technische universität
dortmund

fakultät für informatik

Ausarbeitung zum Fachprojekt

Realisierung eines Tennis-Spiels mit Leap-Motion

Marco Greco
Enes Arpaci
Vincent Reckendrees
Artur Ljulin

30. Juli 2017

Gutachter:
Dipl.-Ing. Thomas Kehrt

Lehrstuhl Informatik VII
Graphische Systeme
TU Dortmund

Inhaltsverzeichnis

1 Einleitung

Die vorliegende Arbeit beschäftigt sich mit dem Abschlussprojekt des Visual Computing Fachprojekts zur Realisierung eines virtuellen Tennis-Spiels, bei dem die Steuerung mithilfe der Leap Motion[1] umgesetzt wird. Zur Umsetzung des Projekts wurden einige der Ergebnisse der Projektaufgaben der ersten Phase des Fachprojekts kombiniert und als Grundgerüst verwendet. Hierzu gehören insbesondere die Projektaufgaben, die sich mit der Leap Motion und Qt[2] befassen.

1.1 Motivation

Für die letzte Phase des Fachprojekts wurde die Vorgabe gestellt, ein eigenständiges Projekt in einer kleinen Gruppe zu entwickeln und dabei die während der ersten Phase des Fachprojekts erworbenen Kompetenzen kreativ einzusetzen. Für das Projekt wurde die Verwendung eines der in den Projektaufgaben vorgestellten Eingabegeräte und eine graphische Ausgabe gefordert. Über das Erwerben von Kompetenzen im Bereich der Softwareentwicklung und das Kennenlernen von Technologien im Bereich des Visual Computing hinaus, sollte auf Grundlage des entwickelten Projekts das Verfassen einer wissenschaftlichen Arbeit eingeübt werden.

1.2 Problemstellung

Für die Realisierung des Tennis-Spiels ergeben sich die folgenden 3 Teilprobleme:

1. Für die Steuerung muss die Leap Motion in die Anwendung integriert werden. Die Daten, die der Sensor liefert, werden zum einen für die Darstellung des Schlägers und zum anderen für die Kollisionsberechnung benötigt. Durch die Nutzung der Leap Motion soll dem Spieler das Gefühl vermittelt werden einen Schläger in der Hand zu halten mit dem er die Flugrichtung des Balles kontrolliert.

[1] Sensor, der in der Lage ist die Hände des Benutzers zu erkennen
[2] Framework zur Implementierung von GUIs (Graphical User Interfaces)

2. Für die graphische Ausgabe müssen die 3D-Modelle der einzelnen Spielobjekte erzeugt und eine sinnvolle Rendering-Architektur implementiert werden. Wichtig dabei ist, dass der Spieler stets eine klare Übersicht über das Spielgeschehen hat.

3. Für ein möglichst realitätsnahes Spielgefühl muss der Ball grundlegenden physikalischen Einflüssen unterlegen sein. Dazu gehört auch eine korrekte Auflösung von Kollisionen des Balles mit den restlichen Objekten im Spiel.

1.3 Regeln und Ziel des Spiels

Das Ziel des Spiels ist es, eine möglichst hohe Punktzahl zu erreichen. Ein Punkt wird vergeben, wenn der Ball eine sich im Spielfeld befindliche Zielscheibe trifft, die nach jedem Treffer zufällig an einen anderen Ort gesetzt wird. Nach jeweils einer bestimmten Anzahl an Treffern wir dem Spieler ein Lebenspunkt gutgeschrieben. Das Ende des Spiels ist erreicht, wenn der Spieler all seine Lebenspunkte verloren hat. Ihm wird ein Lebenspunkt abgezogen, wenn der Ball hinter den Schläger fliegt oder der Ball manuell vom Spieler zurückgesetzt wird.

2 Graphische Ausgabe

In der graphischen Ausgabe der Anwendung sollen alle Objekte dargestellt werden. Zu den darstellbaren Objekten gehören der Ball, der Schläger, das Spielfeld und eine Zielscheibe, wie in Abbildung 2.1 zu sehen ist.

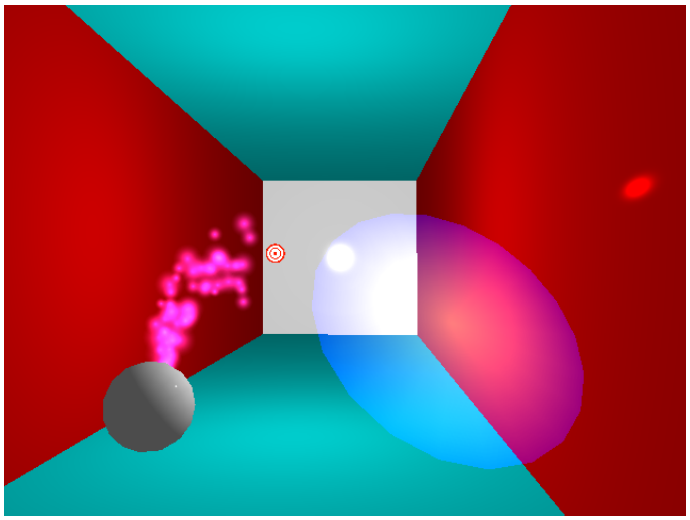


Abbildung 2.1: Szene aus der Anwendung

2.1 Rendering Architektur

Der in Abbildung 2.2 dargestellte Ausschnitt des Klassendiagramms (siehe Anhang A) zeigt die zentralen Klassen des Rendering-Systems und ihre Beziehungen zueinander.
Die Klassen, die die Spielobjekte repräsentieren, also `Ball`, `Racket`, `Box` und `Scorefield`, enthalten Member, die das `Renderable`-Interface implementieren und für die graphische Darstellung zuständig sind. Die Klassen die das Interface implementieren sind:

- `BallRenderable` und `BallParticleRenderable` für den Ball
- `RacketRenderable` für den Schläger

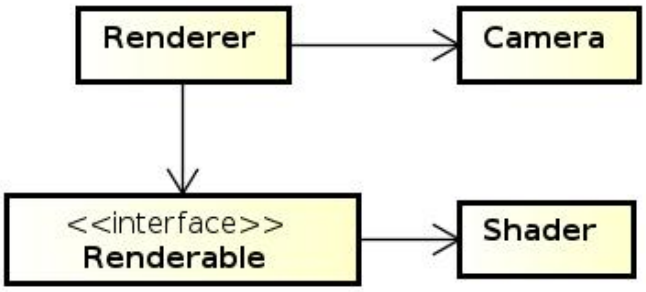


Abbildung 2.2: Klassen des Rendering-Systems

- `BoxRenderable` für das Spielfeld
- `ScorefieldRenderable` für die Zielscheibe

Jedes dieser Renderables kümmert sich bei der seiner Initialisierung um die Erzeugung der Render-Daten, der Buffer, die die Render-Daten halten, und des Shader-Programms. Der Renderer selbst hält ein `Camera`-Objekt und eine Liste der Renderables und delegiert den Render-Call aus der Hauptschleife an die einzelnen Renderables. Dabei wird das `Camera`-Objekt an die Renderables übergeben, sodass diese sich die View- und Projection-Matrix holen können. Bei der Aktualisierung eines Spielobjekts in der Hauptschleife gibt das Spielobjekt die neuen Daten an sein Renderable weiter, damit dieses die Model-Matrix aktualisieren kann. Eine Ausnahme bildet das `BallParticleRenderable`, das für jedes Partikel eine eigene Model-Matrix erzeugen muss und in Abschnitt 2.4 genauer erläutert wird.

2.2 3D-Modelle der Renderables

Für das Erstellen der Modelle, die in der Anwendung zu sehen sind, wurde keine Modelling-Software verwendet. Da es sich bei den Modellen um einfache geometrische Objekte wie Kugeln, Kreise, und Rechtecke handelt, werden die Render-Daten, also Vertizes, Normale, Farben und ggfs. Indizes für die Faces, innerhalb der Renderables erzeugt.

2.2.1 Spielfeld

In der graphischen Ausgabe hat das Spielfeld die Form eines Quaders. Intern werden die einzelnen Wände als Ebenen repräsentiert. Da die `Plane`-Klasse eine Ebene in der

Hesseschen Normalform[1] abspeichert, können aus den Abstandswerten der einzelnen Ebenen leicht die Koordinaten für die 8 Vertices des Modells gewonnen werden.

2.2.2 Schläger

Der Schläger ist in der graphischen Ausgabe ein regelmäßiges Polygon. Daher reduziert sich das Problem der Erzeugung der Vertices auf die Berechnung von Koordinaten auf dem Einheitskreis. Algorithmus 2.1 enthält einen Ausschnitt aus dem Programmcode als Pseudocode.
Für ein besseres visuelles Ergebnis lässt sich über den Parameter s die Anzahl der Vertices steuern, um das Schläger-Modell mehr nach einem Kreis aussehen zu lassen. In der Anwendung besteht Das Schläger-Modell aus 21 Vertices.

2.2.3 Ball

Die Erzeugung der Vertices für das Ball-Modell lässt sich sich auf die Berechnung von Koordinaten auf der Kugeloberfläche der Einheitskugel zurückführen. Das Modell wird dabei in die beiden Pole und eine bestimmte Anzahl an Ringen, auf denen sich Vertices befinden, geteilt. Jeder Ring hat dabei die gleiche Anzahl an Vertices. Algorithmus 2.2 enthält einen Ausschnitt des Programmcodes als Pseudocode.
Auch hier lässt dich mit höheren Werten für die Parameter p und r ein besseres visuelles Ergebnis erzielen. In der Anwendung hat das Ball-Modell 15 Ringe mit jeweils 10 Vertices, also insgesamt 152 Vertices.

2.3 Kamerabewegung

Um die Flugbahn des Balles besser einschätzen zu können und um ein besseres Gefühl für die Position des Schlägers innerhalb des Spielfeldes zu erhalten, bewegt sich die Kamera bei Bewegung des Schlägers mit. Die Kamerabewegung ist in Abbildung 2.3 dargestellt.
In jedem Durchlauf der Hauptschleife wird der Kamera nach dem Aktualisieren der Position des Schlägers die neue Position mitgeteilt, wobei nur die x- und y-Koordinate in die neue Kameraposition mit einfließen. Die Kamera passt dann die View-Matrix an. Der Bewegungsbereich der Kamera ist so eingeschränkt, dass , wenn der Schläger außerhalb dieses Bewegungsbereiches bewegt wird, die Kamera innerhalb des Spielfelds verbleibt.

[1] Näheres dazu in Kapitel 4

Eingabe: Anzahl der Kreissektoren s
Ausgabe: Vertices V
$\Delta\theta := 2 * \pi / s$
new Array $V[s+1]$
$V[0] := (0, 0, 0)$
for $i := 0$ **to** $i < s$ **do**
$V[i+1] := (\cos(i * \Delta\theta), \sin(i * \Delta\theta), 0)$
end for

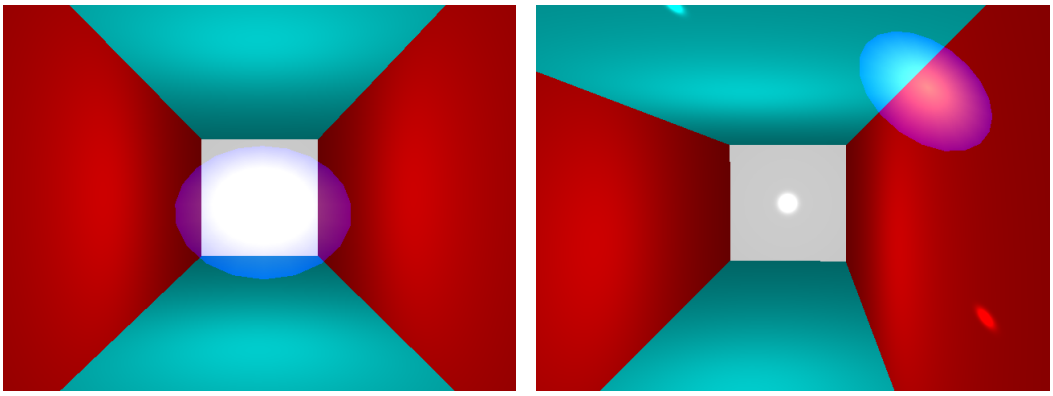
Algorithmus 2.1: Algorithmus für die Erzeugung des Schläger-Modells

Eingabe: Anzahl der Ringe r, Anzahl der Punkte pro Ring p
Ausgabe: Vertices V
$\Delta\phi := 2 * \pi / p$
$\Delta\theta := \pi / (s+1)$
$\phi := 0$
$\theta := 0$
new Array $V[2 + r * p]$
$V[0] := (0, 0, 1)$
$V[1 + r * p] := (0, 0, -1)$
$offset := 0$
for $i := 1$ **to** $i \leq r$ **do**
$\theta := \theta + \Delta\theta$
for $j := 0$ **to** $j < p$ **do**
$\phi := \phi + \Delta\phi$
$V[i + j + offset] := (\sin(\theta) * \cos(\phi), sin(\theta) * \sin(\phi), \cos(\theta))$
end for
$\phi := 0$
$offset := offset + p - 1$
end for

Algorithmus 2.2: Algorithmus für die Erzeugung des Ball-Modells

2.4 Partikelsystem

Partikelsysteme sind in interaktiven Anwendungen wie Computerspiele und in der Postproduktion von Filmen, seien es Real- oder Animationsfilme, allgegenwärtig. Man denke an typische Effekte wie Feuer und Rauch, aber auch Flüssigkeiten lassen sich mithilfe von Partikelsystemen realisieren.
Nach [3] sind Partikelsysteme besonders geeignet für Objekte die sich aufgrund ihrer Komplexität nur schwer mit einem Mesh darstellen lassen und sich im Verlauf der Zeit in ihrer Form ändern. Eine einfache Repräsentation solcher Objekte lässt sich daher mit Wolken aus Partikeln umsetzen, wobei die Partikel einfache Primitive wie Punkte, Linien und Dreiecke sind[3].



(a) Kamera in neutraler Position

(b) Kamera bei Bewegung des Schlägers nach rechts oben

Abbildung 2.3: Kameraposition bei Bewegung des Schlägers

In der Anwendung wird ein Partikelsystem genutzt um den Flugverlauf des Balles darzustellen. Dabei sind die Partikel Quads, bestehend aus jeweils 6 Vertices, auf die eine Textur projiziert wird.

2.4.1 Komponenenten des Partikelsystems

In der Anwendung besteht das Partikelsystem aus zwei Klassen, nämlich dem `Particle` und dem `BallParticleRenderable`.
Die `Particle` Klasse repräsentiert ein Partikel, welches eine Position in Weltkoordinaten, eine Geschwindigkeit, eine Lebensdauer und eine Größe hat. Die `BallParticleRenderable` Klasse ist für die Darstellung und Verwaltung aller im System vorhandenen Partikel zuständig. Zur Verwaltung gehört das emittieren neuer Partikel, das Aktualisieren der aktiven Partikel und Löschen der ‚toten' Partikel, also solcher, deren Lebensdauer null erreicht hat. Die Klasse hält Attribute, die die maximale Anzahl der aktiven Partikel beschränkt sowie die Emittiergeschwindigkeit. Sie hat auch ein Attribut für die Position. Alle neuen Partikel werden relativ zu dieser Position emittiert. Dabei ist die Position des Emittierpunktes auf dem Rand des Balles entgegengesetzt dem Geschwindigkeitsvektor des Balles. Da die Position des Emittierpunktes an die Position des Balles gekoppelt ist, wird bei jedem Update des `Ball`-Objekts die neue Position an den `BallParticleRenderable` weitergegeben.

2.4.2 Instanced Rendering

Instanced Rendering bezeichnet eine effiziente Variante des Renderns von Objekten die dieselben Renderdaten verwenden, sich aber nur darin unterscheiden, wo sie im

Weltkoordinatensystem platziert werden. Für die Partikel des Partikelsystems trifft dies zu, da jedes Partikel, wie bereits erwähnt, ein einfaches Quad ist und sich nur in seiner Position und Größe unterscheidet.
Die Effizienz dieser Methode rührt daher, dass zum Rendern der Partikel nun nicht mehr über die Liste aller Partikel iteriert wird und für jedes Partikel ein Draw-Call getätigt wird. Stattdessen wird pro Frame nur ein einziger Draw-Call getätigt, das heißt, dass nur ein Befehl über den langsamen Peripheriebus zur Grafikkarte gesendet werden muss. Für das Instanced Rendering bietet OpenGL die Funktionen `glDrawArraysInstanced` und `glDrawElementsInstanced`.
Um nicht zu sehr in den technischen Details zu versinken, soll nur kurz schematisch anhand von Abbildung 2.4 der Vorgang beim Rendern der Partikel erläutert werden.

0	v_1	v_2	v_3	v_4	v_5	v_6
1	t_1	t_2	t_3	t_4	t_5	t_6
2	M_1	M_2	• • •			M_n

Abbildung 2.4: VAO im `BallParticleRenderable` nach jedem Update der Partikel

Wie an Abbildung 2.4 zu sehen ist, befinden sich die Vertizes $v_1 \dots v_6$, die Texturkoordinaten $t_1 \dots t_6$ und die Model-Matrizen $M_1 \dots M_n$, wobei n die Anzahl der aktiven Partikel ist, in ihren eigenen Array-Buffer des VAOs des `BallParticleRenderable`-Objekts. Die Vertices und Texturkoordinaten ändern sich nicht. Die Model-Matrizen hingegen werden bei jedem Update der Partikel neu erzeugt und an den Array-Buffer gesendet.
Um Instanced Rendering zu nutzen muss der Array-Buffer der Model-Matrizen als Instance-Attribut des Vertex-Shaders deklariert werden. Dies sollte bei der Assoziation der Array-Buffer mit den Attributen des Vertex-Shader-Programms geschehen. Beim Rendern einer Instanz werden nun alle Vertexdaten und Texturkoordinaten, die keine Instanz-Variablen sind, und die jeweilige Model-Matrix der Instanz verwendet.
Für weitere Informationen zum Thema Instanced Rendering sei auf [2] verwiesen.

2.4.3 Blending und Tiefentest

In dem verwendeten Partikelsystem sollen die Farben der Partikel, wenn sie sich überlagern, addiert werden. Der resultierende Effekt ist, dass bei genügend sich überlagernden Partikel ein leuchtend weißer Bereich zu sehen ist. Blending erlaubt es,

Farben miteinander zu vermischen und ist ein Teil der Verarbeitung der Fragmente in der Rendering Pipeline. OpenGL bietet verschiedene Blending-Arten an, die durch den Aufruf der Funktion `glBlendFunc` eingestellt werden können.

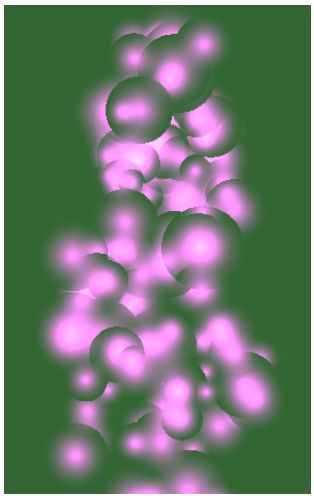


Abbildung 2.5: Partikelsystem bei aktiviertem Blending

Abbildung 2.5 zeigt wie das Partikelsystem bei eingeschaltetem Blending und Verwendung von `glBlendFunc` mit den Argumenten `GL_SRC_ALPHA` und `GL_ONE` aussieht. Eine Übersicht über die verschiedenen Argumente für `glBlendFunc` und ihre Auswirkungen auf die Farbe eines Fragments ist in [4] zu finden.
Wie an Abbildung 2.5 zu erkennen ist tritt trotz eingeschaltetem Blending der gewünschte Effekt nicht auf. Grund dafür ist der Tiefentest. Für den Tiefentest spielt es keine Rolle, ob ein Fragment teilweise transparent ist, da der Tiefenwert für ein Fragment trotzdem in den Tiefenpuffer geschrieben wird, vorausgesetzt der Wert ist geringer.
Um dies zu umgehen, lässt sich mittels der Funktion `glDepthMask` das Schreiben in den Tiefenpuffer für das Rendern der Partikel verbieten. Das endgültige Resultat ist in Abbildung 2.6 zu sehen.

Abbildung 2.6: Partikelsystem bei aktiviertem Blending und nicht-beschreibbarem Tiefenpuffer

3 Steuerung

Die Leap Motion bietet über ihre Tracking API[1] die Möglichkeit auf einzelne Daten der durch den Sensor detektierten Hände zuzugreifen. Da der Schläger im Spiel mit der rechten Hand gesteuert wird, werden nur die Position, der Normalenvektor der Handfläche und der Geschwindigkeitsvektor der rechten Hand benötigt.
Die Position der Hand wird zur Bestimmung der Position des Schlägers in der 3D-Szene verwendet. Der Normalenvektor wird zum einen für die Ausrichtung des Schläger-Modells und zum anderen für die Reflexion des Balles bei Kollision mit dem Schläger verwendet. Kommt es zwischen dem Ball und dem Schläger zur Kollision, so wird der Geschwindigkeitsvektor der Hand auf den Geschwindigkeitsvektor des Balles addiert.
Die Leap Motion erlaubt es Gesten zu registrieren, die von der Leap Motion Software erkannt werden können. Im Spiel wird für die linke Hand die Kreisgeste verwendet, die dem Spieler erlaubt den Ball in die Mitte des Spielfelds zurückzusetzen.

[1] siehe https://developer.leapmotion.com/documentation/cpp/devguide/Leap_Guides2.html

4 Physik

Um die Bewegungen des Ball möglichst realistisch und nachvollziehbar zu realisieren und somit dem Nutzer ein echtes Spielgefühl zu vermitteln, müssen einige physikalische Eigenschaften der "echten Welt" ins Spiel implementiert werden. In diesem Fall heißt es, dass die Reflexionen eines Balls auf einer Oberfläche richtig berechnet werden müssen. Dementsprechend müssen auch Kollisionsabfragen dafür realisiert werden. Des Weiteren soll die Bewegung des Balls innerhalb der Box korrekt berechnet werden sowie ein Gravitationsfaktor die Fallgeschwindigkeit beeinflussen können. Da im Programm in jedem Durchlauf der Hauptschleife zuerst die Bewegungen und dann die Kollisionen bzw. die Reflexionen berechnet werden, wird auch in diesem Kapitel diese Reihenfolge beibehalten.

4.1 Bewegungen

Die verschiedenen physikalische Eigenschaften des Balls werden in der Klasse Ball-Physics verwaltet. Dazu gehören: Masse, Kraft, Gravitation und Geschwindigkeit. Bei jedem Tick wird die Update-Methode des Balls aufgerufen, die wiederum für die Positionsbestimmung zunächst die Update-Methode der Klasse Ballphysics aufruft, die die aktuelle Geschwindigkeit berechnet. Da die Geschwindkeit als 3-dimensioneller Vektor gespeichert wird, ergibt sich daraus auch immer die Richtung in die der Ball sich bewegt. Dabei wird zunächst die Beschleunigung des Balls, die durch die Masse und die Gravitationskraft beeinflusst wird, berechnet und schließlich auf den Geschwindigkeitsvekor addiert.

$$Beschleunigung = (\frac{1}{Masse} + Gravitation) * t \tag{4.1}$$

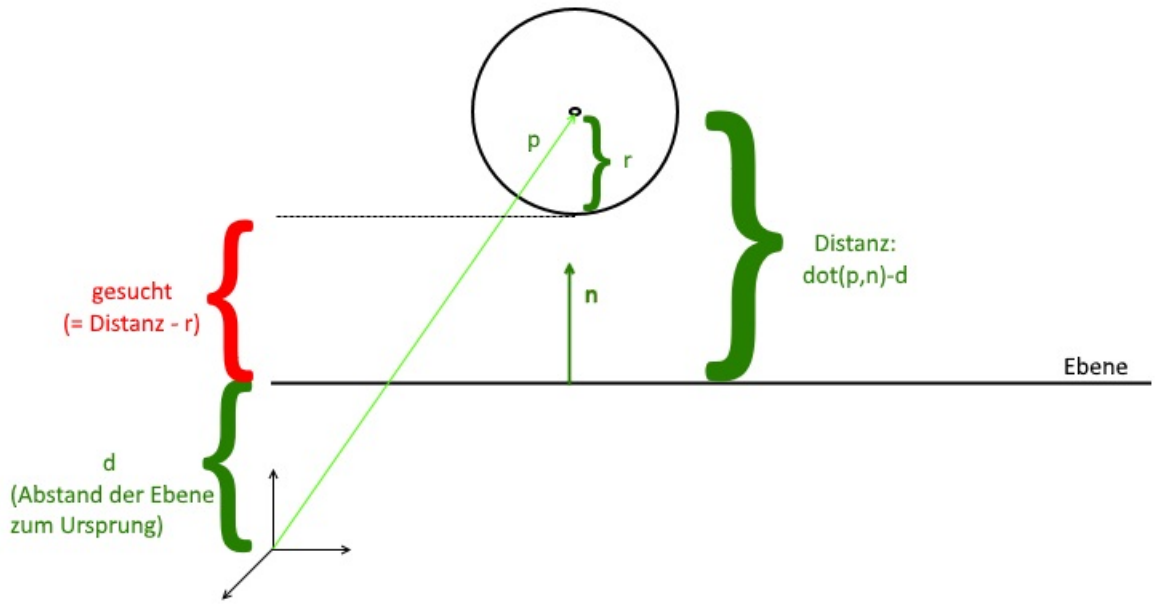
Mithilfe des Geschwindigkeitsvektor lässt sich dann die nächste Position des Balls bestimmen:

$$Position_{neu} = Position_{alt} + Geschwindikeit_{neu} * t \tag{4.2}$$

Schließlich werden dann basierend auf der neuen Position die Rendering-Informationen aktualisert.

4.2 Reflexion

Nachdem die Positionen aktualisert wurden, wird geprüft, ob an den neuen Positionen Kollisionen aufgetreten sind. Dies geschieht in der `CollisionResolver`-Klasse. Die Funktion `resolveCollisions(Ball* ball, Racket* racket, Box* box)` wird von der Hauptschleife aus aufgerufen und überprüft zuerst die Kollision des Balls mit der Box bzw. mit den 6 Ebenen der Box und dann die Kollision mit dem Schläger. Die Ebenen werden in einer Hesseschen Normalform ähnlichen Darstellung gespeichert, sodass der Abstand des Balls zu einer Ebene leicht berechnet werden kann.



(a)

Abbildung 4.1: Übersicht der Kollisionsüberprüfung

Da die Hessesche Normalform wie folgt aussieht,$E : \vec{x} \cdot \vec{n} = d$, und d der Abstand zum Ursprung ist, lässt sich die Formel umformen zu $E : \vec{x} \cdot \vec{n} - d = 0$. Diese Formel ist erfüllt wenn x auf der Ebene liegt. Ersetzt man also 0 durch eine Variable c, kann der Abstand von Punkt x zur Ebene genau ermittelt werden. In unserem

Fall erlauben wir auch negative Abstände; negative Abstände entstehen dann durch Ballpositionen, die hinter der Ebene liegen (entgegen der Richtung des Normalvektors). Der Abstand des Ballmittelpunkts zur Ebene ist also $c = \vec{p} \cdot \vec{n} - d$. Somit lässt sich eine Kollision feststellen, wenn der Abstand zur Ebene kleiner als der Radius des Balls ist.

Damit der Ball bei zu hohen Geschwindigkeiten nicht hinter die Ebene gelangt und die Kollision nicht erkannt wird, wurde eine Kollisionsüberprüfung implementiert, die genau dies verhindern soll (nach [1]). Denn bei dieser Methode wird der Ball auf eine Position vor der Ebene zurückgesetzt, falls er tatsächlich hinter die Ebene gelangt. Das Prinzip ist wie folgt:

Es werden immer die letzten beiden Positionen des Balls gespeichert (C_0, C_1) und die Distanzen zu beiden Zeitpunkten nach schon beschriebener Weise berechnet (d_0, d_1). Befindet sich C_0 vor der Ebene und C_1 hinter der Ebene bzw. ist d_0 positiv und d_1 negativ, so wird die neue Position des Balls zwischen diesen beiden Punkten interpoliert (s. Formel 4.3 und 4.4).

$$u = \frac{d_0 - r}{d_0 - d_1} \tag{4.3}$$

$$C_{neu} = (1 - u)C_0 + uC_1 \tag{4.4}$$

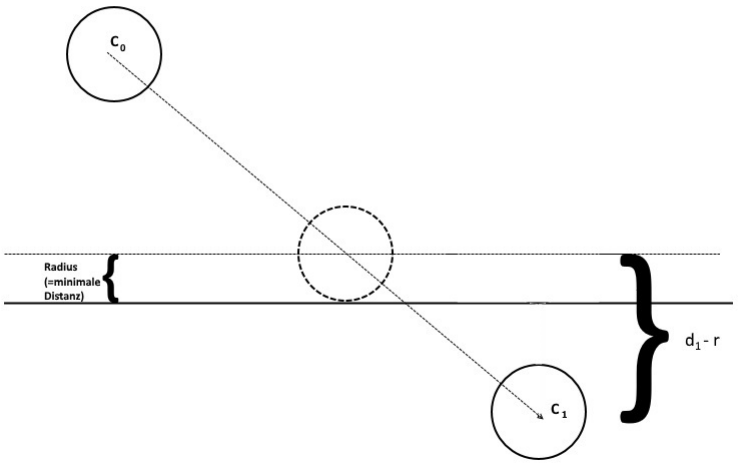


Abbildung 4.2: Interpolation der Ballposition bei vollständigem Durchstoßen der Ebene

Ist der Ball nicht vollständig hinter die Ebene geflogen - d.h. es gilt $d_1 < 0$ und $|d_1| < Radius$ - dann wird der Ball um $Radius - |d_1|$ in Richtung des Normalenvektors der Ebene verschoben.
Die Kollisionserkennung mit dem Schläger verläuft nach ähnlichem Schema ab. Auch die Fläche des Schlägers wird als Ebene gespeichert, sodass auch hier nur der Abstand des Balls zur Ebene berechnet werden muss, der dann bei tatsächlicher Kollision betragsmäßig kleiner als der Radius ist. Jedoch ist Der Schläger keine unendlich große Fläche wie die Boxwände, sondern ist auf eine bestimmte Kreisfläche begrenzt. D.h. es muss eine zusätzliche Abfrage gemacht werden, die überprüft ob der Ball eine entsprechend kleine Distanz zum Schlägermittelpunkt besitzt. Wenn $\vec{r}$ die Position des Schlägermittelpunkts, $\vec{b}$ die Position des Balls und R den Radius des Schlägers beschreibt dann gilt bei Kollision:

$$|\vec{r} \cdot \vec{b}| <= R \tag{4.5}$$

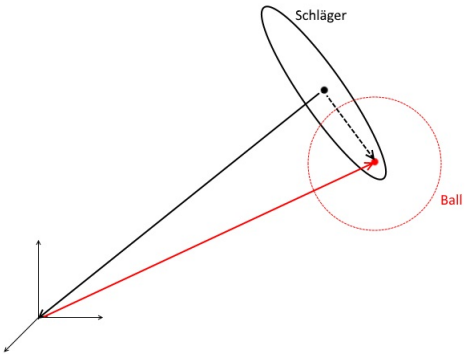


Abbildung 4.3: Kollisionserkennung zwischen Schläger und Ball

Ist eine Kollision aufgetreten, so kann der Ball entsprechend der Formel 4.6) reflektiert werden.

$$\vec{v}_{neu} = 2 * \langle \vec{n}, -\vec{v} \rangle * \vec{n} + \vec{v} \tag{4.6}$$

Die Formel lässt sich wie folgt herleiten:
Zunächst muss der Einfallsvektor des Balls auf den Normalenvektor projiziert werden (gesucht ist also eine Faktor k, sodass $k\vec{n}$ eine Projektion von $-\vec{v}$ nach $\vec{n}$ ist). Der Einfallsvektor wird deswegen invertiert, weil ein Ausfallsvektor gesucht wird, der nicht wie der Einfallsvektor in Richtung der Ebene zeigt. k lässt sich mithilfe des Skalarprodukts $\langle \vec{n}, -\vec{v} \rangle$ berechnen, da der Normalenvektor einer Ebene in unserer Implementierung immer normiert gespeichert wird.

(a) Projektion des Einfallsvektors auf den Normalvektor

(b) Zusammensetzung des Ausfallsvektors

Abbildung 4.4: Berechnung des neuen Vektors bei einer Reflexion an einer Ebene mit Normalenvektor

Die Länge des Projizierten Vektors $k\vec{n}$ muss nun, wie in Abbildung 4.4 zu sehen, verdoppelt werden und zum Einfallsvektors $\vec{v}$ addiert werden. Das Ergebnis ist dann der Ausfallwinkel in entsprechender Richtung. Soll der Ausfallsvektor betragsmäßig kleiner sein (d.h. der Ball soll z.B. beim Abprallen an Wänden an Geschwindigkeit verlieren), kann der Vorfaktor 2 entsprechend verringert werden (In unserer Implementierung beträgt der Wert: 1,75).

5 Ergebnisse und Diskussion

Um die Umsetzung des Fachprojekts zu analysieren und ausreichend zu diskutieren, wurden Testpersonen befragt, nachdem sie eine Zeit lang das Spiel getestet haben. Folgende Fragen wurden ihnen danach gestellt:

1. „Welche Probleme haben Sie bei der Steuerung bemerkt ?“

2. „Wie haben Sie die Steuerung im Vergleich zum normalen Tischtennis empfunden ?“

3. „Als wie realitätsnah haben Sie die Bewegung des Balls empfunden ?“

5.1 Graphische Ausgabe

Die graphische Ausgabe wurde von allen Testpersonen für gut empfunden. Probleme, die bei der Steuerung auftraten, wurden durch die Kamerabewegung beim Bewegen der Hand und die nahe Ansicht auf die Szene verstärkt. Eine Testperson antwortete auf Frage eins: „Da die Kamera sich mit der Hand bewegt, ist es manchmal schwer den Ball zu treffen.“. Die Bewegung der Kamera wäre nicht notwendig, wenn die Szene aus größerer Entfernung betrachtet werden würde.

5.2 Steuerung

Beim Bedienen der Anwendung traten die meisten Probleme auf. Eine Person antwortete auf Frage zwei: „Das tatsächliche Gefühl, Tischtennis zu spielen, ist nicht vorhanden. Der Ball wird mit der flachen Hand geschlagen, nicht mit einem Objekt, was in der Hand gehalten wird.“ Zur Bestimmung der Position des Schlägers sollte nicht der Handmittelpunkt, sondern der Mittelpunkt eines gedachten Schlägers in der Hand des Spielers verwendet werden. Dadurch könnte der Spieler das Gefühl erhalten, er schlage den Ball mit einem Schläger.

5.3 Physik

Einige Testpersonen empfanden, dass die Geschwindigkeit des Balls nach dem Schlagen zu hoch war und zu schnell abnahm. Um den Flug des Balls noch realistischer zu gestalten, könnte die berechnete Gravitation erhöht werden, außerdem sollte die Geschwindigkeit des Balls nach dem er getroffen wurde geringer sein.

A Klassendiagramm

Abbildung A.1 enthält alle in der Anwendung vorkommenden Klassen. Der Übersichtlichkeit halber, wurden die Funktionen und Member der einzelnen Klasse ausgelassen.

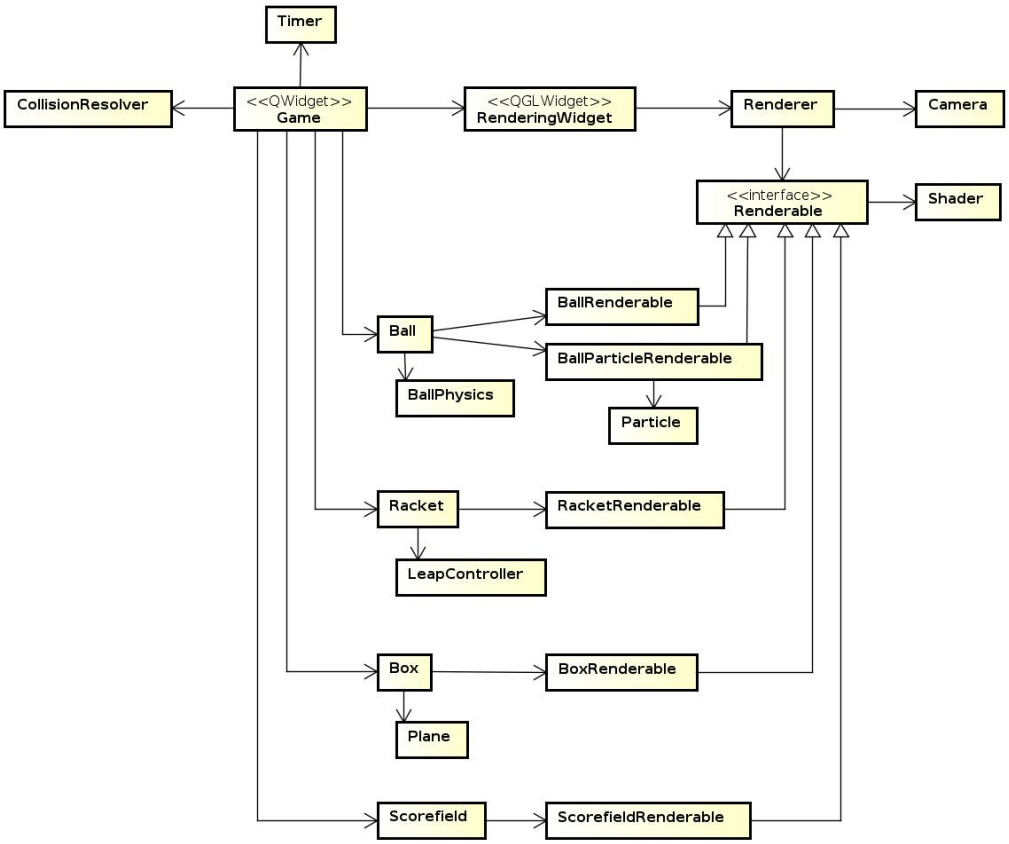


Abbildung A.1: Klassendiagramm

Abbildungsverzeichnis

Algorithmenverzeichnis

Literaturverzeichnis

[1] Gomez, M.: *Simple Intersection Tests for Games*. 1999. – URL: http://www.gamasutra.com/view/feature/131790/simple_intersection_tests_for_games.php, abgerufen am: 23.7.2017

[2] Kessenich, J.; Sellers, G.; Licea-Kane, B.; Shreiner, D.: *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. 8. ed. Upper Saddler River, NJ: Addison-Wesley, 2013

[3] Reeves, W.: Particle Systems - Technique for Modeling a Class of Fuzzy Objects. In: *ACM Transactions on Graphics*, ACM, 1983, S. 359–376

[4] Virag, G.: *Grundlagen der 3D-Programmierung*. München: Open Source Press, 2012