dem Spiel die Parameter nicht geändert werden dürften.

Die Methoden `getIdealWidth(), getIdealHeight(), getWindowwidth()` und `getWindowheight()` wurden sehr oft verwendet. Zum einen, um die Ränder (Bilder) am OpenGL-Fenster zu zeichnen. Z.B. musste beim Header erst bestimmt werden, wo der obere Rand des OpenGL-Fensters anfing und aufhörte. Zum anderen, um die Größe der einzelnen Klassen im Borderlayout festzulegen. Wie oben beschrieben gab es allerdings einen Konflikt zwischen der relativen Positionierung des Layouts und der absoluten Größen von Swing-Komponenten und Bildern. Deshalb wurde kurzfristig für das Layout eine feste Größe vorgegeben. Unabhängig von der Auflösung des Users wird zum Aufbau des Layouts eine Auflösung von 1024x768 genommen. Damit können alle Komponenten mit festen Werten positioniert werden. Ein weiteres Argument für feste Größe war, dass sich die Layouts bei hoher Auflösung vergrößerten, die Komponenten jedoch nicht. Dies ergab viel "Freiraum" in den einzelnen Layout-Bereichen.

Sowohl die Breite der JPanel-Klassen als auch der Bilder war nicht immer gegeben. Deshalb wurden die oben genannten Methoden des Öfteren benötigt. Die Größe einer Klasse, welche von `JPanel` erbte, war nicht immer zur Laufzeit ermittelbar. Dasselbe galt auch für Bilder. Bei manchen Bildern konnten die Eigenschaften zur Ermittlung von Breite und Höhe des Bildes innerhalb der `drawImage` Methode verwendet werden. Bei anderen wurden sie noch nicht definiert und enthielten den Wert `NULL`.

Grafikengine

Anforderungen an die Engine

Die visuelle Repräsentierung des Billardtisches und der darauf befindlichen Kugeln ist von zentraler Bedeutung sowohl für die Benutzung des Programmes, sowie für dessen Entwicklung. Es leuchtet ein dass der Entwicklungsprozess im Bereich der Physik speziell im Anfangsstadium erschwert wird, wenn als einziges Werkzeug zur Verifizierung der verwendeten Formeln Konsolenausgaben verwendet werden können, insbesondere deshalb weil die Menge der Berechnungsschritte sehr hoch ist. Daraus folgt also dass schon zu Anfang des Projekts eine simple, jedoch stabile Repräsentierung des Tisches existieren muss um das Arbeiten an den anderen Modulen zu gewährleisten. Wichtig ist dabei ebenfalls dass neben der herkömmlichen Interaktion mit dem Tisch (Schlagen einer Kugel) auch Mechanismen zu Verfügung gestellt werden um die Entwicklung der Spielregeln und der Physik zu vereinfachen. Diese Mechanismen umfassen in unserem Fall das hinzufügen, entfernen und verschieben beliebiger Kugeln zur Laufzeit.

Seitens des Anwenders sind diese Anforderungen unerheblich, da hier das Augenmerk auf einer schönen Visualisierung und einer einfachen, verständlichen Usability des Spiels liegt. Dies ist jedoch problematisch, da eine hübsche Grafik mit einem erheblichen Mehraufwand in Design, Technik und Aufbau verbunden ist, was sich auf die Entwicklungszeit dieses elementaren Bestandteils des Programms auswirkt.

Umsetzung

Es leuchtet ein dass eine simple Engine zur Entwicklung des Programmes im Anfangsstadium des Projekts wesentlich höher zu Priorisieren ist als eine Engine welche für den User optimiert sein soll. So wurde entschieden zuerst eine einfache Engine zu schreiben, welche im Laufe der Zeit dann einer ansprechenderen Engine weichen sollte.

Um zu verhindert dass beim späteren Austauschen der Grafikengine keine Komplikationen durch Interdependenzen der abhängigen Module entsteht, wurde beschlossen die Grafikengine als abstrakte Schnittstelle `GraphicEngine` zu definieren und für die un-

terschiedlichen Anforderungen einzelne, spezielle Grafikengines zu programmieren welche auf dieser Aufbauen und nur über diese Angesprochen werden können.

Neben den unmittelbaren Vorteilen für die Entwicklung resultiert daraus ebenfalls die Möglichkeit die `GrafikEngine` in späteren Phasen des Projekts an die noch unbekannten Anforderungen und Features zu adaptieren welche möglicherweise noch entstehen können (Spiel als verteilte Anwendung, pipes in Flash-GUI, etc.)

Für die simple Engine wurde die Bezeichnung `GraphicEngine2D` verwendet, was dadurch begründet ist, dass sehr früh im Projekt klar war dass die finale GUI für den Andwender eine 3D Grafik beinhalten würde, und den Namen `GraphicEngine3D` tragen wird. Auf diese beiden Engines soll nun etwas genauer eingegangen werden.

GraphicEngine2D

Die Umsetzung der 2D Engine wurde erfolgte durch AWT Toolkit, welches in der Java Standartbibliothek enthalten ist. Der Grund für dessen Verwendung war dass dieses Toolkit alles Nötige für das erstellen einer simplen Grafikengine mitliefert und es dadurch dass es Teil der Standardbibliothek ist ausgesprochen gut Dokumentiert ist, was wiederum einen positiven Einfluss auf die Entwicklungszeit hat. Desweiteren ist mit dieser Bibliothek garantiert dass die Anwendung Plattformunabhängig ausgeführt werden kann, auch wenn diese Unabhängigkeit nicht verhindert dass das Verhalten der Engine von Plattform zu Plattform variiert. So sollte an dieser Stelle erwähnt werden dass die Performance der Anwendung vom benutzten Betriebssystem abhängt und die Anwendung trotz Double Buffering bei 25 gezeichneten Bildern pro Sekunde unter Windows und Linux zu flackern beginnt. Erklärt kann dies unserer Meinung nur dadurch werden, dass zum einen der AWT-Canvas im Gegensatz zu normalen AWT Komponenten nicht direkt an das Betriebssystem und infolge dessen nicht direkt an die Grafikkarte übergeben wird, sondern „Software“ gerendert wird. Ein weiterer Grund für die unterschiedliche Geschwindigkeit auf den Plattformen ist, dass ein AWT Objekt über das Betriebssystem aufgefordert werden kann sich selbst neu zu zeichnen, was durchaus sinnvoll ist wenn beispielsweise ein Fenster vom User in seiner Größe verändert wird. Unglücklicherweise geschehen diese Aufrufe aus dem System bei Linux und Windows häufiger als auf einem MacOSX Rechner, was sich wiederum auf die Performance auswirkt.

Die Performance war und ist das größte Problem mit der GraphicEngine2D und konnte trotz Implementierung eines Algorithmuses zum automatischen Clipping von unveränderten Bereichen auf dem Tisch nicht behoben werden, was auch dazu führte dass die GraphicEngine2D nur noch für den internen Gebrauch verwendet wird und diese als etwaige Performance Alternative zur GraphicEngine3D nicht verwendet wurde.

GraphicEngine3D

Java hat im Allgemeinen den Ruf eine 3D feindliche Programmiersprache zu sein. Und diese Kritik ist alles andere als unbegründet. Während für C++ eine unzählbare Anzahl an out-of-the-box 3D Engines und APIs existieren, ist das Angebot für Java sehr rar.

So zeigte sich schnell dass für uns unter den gegebenen Umständen nur 3 Möglichkeiten in Frage kamen: das manuelle Programmieren der Engine in LWJGL/JOGL, oder die Verwendung einer 3D API. Hierbei kamen nur die API's JME2 oder OGRE4J in Frage.

LWJGL / JOGL

LWJGL steht für Lightweight Java Game Library, während JOGL für „Java Bindings for OpenGL“ steht. Beide Bibliotheken ermöglichen es dem Java-Programmierer auf die C-Funktionen von OpenGL zuzugreifen, womit es möglich ist 3D Programme unter Java zu schreiben welche nahezu die selbe Geschwindigkeiten haben wie native, in C/C++ geschriebene OpenGL Anwendungen.

Leider hat die Verwendung von LWJGL oder JOGL zur Folge dass der Programmierer seine 3D Engine komplett von Grund auf selbst schreiben muss. Dies ist nicht weiter Hinderlich wenn nur eine sehr simple graphische Ausgabe erfolgen muss, sobald man jedoch den Anspruch hat eine ansprechende GrafikEngine zu verwenden müssen, muss neben Beleuchtung und Schattenwurf, Tiefenverzerrung, Texturierung, Culling, Szene-Graph, UV-Mapping, ein Importer für 3D-Modelle, Shader und vieles mehr eigenhändig Programmiert werden.

Für unsere Zwecke ist dieser Aufwand enorm hoch, und es wurde beschlossen das Rad nicht neu zu erfinden, sondern eine vorgefertigte API zu benutzen, welche all dies und mehr bereits beherrscht.

OGRE4J

Hinter Ogre4J versteckt sich ein Projekt welches über das Java Native Interface den Zugriff auf die meistgenutzte C++ Grafik- und Spieleengine Ogre3D bietet. Ogre steht für Obect Oriented Graphics Rendering Engine und wird von einer sehr großen Community eingesetzt und weiterentwickelt. Die Dokumentation ist ausgesprochen gut und im Netz und der Bücherhandlung findet man Hilfe bei Problemen. Hinzukommend wurde Ogre bereits von den Programmieren im Rahmen einer Hochschulveranstaltung und Grundzügen erlernt, womit sich Ogre4J als ideal für das Projekt erwiesen hätte.

Das Problem ist jedoch das Java Binding von Ogre. Dass sich dieses zum Zeitpunkt des Projekts noch in einer unvollständigen Beta-Phase befand war nicht der ausschlaggebende Punkt weshalb das Team sich gegen Ogre entschieden hat.

Ogre bietet dem Programmierer ausgesprochen viele Möglichkeiten, was sich leider in der größe der Bibliotheken wiederspiegelt. Dies wäre nicht unbedingt tragisch wenn unser Programm nur für eine Plattform konzepiert wäre, da in diesem Falle nur die

Binaries für die benutzte Plattform mitgeliefert werden müsste. Da unser Programm jedoch auf Linux, Windows und Mac funktionieren muss, müssten die 3 Binaries für jede Plattform kompiliert und mitgeliefert werden, wobei noch Kompatibilitätsprobleme im Bezug auf x32 und x64 Architekturen nicht berücksichtig werden.

Dies ist sehr zu bedauern, da Ogre in sonst allen andern Punkten glänzt und die konkurrierenden APIs Ogre kaum was Wasser reichen können.

JME2

Die JMonkey Engine 2 ist eine Java API, welche versucht dem Programmierer eine Alternative zu Ogre und ähnlichen APIs auf Javabasis zu bieten. Zur Kommunikation mit der Grafikkarte kann es sowohl LWJGL als auch JOGL verwenden und ist somit bis auf die native Bibliothek des Rendersystems vollständig Plattformunabhängig. Desweiteren ist es die einzige Nennenswerte Engine für Java, weshalb das Team sich letztendlich für Jme2 entschieden hat.

Zu Jmonkey muss gesagt werden dass es für Java Programmierer die beste Möglichkeit darstellt eine 3D Anwendung zu schreiben ohne sich selbst um die Engine kümmern zu müssen. Es stellte sich jedoch im Laufe des Projektes heraus dass Jmonkey sehr schlecht Dokumentiert ist, und speziell beim suchen von Beispielen, Lehrmaterialien und Hilfestellung man oft kaum oder nicht fündig wurde. Dies war beispielsweise Störend als versucht wurde das gerenderte OpenGL Fenster in ein Swing-Fenster einzubetten. Dies ist ein Szenario welches bei Java-Programmierern eigentlich sehr häufig auftreten müsste, da ein Großteil aller Java-Programme Swing für die Graphische Benutzeroberfläche verwendet. Doch allein hierfür musste sehr lange und ausgiebig in zahlreichen Internetforen gesucht werden um halbwegs brauchbares Material zu finden. Dies ist bedauerlich, da Jme hinsichtlich der mangelnden Konkurrenz durchaus potential hat. Es bleibt also abzuwarten ob es Jmonkey im Laufe der Zeit schafft eine größere Community auszubilden, bisher kann jedoch festgehalten werden dass sich Jmonkey in einem noch unreifen Stadium befindet.

Gestalterische Werkzeuge

Der Grund für die Verwendung einer 3D Engine war eine attraktivere Visualisierung des Tisches zu ermöglichen. Die 3D Engine alleine reicht hierfür jedoch nicht aus, da sie zwar Wege liefert texturierte 3D-Modelle darzustellen, diese jedoch mit einer Engine nicht erstellt werden können. Hierfür bedarf es einer Software mit welcher die zu Zeigenden Objekte modelliert und texturiert werden können. In unserem Falle wurde „Autodesk Maya 2008“ verwendet.

Entwicklung des Tisches

Bei der Modellierung eines Objektes für ein Computerspiel gibt es im Gegensatz zu einem Objekt für ausgerendertes Video oder Bild einige Einschränkungen welche es zu beachten gilt. Jmonkey stellt hierbei keine Ausnahme dar und man kann verallgemeinert sagen dass diese Einschränkungen bis auf wenige Ausnahmen auf alle 3D Engines und API's zutreffen.

So muss es sich beim modellierten Mesh um ein reines Polygon Mesh handeln, da Nurbs- und Subdiv-Flächen von Spieleengines aufgrund ihrer komplizierten und dadurch langsamen Berechnung nicht unterstützt werden. Dabei ist zu beachten dass die Anzahl der Polygone möglichst klein bleibt, da sonst die Performance des Spieles abnimmt. Wenn mehrere Meshes exportiert werden sollen muss darauf geachtet werden dass diese in einzelne Dateien aufgeteilt werden, da das exportieren einer Gruppe von Meshes für gewöhnlich zu fehlerhaftem Laden in der Spieleengine oder falscher Zuordnung von Materialien und Texturen führt.

In unserem Falle wurde der Tisch mit 640 Polygonen modelliert, und eine kubische Umgebung mit 6 Polygonen. Um das kantige Modell des Tisches etwas abgerundeter erscheinen zu lassen wurden die Normalen der Polygone abgeflacht um zu erwirken dass das Modell abgerundeter wirkt, ohne die Zahl der Polygone weiter zu erhöhen.

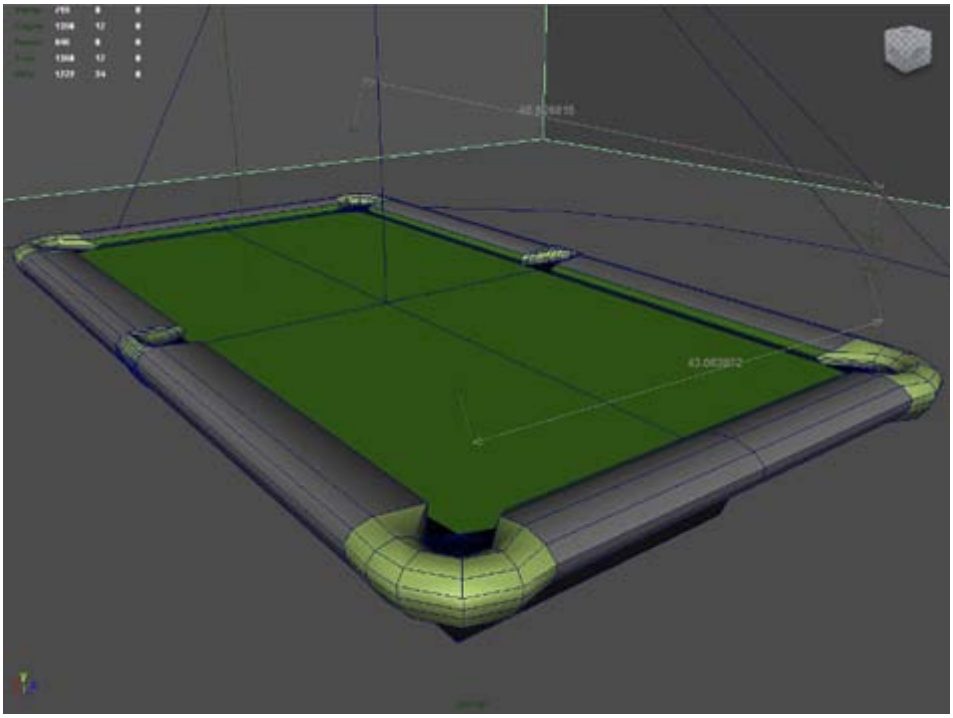

Abb. 5.2

Die Einschränkungen beim Modellieren sind leicht zu berücksichtigen und sind selten der Grund für Probleme. Schwieriger gestaltet es sich die Texturierung des Meshes.

Für gewöhnlich wird ein Polygon Mesh mit einer Vielzahl verschiedener Materialien und Texturen bedeckt, welche einzeln Positioniert und Konfiguriert werden können. Dadurch ist gewährleistet dass beispielsweise das Vlies des Tisches Matt und mit einer Vlies Textur bezogen werden kann, während die Holzumrandung des Tisches leicht schimmert und mit einer Holztextur bedeckt ist.

Im Bereich der Echtzeitanwendungen gilt jedoch die Einschränkung dass ein Mesh immer nur einen Shader und eine Textur besitzen kann, welche über die UV-Koordinaten im Mesh auf das Mesh gemappt wird. Zwar existieren inzwischen Engines die mehrere Texturen und Shader pro Mesh unterstützen, diese bilden jedoch die Ausnahme, und Jmonkey gehört nicht zu dieser Gruppe.

Um diese enorme Einschränkung zu Umgehen wurde vom Modell zuerst eine UV-Map erzeugt, also das Mesh wurde manuell „auseinander gefaltet“ und auf eine 2D Fläche Projiziert. Im Anschluss wurden die einzelnen Texturen den jeweiligen Polygonen des Meshes zugewiesen und zu einer einzelnen Textur gerastert.

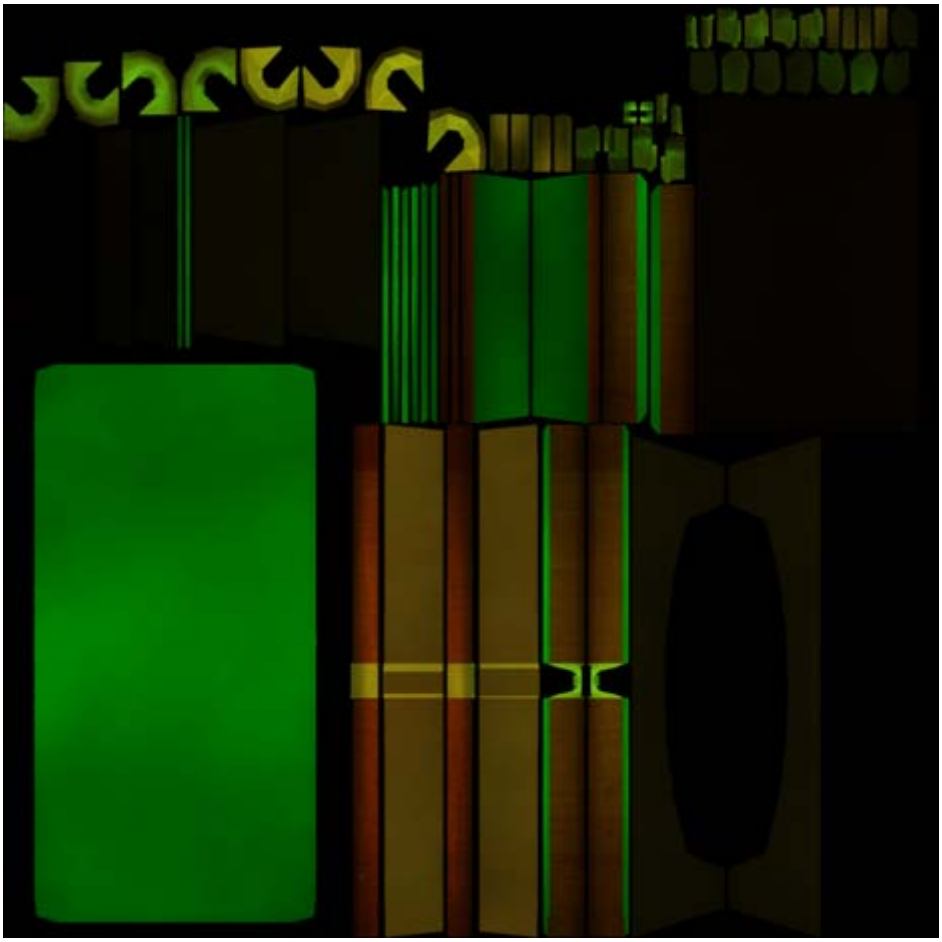

Abb. 5.3 *UV-Texturemap des Tisches*

Die letzte Hürde waren Licht und Schattenwurf. Während eine 3D Software für die exakte Berechnung der Beleuchtung und der daraus resultierenden Schatten für ein Bild mehrere Stunden rechnen kann, müssen in einer Echtzeitanwendung die Schatten mindestens 25mal pro Sekunde berechnet werden. Aus diesem Grund beschränken sich Grafikengines normalerweise auf Schattenberechnung durch Raytraceing oder Depthmaps, welche sich schnell Berechnen lassen, jedoch dementsprechend unnatürlich wirken.

Um schöne und hochperformante Schatten für unseren Tisch zu erhalten wurden der Tisch in Maya ausgeleuchtet und mit „MentalRay“, einem professionellen Renderer, vollständig ausgerendert. Das Berechnen der Beleuchtung benötigte ungefähr 20 Minuten. Das Resultat dieses Rendervorgangs wurde dann auf die Textur des Tisches und auf den Umgebenden Würfel „gebaked“, also auf die Textur gebrannt. Mittels dieses Tricks konnte nun in Jmonkey ein Tisch dargestellt werden, der realistisch ausgeleuchtet war, ohne die Beleuchtung über die Engine durchzuführen, da sämtliche Helligkeitsinformationen sich in den Farbinformationen der Textur befanden. Das Resultat war also ein bestens beleuchteter Tisch, sowie eine bessere Performance aufgrund des Einsparens der Licht- und Schattenberechnung Seitens der Grafikengine.

Abb. 5.4

Textur des Fußbodens. Der schwarze Fleck in der Mitte ist der Schatten des Tisches

Physikengine

Wie schon oben skizziert (Abb. 5.1) ist der Aufbau der Physik modular. Die abstrakte Klasse `PhysicEngine` enthält die nötigen Methoden die für eine Implementation einer beliebigen Physik notwendig sind. Außerdem stehen dem Package eine Bibliothek mit allgemeinen physikalischen Formeln sowie die Klasse `Cushion`, die ein Bandenobjekt zur Laufzeit repräsentiert, zur Verfügung.

Um eine einheitliche Kommunikation mit der Physik und ihren Ablegern sicherzustellen wurde die Klasse `PhysicParameter` geschaffen die z.B. von den Packages `GraphicEngin3D` und `Player` benutzt wird.

Logikengine

Um einen leichteren Einstieg in das Spiel zu erreichen, wurde darauf verzichtet, die offiziellen Regeln zu implementieren. Stattdessen hat sich das Team für einen Regelsatz entschieden, der von vielen Freizeitspielern und Personen ohne Billarderfahrung leichter zu verstehen ist.

Umgesetzte Regeln

Ein Spieler bleibt am Zug, solange er mit jedem Stoß mindestens eine seiner Kugeln versenkt ohne ein Foul zu begehen.

Foul

Wenn ein Spieler ein Foul begeht, hat der Gegenspieler ‚Ball in Hand'. Der Gegenspieler kann die weiße Kugel an eine beliebige Stelle des Tisches legen und von dort aus in eine beliebige Richtung weiterspielen. Ein Foul ist das Einlochen der weißen Kugel oder wenn die weiße Kugel zuerst auf die Acht trifft. Sobald der Tisch nicht mehr offen ist, ist es auch ein Foul, wenn weiße Kugel zuerst auf eine gegnerische Kugel trifft.

Offener Tisch

Spielen auf dem offenen Tisch bedeutet, dass jeder Spieler auf halbe und volle Bälle spielen kann. Der Tisch ist nicht mehr offen, sobald ein Spieler nach dem Anstoß eine farbige Kugel ohne ein Foul zu begehen versenkt. Der Spieler spielt von nun an auf die Art von Kugel, die er versenkt hat. Hat er sowohl Halbe als auch Volle beim Stoß eingelocht, kann er bestimmen auf welche er weiterspielen will.

Spielende

Sind alle Halben oder Vollen versenkt worden, kann der Spieler mit den jeweiligen Kugeln versuchen, die Acht einzulochen. Die erste Kollision der Weißen mit der Acht ist

jetzt kein Foul mehr. Versenkt er die Acht ohne ein Foul zu begehen, hat er gewonnen. Begeht er beim Einlochen der Acht ein Foul, hat er verloren.

Sonderregeln für den Anstoß

Wird beim Anstoß eine farbige Kugel versenkt, darf der Spieler einen neuen Stoß ausführen. Es werden ihm jedoch keine halben oder vollen Bälle zugeordnet. Das Einlochen der schwarzen Kugel ist kein Foul. Der Spieler muss aber einen neuen Anstoß machen. Werden jedoch die Schwarze und die Weiße beim Anstoß versenkt, darf der Gegner den neuen Anstoß ausführen.

Aufbau des Packages *gameLogic*

Das Package `gameLogic` dient zur Überprüfung und Ausführung der Regeln. Die Kommunikation zu anderen Packages erfolgt ausschließlich über den `LogicProcessor`.

Es können neue Regelwerke durch das Erben von Rules erzeugt werden. Wobei beim momentanen Aufbau nur das Implementieren von Poolbillard-Spielregeln vorgesehen ist. Snooker und Carambolage Regeln wurden bei der Konzeption nicht berücksichtigt, da diese einen schwer einzuschätzenden Aufwand für die Recherche und die Konzeption mit sich gebracht hätten.

`EventSnapshots` sind für die Regeln wichtige Ereignisse. Dies sind zum Beispiel das Einlochen einer Kugel oder die erste Kollision der Weißen mit einer anderen Kugel. Eine `EventSnapshotListe` wird bei jedem Schuss durch die Physik erzeugt und muss von Easy8BallRules oder einem anderen Regelwerk ausgewertet werden.

`RulesData` ist eine reine Datenhaltungsklasse, in der die ausgewerteten Ereignisse eines Stoßes gespeichert werden. Handelt es sich bei einem Stoß um einen `virtualShot`, endet die Ereignisbehandlung durch Easy8BallRules nach dem Erzeugen des `RulesData` Objektes. Das `RulesData` Objekt kann dann im nächsten Schritt vom Billard-Roboter, der den `virtualShot` ausgeführt hat, ausgewertet werden. Handelt es sich um einen normalen Stoß, ruft Easy8BallRules mit Hilfe von `RulesData` die entsprechenden Methoden im `LogicProcessor` auf, die dann zum Beispiel einen Spielerwechsel sowie die schriftliche Ausgabe der Regeln in die Wege veranlassen.

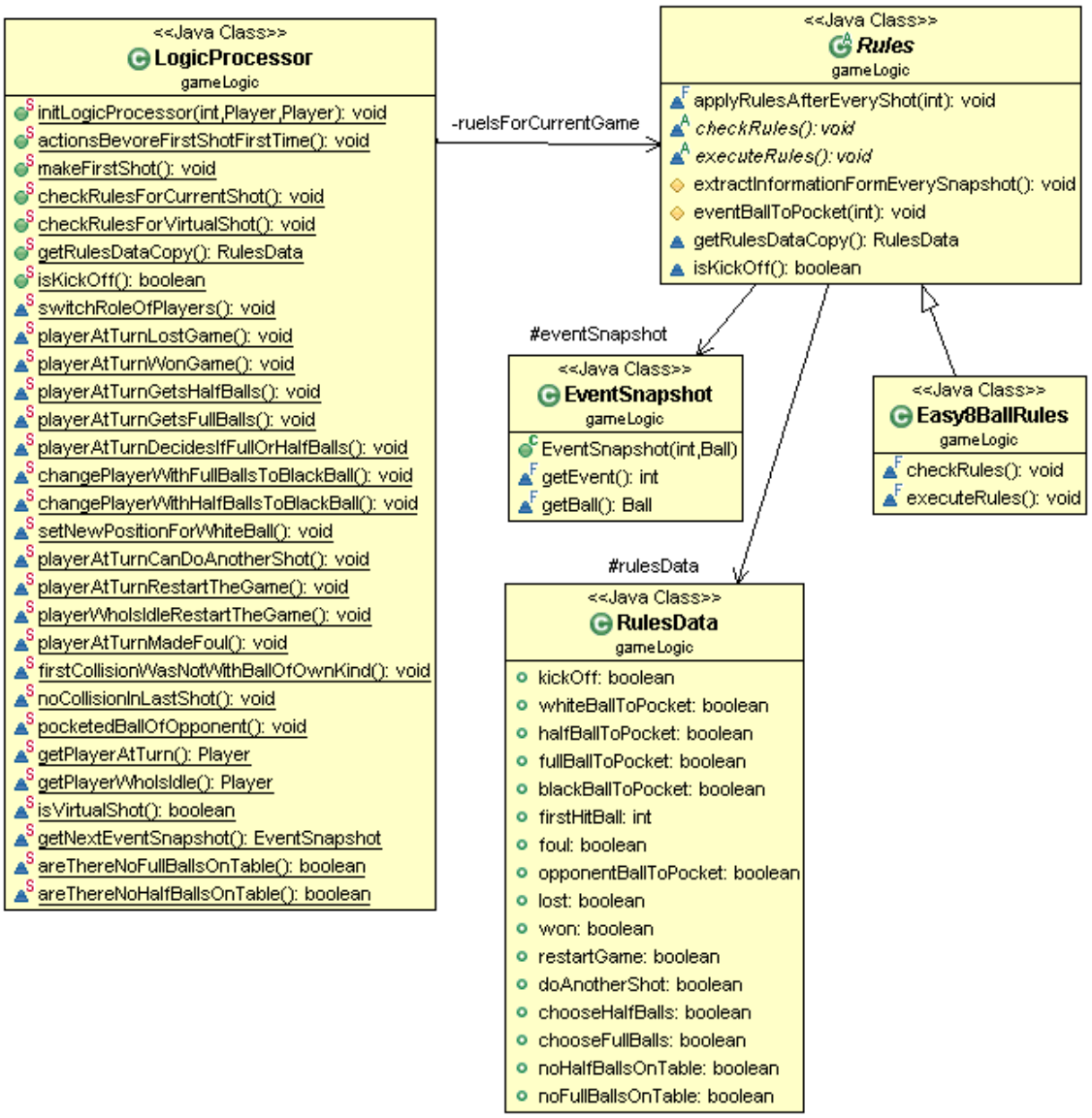

Abb. 5.4

Klassendiagramm des Packages `gameLogic`

Das Package player und Implementierung von Billard-Robotern

Es gibt 2 Arten von Spielern: `HunmanPlayer` und `GenericPlayer`. Beide Erben von `Player`, der Grundfunktionalitäten bereitstellt und für Kompatibilität sorgt. Würde man zum Beispiel das Spiel als Netzwerkspiel erweitern, würde hier zusätzlich ein *`NetworkPlayer`* entstehen.

Der `HunmanPlayer` hat keine weiteren Funktionalitäten. Beim `GenericPlayer` gab es zwei große Teilaufgaben: das dynamische Kompilieren sowie Laden von Klassen und das Erstellen eines leicht verständlichen Interfaces zum Programmieren von eigenen Billard-Computern.

Dynamisches Kompilieren und Laden von Klassen

Um Java Code zur Laufzeit zu kompilieren muss eine JDK auf dem Hostrechner installiert sein. Auf den meisten Computern ist aber nur ein JRE vorhanden. Bei einer Verteilung des Codes müsste also die JDK mitgeliefert werden. Als Alternative gäbe es auch die Möglichkeit das Kompilieren des Java Codes als Webservice auszulagern.

Dynamisches Classloading ist die Fähigkeit von Java Klassen erst zur Laufzeit zu laden. Im Grunde ist dies auch nicht besonders schwer zu realisieren. Jede Klasse ist mit dem ClassLoader assoziiert, von dem sie geladen wurde. Somit wäre es ein Leichtes mit `this.getClass().getClassLoader().loadClass(className)` weitere Klassen zu laden. Es gibt jedoch zwei wichtige Dinge, die zu beachten sind.

Wurde schon eine Klasse mit diesem Namen geladen, gibt der ClassLoader diese zurück, anstatt die neue veränderte Klasse zu laden. Zweitens, besteht die Möglichkeit schädlichen Code zu laden.

Beide Probleme lassen sich lösen, indem man einen eigenen `CustomClassLoader` implementiert, der von `ClassLoader` erbt. Möchte man nun eine Klasse, in unserem Fall einen selbst geschrieben Billard-Roboter, nach einer Veränderung erneut laden, wird eine neue Instanz des `CustomClassLoaders` erzeugt und der veränderte Billard-Roboter mit der neuen Instanz geladen. Das Verhindern des Ausführens von schädlichem Code wurde aus Zeitgründen nicht implementiert. Möglichkeiten dies zu regeln, wären über Java Reflections oder in der vom *`CustomClassLoader`* überschrieben `loadClass(className)` nur bestimmte Klassen laden zu zulassen.

Erzeugen von Billard-Robotern

Um das Schreiben eines eigenen Billard-Roboters so einfach wie möglich zu gestalten, wurde die gut dokumentierte Klasse `CustomCodeSuperClass` erzeugt, die eine abstrakte Methode `run()` und die Methoden `makeShot(PhysicParameter)`, sowie `makeVirtualShot(PhysicParameter)` bereitstellt. Zusätzlich beinhaltet `CustomCodeSuperClass` immer die aktuellen Postionen der Kugeln auf dem Tisch sowie die Positionen der Löcher.

Jeder Billard-Roboter muss von `CustomCodeSuperClass`. `Run()` wird jedes Mal aufgerufen, wenn der Billard-Roboter einen Schuss machen soll. Mit Hilfe von `makeVirtualShot(PhysicParameter)`, das ein `DataAfterVirtualShot` Objekt zurückliefert kann In `run()`, der optimale Schuss gefunden und mit `makeShot(PhysicParameter)` ausgeführt werden.

Ein Objekt vom Typ `DataAfterVirtualShot` beinhaltet immer:

-Eine Liste der Bälle die während des virtuellen Schusses versenkt wurden.

-Die Positionen der Bälle auf dem Tisch nach dem virtuellen Schuss.

-Ein `RulesData`-Objekt, welches auch im Logik Package zum Anwenden der Regeln benutzt wird.

Um einen Code mit Code Completion zu schreiben, ist es nicht notwendig den vollen Quellcode des Billard-Programms zu besitzen. Es reicht eine kleine JAR-Datei, die aus dem player Package sowie den Klassen `RulesData` und `PhysicParameter` besteht. Dies muss dann nur in die bevorzugte Entwicklungsumgebung eingebunden werden. Wer jedoch seinen Code debungen will, kommt um das Nutzen des vollen Quellcodes nicht herum.

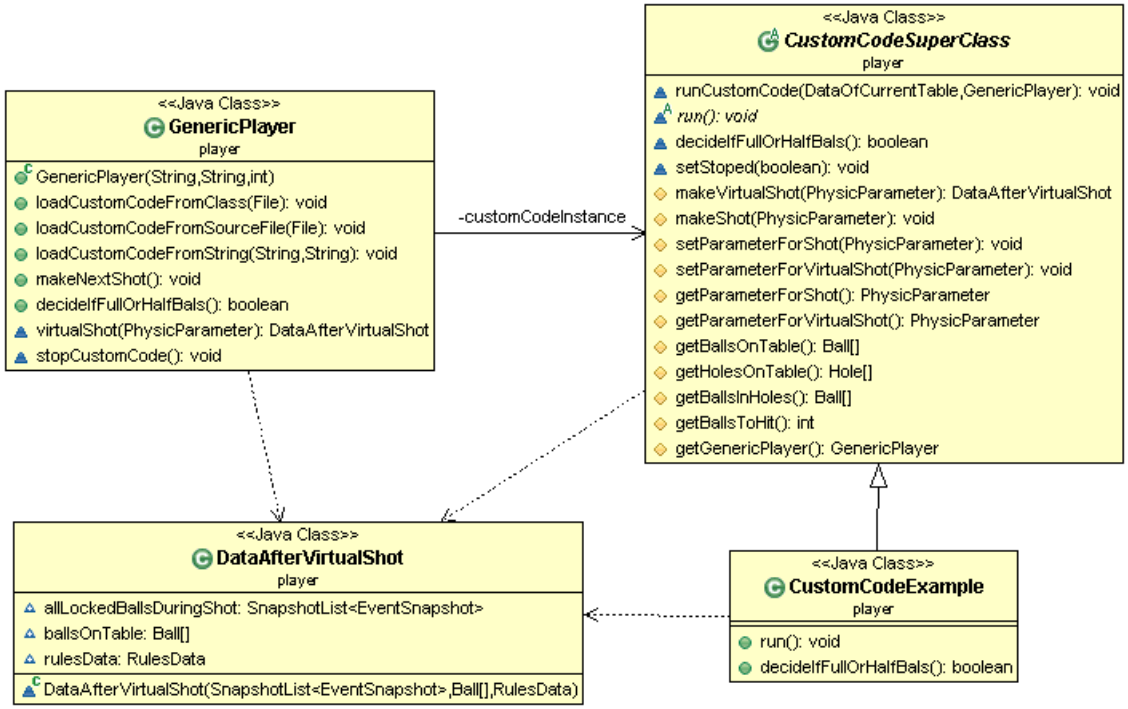

Abb. 5.5

Klassendiagramm zur Veranschaulichung der Funktion eines Billard-Roboters

Quellenverzeichnis

Verwendete Quellen

[1] http://www.komm-mach-mint.de/

[2] http://www.mintzukunftschaffen.de/die-initiative.html

[3] http://de.wikipedia.org/wiki/Portal:Billard

[4] http://pl.physik.tu-berlin.de/groups/pg287/pdf/PG

[5] http://www.psd-tutorials.de/modules.php