HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

Minigolf im Wohnzimmer

Bachelorarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

Frühlingssemester 2012

Autoren	Philipp Eichmann und Roman Giger
Betreuer	Prof. Oliver Augenstein
Gegenleser	Prof. Dr. Peter Heinzmann
Experte	Dr. Thorsten Kramp, IBM Research

Inhaltsverzeichnis

Kapitel 1

Allgemein

1.1 Aufgabenstellung

Minigolf im Wohnzimmer

Studiengang:	Informatik (I)
Semester:	FS 2012 (20.02.2012-16.09.2012)
Durchführung:	Bachelorarbeit
Fachrichtung:	Software
Institut:	IFS: Institut für Software
Gruppengrösse:	2 Studierende
Status:	zugewiesen
Verantwortlicher:	Augenstein, Oliver
Betreuer:	Augenstein, Oliver
Gegenleser:	Peter Heinzmann
Experte:	Dr. Thorsten Kramp, IBM Research Zürich, Rüschlikon
Industriepartner:	[Nicht definiert]
Ausschreibung:	Zu Lösen ist eine Augmented Virtuality Aufgabe, z.B. die Implementation eines Minigolf-Spiels Die Idee: - das Spielfeld, der Ball und das Loch werden per Beamer auf den Boden projiziert - Auf das Spielfeld können reale Hindernisse gestellt werden (z.B. ein Teller, eine Schachtel, ...) - Eine Kamera (ev. auch eine Kinect) erkennt die Hindernisse und den Schläger (im wesentlichen eine Kantendetektion) - Der Spieler schiesst mit dem realen Schläger auf den virtuellen Ball (erkennen einer Bewegung) - Die Kamera zeichnet das auf. - Der Computer errechnet aus der Schlagbewegung und den Hindernissen die Bewegung des Balls (physics engine) - Das Ergebnis wird über den Beamer auf den Boden projiziert Die Aufgabe soll in dieser Arbeit nur für eine Ebene Spielfläche implementiert werden.
Voraussetzungen:	Kenntnisse in OpenCV.

O. Aygki

Abbildung 1.1: Unterschriebene Aufgabenstellung

1.2 Erklärung über die eigenständige Arbeit

Wir erklären hiermit, dass wir die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt haben, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde, dass wir sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben haben.

Rapperswil, 15.06.2012

Philipp Eichmann Roman Giger

1.3 Abstract

In der Augmented Virtuality [Sims1996] wird die virtuelle Welt durch Einwirkung von realen Objekten oder Spielern beeinflusst.
In dieser Arbeit geht es konkret um ein 2D-Minigolfspiel, das durch die Realität erweitert wird. Dabei wird das Spielfeld, der Schläger, der Ball und das Loch auf den Boden projiziert, während Hindernisse und Spieler real sind. Der Spieler nimmt mit seinen Bewegungen Einfluss auf die Position des virtuellen Schlägers. Mit diesem kann der virtuelle Ball angeschlagen werden, welcher wiederum von realen Hindernissen abprallt.
Um den Spieler und das Spielfeld mit den realen Hindernissen zu erfassen wird eine Kinect eingesetzt. Das Spielfeld wird mittels Beamer so von oben auf den Boden herunter projiziert, dass es wie auf dem Bildschirm rechteckig erscheint. Die Kinect nimmt das Spielfeld von der Seite auf. Mithilfe des integrierten Tiefensensors ist es möglich, eine perspektivisch korrekte, interne 2D-Repräsentation des Spielfeldes und deren Hindernissen zu rekonstruieren. Diese wird von der Physics-Engine benötigt, um die Bahn des virtuellen Balles richtig zu berechnen. Die Resultate der Physics-Engine werden ans GUI weitergeleitet und dadurch über den Beamer am Boden sichtbar.
Unter Verwendung der Kinect-Skeletonerkennung wird durch die Haltung des Spielers die Position und der Winkel des virtuellen Schlägers ermittelt. Auch diese Daten werden nun in die interne Repräsentation für die Physics-Engine umgerechnet und ans GUI-Framework gesendet. Die Position des virtuellen Schlägers ist für den Spieler am Boden sichtbar. Berührt der Spieler mit dem virtuellen Schläger den virtuellen Ball, ermittelt die Physics-Engine eine Kollision und leitet die daraus resultierende Ballbewegung ans GUI weiter.

Diese Arbeit zeigt, wie mittels Kinect ein ursprünglich rechteckig projiziertes Spielfeld von der Seite aufgenommen, wieder als rechteckiges Spielfeld abgebildet werden kann. Daneben wird vorgeführt, wie auf solch ein Spielfeld interaktiv Einfluss genommen werden kann.

Kapitel 2

Management Summary

2.1 Ausgangslage

Computerapplikationen, insbesondere Games, setzen neben den gängigen Interaktionsmöglichkeiten wie Maus und Tastatur auch andere Geräte zur Steuerung grafischer Benutzeroberflächen ein. Ein Beispiel dafür ist die Kinect von Microsoft, ein Sensor, der Benutzerbewegungen wahr nimmt und auf diese Weise eine Bedienung mit blossen Gesten ermöglicht.
Ziel unserer Arbeit war es, ein virtuelles Minigolfspiel zu entwickeln, bei dem sich der Spieler auf einem auf den Boden projizierten Spielfeld bewegen und einen virtuellen Ball anschlagen kann. Er hat zudem die Möglichkeit, reale Hindernisse ins Spielfeld zu legen, an denen der virtuelle Ball abprallt. Anhand der Schlagbewegung und den realen Hindernissen wird der Verlauf des virtuellen Balls bestimmt und anschliessend auf den Boden projiziert.
Speziell am Spiel ist also nicht nur die Art der Steuerung, sondern auch dass der Benutzer die Spielfeldtopografie direkt mit realen Gegenständen beeinflussen kann.

2.2 Vorgehen, Technologie

Ein Beamer wird oberhalb der Spielebene befestigt und projiziert ein rechteckiges Spielfeld auf den Boden. Eine Kinect wird so ausgerichtet, dass sie das Spielfeld und den Spieler erfassen kann. Benötigt werden optimalerweise ca. 2m x 4m Platz. Nach 3 Menüschritten kann das Spiel gestartet werden.
Die Software, geschrieben in C#, analysiert die Daten der Kinect mit Hilfe einer Bildbearbeitungsbibliothek und bestimmt Grösse und Position des Spielfelds und der Hindernisse. Danach simuliert eine Physics-Engine die Ballbewegung und Kollisionen mit den Objekten. Mithilfe des XNA Frameworks wird das Spielfeld neu gezeichnet und zur Anzeige an den Beamer übergeben.

2.3 Ergebnis

Um den virtuellen Ball anzuschlagen, braucht der Spieler keinen realen Schläger zu halten sondern nur die Haltung einzunehmen, die ein Mensch typischerweise beim Halten eines Schlägers hat. Schlagbewegungen werden von unserem System erkannt und verschiedene Parameter wie Anschlagsgeschwindigkeit und -winkel ausgewertet, um das Minigolfspiel möglichst realitätsnah zu simulieren. Trifft der virtuelle Ball nach dem Anschlag auf ein

platziertes Hindernis oder den Spielfeldrand, prallt er nach den üblichen physikalischen Gesetzen davon ab. Erreicht der Ball im Laufe des Spiels ein virtuelles Loch, so kann er darin versenkt werden und das Spiel wird beendet.
Der Endstand der Arbeit kann als Prototyp angesehen werden. Das Spiel funktioniert soweit, dass der virtuelle Ball mit dem virtuellen Schläger angeschlagen und im virtuellen Loch versenkt werden kann. Trifft er unterwegs auf ein reales Hindernis oder den Spielfeldrand, prallt der Ball daran ab.
Unsere Implementation sieht vor, dass der Beamer senkrecht von oben auf den Boden projiziert, da nur dadurch keine Verzerrung hervorgerufen wird. Für das Endprodukt vorgesehen und vorteilhaft ist eine Korrektur der Projektion, welche eine beliebige Platzierung des Projektors erlaubt, ohne dass eine Verzerrung hervorgerufen wird. Dies ist der schwierigste noch ausstehende Teil der Implementierung. Unsere Arbeit beschreibt Ansätze, wie dies realisiert werden kann. Um Produktreife zu erlangen stehen noch Optimierungen der Spielatmosphäre sowie die Einbindung weiterer Funktionen wie beispielsweise das Speichern von Spielfeldern und Spielständen, ein Szenarieneditor, interaktive Hindernisse mit Anziehungskraft und Abstosseigenschaften sowie ein Mulitplayermodus an. Was von der Programmierung her mehr ein Zeit- als ein Schwierigkeitsfaktor darstellt.

2.4 Ausblick

Das Prinzip der Interaktion mit dem auf den Boden projizierten Spielfeld kann mit etwas Aufwand für weitere Spiele dieser Art zugänglich gemacht werden. Denkbar sind jegliche Arten von Brettspielen.

Kapitel 3

Technischer Bericht

3.1 Technologieentscheide

Das Projekt kann in folgende individuelle Teilprojekte unterteilt werden:

- Objekterkennung als Input für das Spiel
- Aufbereitung der Daten für die Physics-Engine und das Grafische User-Interface
- Physics-Engine
- Grafisches User-Interface

Um schnell einen Proof of Concept zu erreichen, war es wichtig, Technologieentscheide früh zu fällen.

Als Sensor entschieden wir uns für die Kinect mit dem Kinect-SDK von Microsoft, da diese neben einer herkömmlichen Kamera auch einen genügend genauen Tiefensensor [Khos2011] und eine Gelenkserkennung mit ausreichender Framerate bereitstellt (siehe 3.2.3). Es wäre auch möglich, nur mit Farbbildern und Algorithmen eine dreidimensionale Ansicht des Spiels zu gewinnen. Dies hätte jedoch den Rahmen dieser Arbeit gesprengt.
Als Technologie die Aufsehen erregt und Technikfreaks begeistert nutzten wir die Gelegenheit, um uns vertieft mit der Kinect auseinanderzusetzen.

Abbildung 3.1: Kinect Logo

Damit stand auch fest, dass C# 4.0 als Progammiersprache und Visual Studio 2010 als Entwicklungsumgebung eingesetzt wird. Weiter setzten wir den Resharper als Productivity- and Refactoring-Plugin ein.

Abbildung 3.2: Visual Studio 2010 und Resharper Logo

Beim Recherchieren nach einem geeigneten GUI-Framework stellte sich heraus, dass sich XNA besonders gut für unsere Bedürfnisse eignet. XNA bietet eine integrierte Laufzeitumgebung speziell für Games für verschiedene Microsoft Plattformen, u.a. Windows Vista, Windows 7, Windows Phone und XBox. Zum einen erspart uns dies ein individuelles Memory-Management, zum anderen würde es ein allfälliges Portieren auf die erwähnten Plattformen erleichtern.
Das XNA-Framework bietet sowohl für 2D- als auch für 3D-Applikationen Unterstützung. Ersteres erwies sich jedoch als komplizierter, als man es sich von 2D-Zeichnungsroutinen in Java oder Flash gewohnt ist. Grösste Stärke des Frameworks ist der Support diverser Grafikkarten. XNA lagert das Rendering der Applikation bestmöglich auf den Grafikprozessor aus. Dies ist eine willkommene Entlastung, da die Verarbeitung des Kinect-Inputs die CPU ohnehin schon erheblich beansprucht.

Bezüglich der Physics-Engine haben wir uns für Farseer Physics, ein Port der berühmten Box2D-Library für C#, entschieden. Als weit verbreitet und dadurch auch gut dokumentierte Funktionsbibliothek, eignet sie sich besonders gut.
Als Computer-Vision Library entschieden wir uns für Emgu, ein C# Wrapper der in C/C++ geschriebenen OpenCV Library, mit der wir bereits aus der Studienarbeitszeit vertraut sind.

Abbildung 3.3: XNA, Farseer Physics und Emgu Logo

Bezüglich Logging entschieden wir uns für log4net, die .NET Variante von log4j die es ermöglicht, Nachrichten nach Wichtigkeit, Namensräume und Klassen zu filtern.

Abbildung 3.4: Logging Services

Um aus der XML-Dokumentation im Code eine Code-Dokumentation zu generieren kommt Sandcastle zum Einsatz.

3.2 Analyse

3.2.1 Spielaufbau

Um keine speziellen Vorrichtungen für den Spielaufbau vorauszusetzen, werden der Beamer und die Kinect idealerweise auf einem Tisch platziert. Der Beamer projiziert schräg auf den Boden. Mittels Keystone-Korrektur[1] erscheint das Spielfeld wie in Abb. 3.5 rechteckig auf dem Boden. Die Kinect befindet sich idealerweise etwas erhöht auf einem Tisch. Die Kinect darf nicht zu flach auf das Spielfeld gerichtet sein, da ansonsten zu viele wertvolle Informationen über das Spielfeld verloren gehen und es nicht mehr als Rechteck erkannt wird. Andererseits darf die Kinect nicht zu steil auf das Spielfeld zeigen, da ansonsten der Spieler nicht an jeder Stelle des Spielfeldes erkannt wird. Um Ungenauigkeiten bei der Spielererkennung zu vermeiden, muss genügend Platz im Raum vorhanden sein (3.2.3.3).

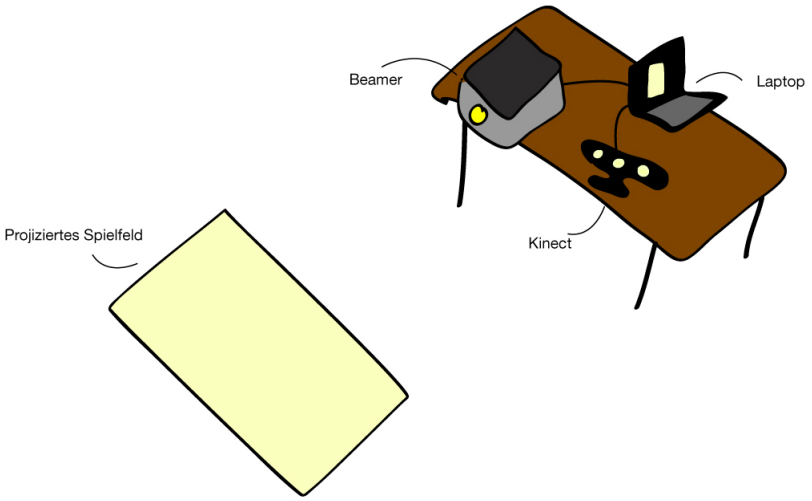

Abbildung 3.5: Spielaufbau

Das Spielfeld ist so zu gestalten, dass möglichst kein Schatten auf das Spielfeld geworfen wird. Weiter wirken sich Schatten auch bei der Hinderniserkennung negativ aus.

Je nach Material des Bodens kann es zu Reflexionen verschiedener Lichtquellen kommen, die das Erkennen des Spielfelds erschweren oder verunmöglichen. Am einfachsten lässt sich dies vermeiden, indem man eine lichtabsorbierende Unterlage, z.B. einen Teppich, verwendet.

3.2.1.1 Schlagerkennung

Variante I: Virtueller Schläger Das Kinect-SDK bietet eine Skeletonerkennung an, die es erlaubt, den Spieler und seine wichtigsten Gelenke zu erkennen. Mithilfe der 3D-Koordinaten dieser Gelenke kann die Haltung eines Golfers erahnt und ein virtueller Schläger am richtigen Ort im richtigen Winkel platziert werden. Die Erkennung der Gelenke ist jedoch nicht sonderlich stabil. Daher kommt es vor, dass Gelenke ruckeln, was zu einem Zittern des Schlägers führt. Dieses Problem kann mit der Methode der kleinsten Quadrate minimiert

[1] Korrektur einer trapezförmigen Verzerrung eines projizierten Bildes [MICR].

werden (siehe 4.2) . Um Fehler zu minimieren, müssen unter Umständen einige Werte zwischengespeichert werden, was eine Verzögerung der Reaktionszeit zur Folge hat. Der grosse Vorteil ist dagegen die bessere Kontrolle über den Schläger.
Die Kinect Skeletonerkennung arbeitet mit 30 Hz (siehe 3.2.3). Dies bedeutet, dass pro 33 ms ein Messwert erstellt wird. Wird versucht, den Ball zu schnell anschlagen, kann es vorkommen, dass er genau überhüpft wird. Daher kann künstlich ein Zwischenwert zwischen zwei Positionen berechnet werden, was dann eine Verzögerung von ca. 16 ms zur Folge hat.

Variante II: Realer Schläger Die Alternative zum virtuellen Schläger birgt einige Vor- und Nachteile:

+ ein realer Schläger gibt dem Spieler mehr Kontrolle über das Spiel
+ mit geeigneter Hardware könnte ein Widerstand beim Ballanschlag simuliert werden
- das System erkennt den Schläger nicht an seiner exakten Position → ein Offset entsteht
- der Spieler benötigt einen Schläger um zu spielen

Variante III: Kombination virtueller und realer Schläger Der Spieler hat lediglich ein Griff in der Hand, der durch Vibration einen Ballanschlag simuliert (analog zum Nintendo Wii Controller).

3.2.2 Beobachterproblem

Unter der Voraussetzung, dass der Beamer und die Kinect frei im Raum positioniert werden dürfen, ergeben sich zwei Teilprobleme:

1. Das reale Spielfeld muss im System virtuell abgebildet werden, so dass die Proportionen erhalten bleiben
2. Falls der Beamer keine automatische Keystone-Korrektur durchführen kann, muss dies das System vornehmen, bevor es das Bild an den Beamer weitergibt.

Das erste Problem wird dadurch vereinfacht, dass die Kinect zu jedem 2D-Punkt die entsprechenden 3D-Koordinaten liefert. Damit wird es möglich, mittels vektorgeometrischen Umrechnungen das aufgenommene 3D-Spielfeld zweidimensional zu rekonstruieren. Eine detaillierte Herleitung ist unter 3.3.5.1 zu finden.

Die Keystone-Korrektur ist dagegen etwas komplizierter. Ziel ist es, das Ausgangsbild mittels Pre-Warping[2] so zu transformieren, dass z.B. ein Quadrat auf dem Bildschirm wieder als solches am Boden erscheint. Wir haben zwei Lösungsansätze etwas genauer angeschaut:

Ermittlung der perspektivischen Transformation Die Arbeit von Rahul Sukthankar et. al. [Sukt2000] zeigt, wie mit Hilfe einer handelsüblichen Kamera und einer perspektivischen Transformationen eine Keystone-Korrektur realisiert werden kann. Der Umstand, dass dank der Kinect nicht nur 2D- sondern auch 3D-Koordinaten zur Verfügen stehen, würde diese Umrechnung sogar noch vereinfachen.

[2]Unter Pre-Warping versteht man die Verzerrung eines Bildes, bevor es auf dem Bildschirm gezeichnet wird.

Ermittlung der Beamer-Position Anstatt die Darstellung des Spielfeld mit 2D-Routinen zu zeichnen, könnte das Potenzial einer 3D-Engine, so wie sie auch in XNA verwendet wird, ausgenutzt werden. Dabei könnte man die Spielebene im dreidimensionalen Raum zeichnen und die virtuelle Kamera der 3D-Engine so platzieren, dass sie die Position des realen Beamers einnimmt. Die 3D-Engine würde dann das Bild automatisch perspektivisch richtig rendern, was eine manuelle Tranformation überflüssig machen würde.

3.2.3 Kinect

Die Kinect besteht aus einer Kamera, einem 3D-Mikrofon und einem PrimeSense-Tiefensensor[3].

3.2.3.1 PrimeSense-Tiefensensor

Kourosh Khoshelham beschreibt die funktionsweise des Tiefensensors in seinem Paper „Accuracy analysis of kinect depth data“ kurz und prägnant.
The Kinect sensor consists of an infrared laser emitter, an infrared camera and an RGB camera. The inventors describe the measurement of depth as a triangulation process (Freedman et al., 2010). The laser source emits a single beam which is split into multiple beams by a diffraction grating to create a constant pattern of speckles projected onto the scene. This pattern is captured by the infrared camera and is correlated against a reference pattern. The reference pattern is obtained by capturing a plane at a known distance from the sensor, and is stored in the memory of the sensor. When a speckle is projected on an object whose distance to the sensor is smaller or larger than that of the reference plane the position of the speckle in the infrared image will be shifted in the direction of the baseline between the laser projector and the perspective centre of the infrared camera. These shifts are measured for all speckles by a simple image correlation procedure, which yields a disparity image. For each pixel the distance to the sensor can then be retrieved from the corresponding disparity, as described in the next section. Figure 1 illustrates the depth measurement from the speckle pattern. [Khos2011] [4]

3.2.3.2 Streams

Die Kinect bietet drei verschiedene Streams an. Der `ColorImageStream` liefert Bilddaten, der `DepthImageStream` liefert Tiefendaten und der `SkeletonStream` liefert 3D-Punkte zu den erkannten Gelenkspunkten. Diese Streams können einzeln aktiviert und deaktivert werden. Dabei gibt es zwei Möglichkeiten zur Ansteuerung. Einerseits kann ein Eventhandler pro Stream registriert werden und sobald Daten verfügbar sind, wird ein Event geworfen. Über das Eventargument gelangt man zum entsprechenden Frame, das die gewünschten Daten enthält. Diese Frames sind „Unmanaged Ressources“[5] und müssen nach Gebrauch selber geschlossen werden. Andererseits kann der Streams direkt mit `OpenNextFrame()` gepollt werden. Durch Angabe eines `int` wird bestimmt, wie viele Millisekunden auf ein Frame gewartet wird, bevor ein Timeout entsteht und `null` zurückgeliefert wird.
Es kann auf ein Stream nicht gleichzeitig via Events und Polling zugegriffen werden. Jedoch kann beispielsweise der `ColorImageStream` gepollt und der `SkeletonStream` mit Events angesteuert werden.

[3] Prime Sense ist ein israelische Firma die für die Kinect den Tiefensensor liefert.
[4] Er wiederum bezieht sich auf das Patent [Free2010] das um einiges umfangreicher ist.
[5] Diese werden vom Garbage Collector nicht aufgeräumt.

`ColorImageStream` ist in verschiedenen Auflösungen verfügbar, nähmlich `RgbResolution640x480Fps30` und `RgbResolution1280x960Fps12`. Der Stream wird aktiviert indem die gewünschte Auflösung angegeben wird.

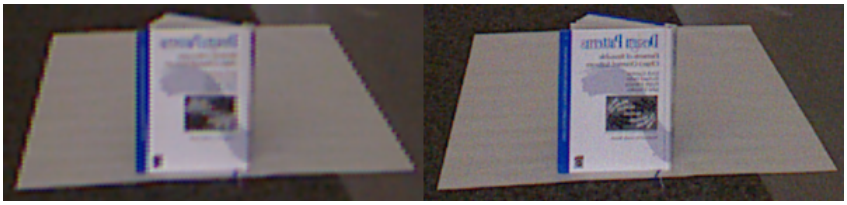

Abbildung 3.6: Vergleich Bildausschnitt 640x480 zu 1280x960

Wird in Zukunft von einem „Bild“ gesprochen, ist ein Farbbild der Kinect gemeint, wie es in Abb. 3.6 zu sehen ist.

`DepthImageStream` ist ebenfalls in verschiedenen Auflösungen verfügbar `Resolution640x480Fps30`, `Resolution320x240Fps30` und `Resolution80x60Fps30`. Es werden verschiedene Varianten angeboten, um von einem Farbpixel die entsprechenden Tiefendaten zu erhalten. Wir benötigen lediglich `MapToSkeletonPoint`. Tiefenwerte werden ab einer Distanz von 0.40m geliefert. Der Fehler wächst mit steigender Distanz exponentiell an. Ab 5m beträgt er über 0.04m [Khos2011]. Wie in Tab. 3.1 zu sehen ist, ist die Gültigkeit der Kinectdaten beschränkt und so muss bei jedem Aufruf von `MapToSkeletonPoint` geprüft werden, ob gültige Koordinaten zurückgeliefert wurden, was durch einen negativen `Z`-Wert signalisiert ist. Die Daten werden von der Kinect nur gewisse Zeit gehalten. Danach werden die Referenzen ungültig.

Format	Verfügbarkeit
`ColorImageFormat.RgbResolution1280x960Fps12`	$\sim 200ms$ verfügbar
`ColorImageFormat.RgbResolution640x480Fps30`	$\sim 70ms$ verfügbar
`DepthImageFormat.Resolution640x480Fps30`	$\sim 70ms$ verfügbar

Tabelle 3.1: Gültigkeitsdauer Kinect-Daten

In Abb. 3.7 sind die Reichweiten der zwei verfügbaren Modi für den `DepthImageStream.Range` zu sehen. Wir benötigen möglichst viel Reichweite und aktivieren dazu den `DefaultRange`.

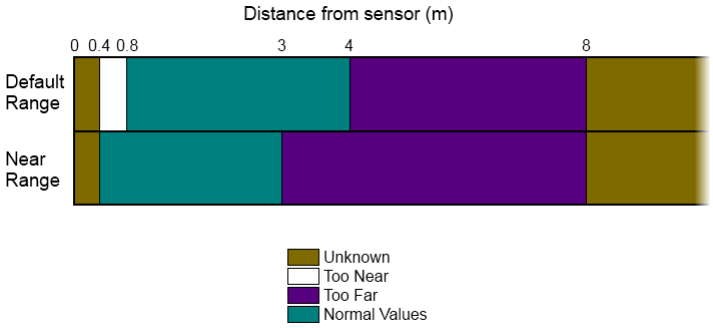

Abbildung 3.7: Gültigkeitsbereich Kinectsensor [KSDK2012]

Abbildung 3.8: Tiefenbild mit zusätzlicher Skeletonvisualisierung

Im Gegensatz zum „Bild“ des `ColorImageStreams` wird das „Tiefenbild“ des `DepthImageStream` in Zukunft explizit als solches bezeichnet. Mit „Tiefendaten“ sind die aus dem Tiefenbild gewonnenen Daten gemeint.

`SkeletonStream` nimmt als einzigen und optionalen Parameter `TransformSmoothParameters` entgegen, über die die Skeletonerkennung justiert wird. Die Parameter gelten global für jedes Gelenk (siehe Tab. 3.2).

Parameter	Wert	Bedeutung
`Correction`	0 - 1	Nimmt eine Zahl zwischen 0 und 1 entgegen. Je kleiner die Zahl, desto mehr wird das Zittern korrigiert.
`Prediction`	0.0 +	Die Anzahl der Frames die vorausgesagt werden.
`Smoothing`	0 - 1	Je höher die Zahl, desto stärker ist die Glättung.
`JitterRadius`	in Meter	Setzt einen Radius auf den vorherigen Punkt. Befindet sich der neue Punkt innerhalb des Radius, so wird er geglättet. Jeder Punkt ausserhalb dieses Radius wird auf den Radius gesetzt und anschliessend geglättet (siehe auch `MaxDeviationRadius`).
`MaxDeviationRadius`	in Meter	Gehört zur `JitterRadius`-Einstellung. Geht vom neuen, ursprünglich ungeglätteten Punkt aus und vergleicht ihn mit dem geglätteten Punkt. Liegt der geglättete Punkt ausserhalb des `MaxDeviationRadius` des ursprünglichen Punktes, wird der neue Punkt auf dem `MaxDeviationRadius` in Richtung des geglätteten Punktes gesetzt.

Tabelle 3.2: Parametrisierung `TransformSmoothParameters` [KSDK2012]

Genaueres zur Parametrisierung ist im Anhang unter 4.1 aufgeführt.

3.2.3.3 Skeletonerkennung

Wie genau die Skeletonerkennung der Microsoft-Implementation funktioniert, konnte mangels Quellen nicht ermittelt werden.
Eine mögliches „Skeletal Tracking“ des Oberkörpers ist jedoch im Paper von Abhishek Kar dokumentiert. Dabei wird eine Foregroundsegmentation[Kim2006] auf dem Tiefenbild durchgeführt um den irrelevanten Hintergrund auszublenden. Anschliessend wird auf dem Farbbild eine Face- and Uppderbodydetection nach Viola-Jones [Viol2001] durchgeführt. Anhand dieser Gesichts- und Torsoerkennung sind die ersten Fixpunkte des Skeletons bekannt. Mithilfe von Antropometrie[6] und „Skin Color Segmentation“[7] werden die Arme eingepasst.[Kar2011]
Tests mit dem Micrsoft Kinect SDK zeigen, dass der Skeleton erst erkannt wird, wenn ein Gesicht auf dem Farbbild zu sehen ist, wobei der Skeleton erhalten bleibt, wenn das Gesicht wieder verschwindet. Die Implementation von Microsoft und diejenige von Abishek Kar scheinen also ähnlich zu funktionieren.

Um Fehler zu minimieren empfiehlt es sich, genügend Platz für die Spielfläche zur Verfügung zu haben, da Tischbeine o.Ä. das Tracking negativ beeinflussen.
Ausserdem ist das Kinect SDK fähig, mehrere Skeletons zu erkennen. Anhand der gelieferten Daten lassen sich unerwünschte Skeletons einfach herausfiltern, zumal oft nur die am nächsten zur Kinect stehende oder die aktivste Person relevant ist.

[6] Die Lehre der Vermessung des menschlichen Körpers.
[7] Dabei wird versucht alle Teile des Bildes zu Filtern auf denen keine Hautfarbe vorhanden ist. Damit lassen sich irrelevante Bereiche für Trackings ausblenden. [Iraj2011]

3.2.3.4 Ermittlung der Tiefendaten

Die Kinect liefert zu jedem Pixel im Farbbild die 3D-Koordinaten. Obwohl die Tiefenbilder eine maximale Auflösung von 640x480 haben, ist es sinnvoll, die qualitativ eher schlechten Farbbilder mit der höchsten Auflösung aufzunehmen, um an mehr Details bei der Hindernis- oder Spielfelderkennung heranzukommen (siehe Abb. 3.6).
Die Framerate der höchsten Bildauflösung beträgt maximal 12 Hz und drosselt damit den `SkeletonStream` auf diese Rate herunter. Dies ist deutlich zu wenig für ein Spiel, daher muss jeweils zwischen zwei Modi gewechselt werden. Für die Kalibrierung und Erkennung der Hindernisse werden `Color-` und `DepthImageStream` verwendet. Für das Spiel selbst ist nachher nur der `SkeletonStream` aktiv, welcher bis zu 30 Frames pro Sekunde liefern kann.

3.2.3.5 Kompatibilität

Bei Tests wurden Kompatibilitätsprobleme mit USB3-Anschlüssen festgestellt. Zwar wurde die Kinect richtig erkannt, jedoch reagiert sie nur mit grosser Verzögerung.

3.2.3.6 Threads

Die Kinect bzw. das SDK arbeitet mit `Background M-NUI Threads`. Diese führen registrierte Eventhandler aus oder sorgen für die Aufbereitung der Daten der einzelnen Streams.

3.3 Implementation

3.3.1 Domain Model

Das `Domain Model` in Abb. 3.9 bietet eine Übersicht über alle Packages des Projekts.

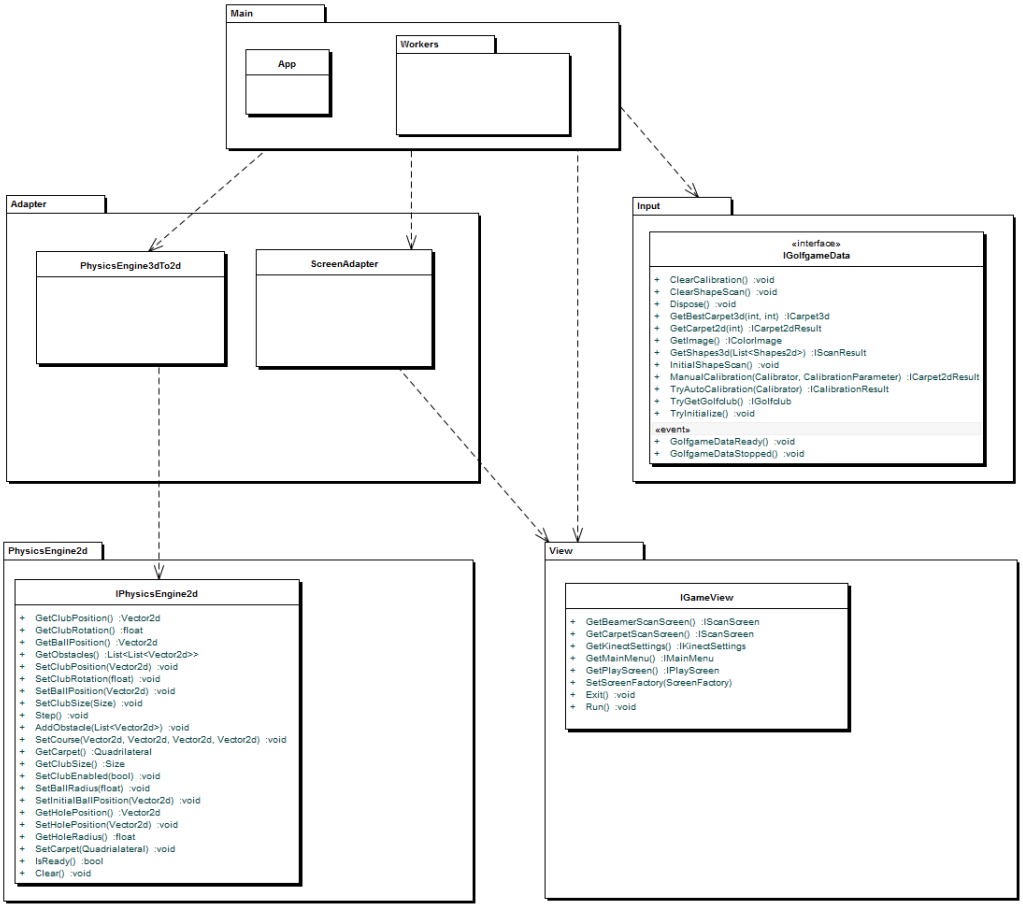

Abbildung 3.9: Domain Model

`Main` Die Hauptkomponente des Projekts, sie steuert den Programmablauf.

`Input` Dieses Package bereitet die Inputdaten für die anderen Schichten auf.

`Adapter` Die Konversion von Einheiten sowie die Umrechnung von 3D auf 2D geschieht hier.

`PhysicsEngine` Das korrekte physikalische Verhalten der einzelnen Spielkomponenten wird von die `PhysicsEngine` berechnet.

`View` Beinhaltet alle User-Interface-Komponenten des Spiels.

3.3.2 Technologieübersicht

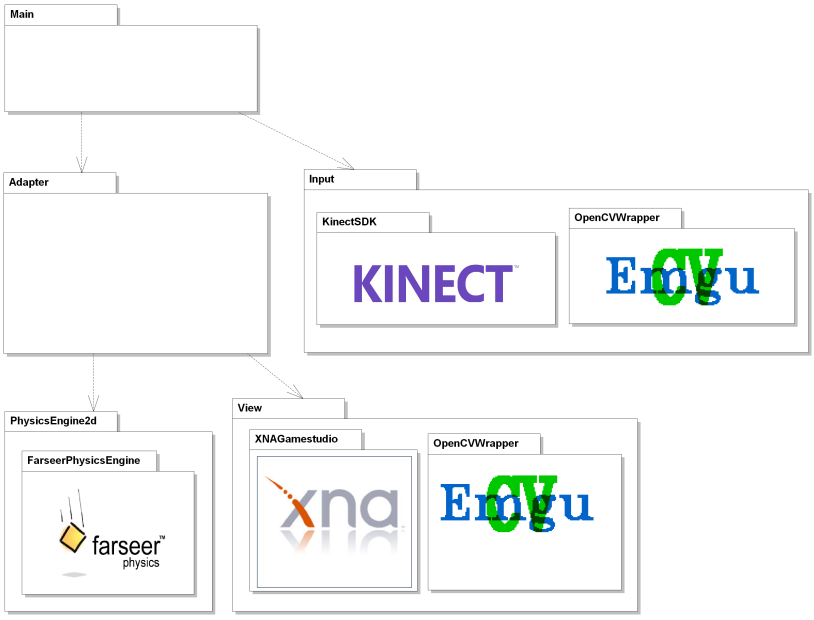

Abbildung 3.10: Technologien

3.3.3 `Input`

Das `Input`-Package ist eine Abstraktion, die den Zugriff auf die Kinect kapselt und den Bedürfnissen des Spiels anpasst.

3.3.3.1 Programmfluss

Anfangs wurde ein asynchroner Programmfluss realisiert bei dem sich der Glue-Code im `Main`-Package für diverse Events im Input registrierte. Die Übersichtlichkeit des Programmcodes litt aber unter dem ständigen Eventhandler registrieren und deregistrieren. Deshalb wurde der Programmfluss zu einer synchronen Variante umgeschrieben. Dieser Designentscheid hat sich sehr gelohnt. Der Programmfluss ist so verständlicher und besser kontrollierbar. Was vorher durch mehrere Zeilen Code realisiert wurde, konnte so durch einen einfachen Methodenaufruf ersetzt werden.

3.3.3.2 Datenquelle

Als Datenquelle dienen die verschiedenen Streams der Kinect. Die Kinect wurde durch die `KinectManager`-Klasse gekapselt um nur die nötige Funktionalität zu propagieren.

3.3.3.3 Das Spielfeld

Anstatt das Spielfeld per Beamer zu projizieren, kann zu Versuchszwecken auch ein grosses, weisses Blatt Papier auf den Boden gelegt werden.

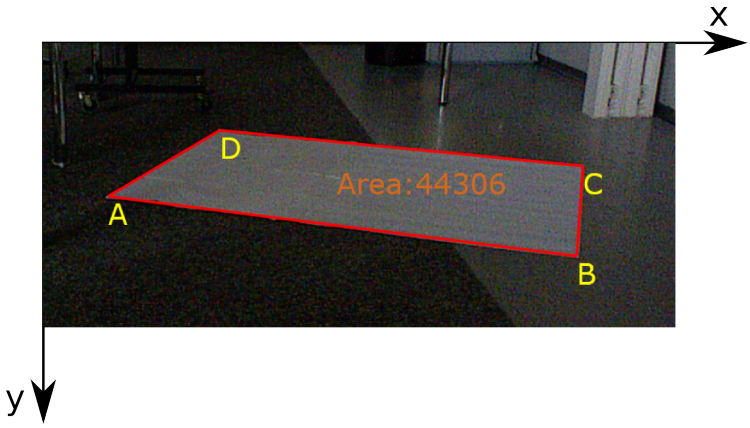

Abbildung 3.11: Das Spielfeld in 2D - `Carpet2d` in Pixel

Dieses Spielfeld wird von der Kinect aufgenommen und mittels Imageprocessing erkannt (3.3.3.4). Dafür müssen die Kinect richtig positioniert und gegebenenfalls die Kalibrierungsparameter angepasst werden. Diese Anpassung kann manuell (`ManualCalibration`) oder automatisch (`TryAutoCalibration`) ausgeführt werden. Die manuelle Kalibration setzt lediglich auf das Auge des Betrachters (Abb. 3.11). Die automatische Kalibrierung ist ein BruteForce-Vorgang, bei dem solange die Werte Schritt für Schritt angepasst werden, bis ein Resultat gefunden wurde. Grenzwerte definieren das Spektrum der zu verwendenden Parameter. Um bei mehreren möglichen Resultaten das Beste auszuwählen, wird ihre Rechteckigkeit als Indikator verwendet (3.3.3.5).

3.3.3.4 Spielfelderkennung 2D

Um das Spielfeld zu erkennen, wird die Konturensuche von OpenCV verwendet [Brad2008], die als Input ein Binärbild verlangt (siehe weiter unten). Anschliessend wird jede gefundene Kontur zu einem Polygon approximiert und anhand dessen Eigenschaften entschieden, ob es sich um ein Spielfeld handelt. Wenn ja, wird die Suche abgebrochen und das Spielfeld als Resultat zurückgegeben. Die entscheidenden Merkmale des 2D-Spielfelds sind:

- 4 Ecken: mehr oder weniger Ecken können ausgeschlossen werden
- eine Mindestfläche: eine Vielzahl kleiner Konturen kann so ignoriert werden
- die Eigenschaft der 4 Winkel: bei extrem spitzen oder stumpfen Winkeln kann ein Spielfeld ausgeschlossen werden
- konvexe Defekte: hat eine Kontur einen konvexen Defekt, kann ein Spielfeld ebenfalls ausgeschlossen werden (Abb. 3.12).

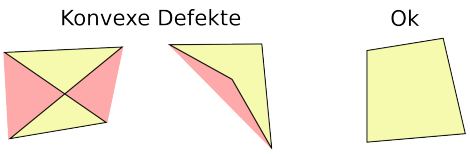

Abbildung 3.12: Konvexe Defekte

Um eine Kontur anhand verschiedener Eigenschaften zu beurteilen, wurde das Interpreter Pattern implementiert (3.3.3.6).

Als Eingabe für die OpenCV-Konturensuche wurden zwei Arten von Binärbildern mit unterschiedlichen Vor- und Nachteilen eingesetzt.

`CannyStrategy` Dieser Ansatz beruht auf der von John Canny entwickelten Kantendetektion [Cann1986] (siehe Abb. 3.13).

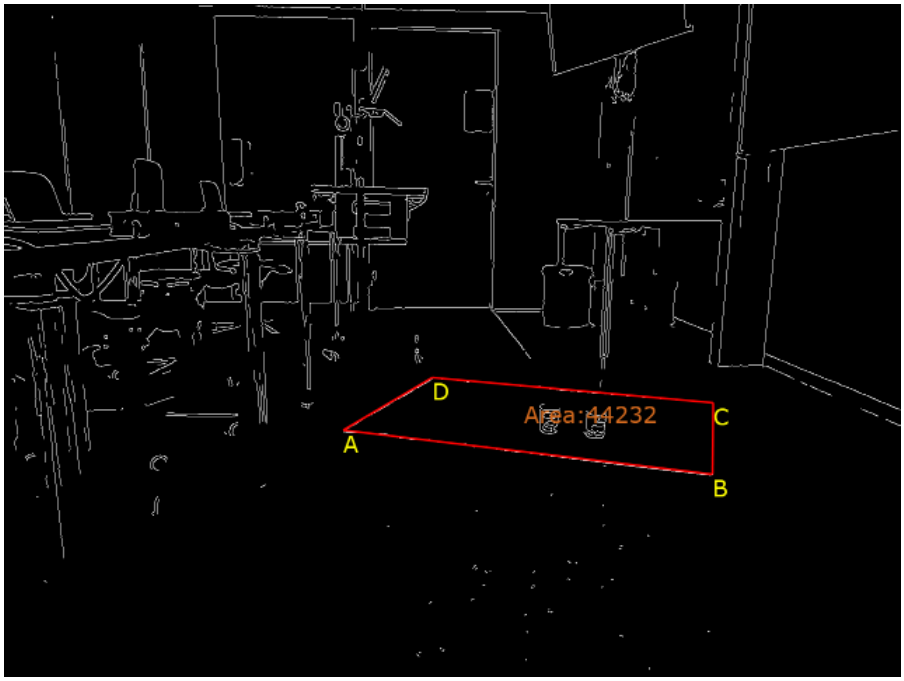

Abbildung 3.13: Canny Edge Detection

Vor-/Nachteile

- aufwendigere Parametrisierung
- Flimmern der Kinectkamera führt zu Zufallsergebnissen
- rechenaufwändig (prinzipiell aber vernachlässigbar, da kein fortlaufender Prozess)
- Verhalten abhängig vom Untergrund
  - Beläge mit Struktur wie Teppich
  + einfarbige, solide Beläge

`ThresholdStrategy` Hierbei wird der Helligkeitsunterschied des Spielfeldes gegenüber der restlichen Umgebung als Indikator genutzt. Ein Grenzwert entscheidet, ob ein Pixel schwarz oder weiss wird (siehe Abb. 3.14).

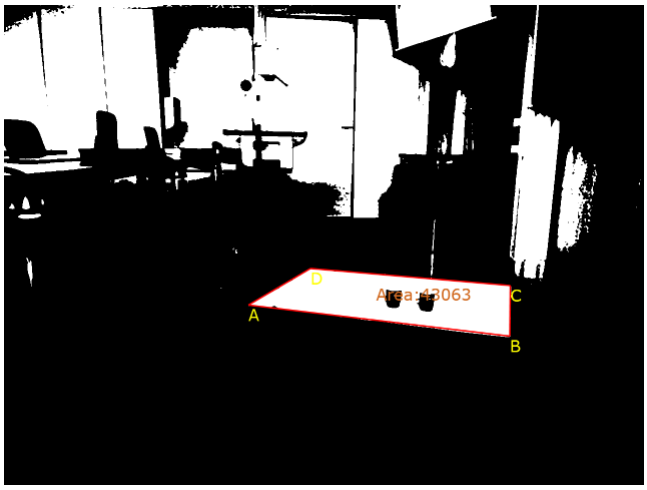

Abbildung 3.14: Threshold

Ein Histogramm über die Farbwerte könnte zusätzlich an dieser Stelle den Start- und Endwert der automatischen Kalibrierung erkennen und so den Vorgang beschleunigen (nicht implementiert).

Vor-/Nachteile

- anfällig auf Veränderungen der Lichtverhältnisse und Reflexionen am Boden
- problematisch bei hellem Untergrund
+ stabil gegenüber Bodenstruktur und Unebenheiten des Spielfelds[8]

Während den Arbeiten hat sich dieser Ansatz als effizienter und stabiler erwiesen, obschon sich bei Reflexionen am Boden die `CannyStrategy` besser eignet.

[8] Als Spielfeld Prototyp wurde ein grosses, weisses Blatt Papier eingesetzt. Sobald dies uneben ist, entstanden störende Schatten und Kanten, bei denen die `CannyStrategy` fehleranfällig reagierte.

3.3.3.5 Spielfelderkennung 3D

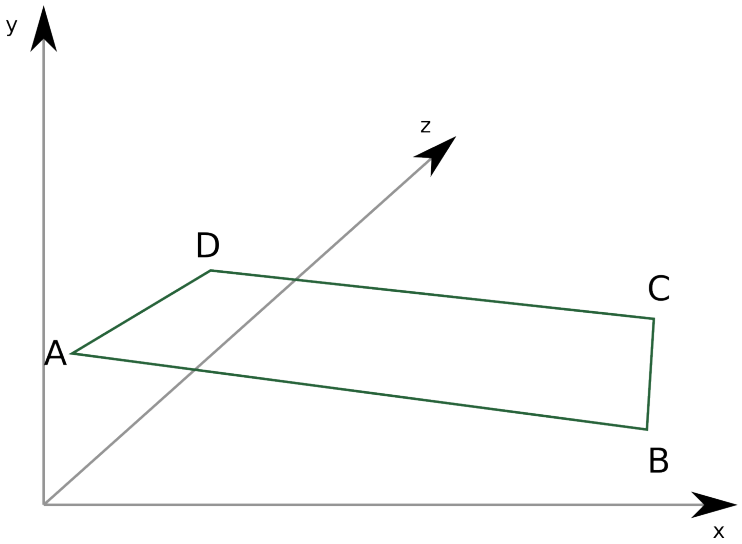

Abbildung 3.15: Das Spielfeld in 3D - `Carpet3d` in Meter

Mithilfe des Kinect SDKs lassen sich die gefundenen zweidimensionalen Eckpunkte in Pixel auf dreidimensionale Punkte in Meter abbilden (siehe Abb. 3.15). Mit der zusätzlichen Dimension lässt sich feststellen, wie rechteckig das Spielfeld ist, indem die Skalarprodukte aller Winkel summiert werden. Ist der Fehler zu gross, wird das Resultat verworfen. Dabei müssen die Messungenauigkeiten der Kinect berücksichtigt werden.

3.3.3.6 Hindernisse

Sobald Ort und Grösse des Spielfeldes bekannt sind, ist es möglich, das Spielfeld nach Hindernissen abzusuchen (`ShapeScanner`). Hindernisse werden mit einem Graustufendifferenzbild[9] bestimmt. Zuerst wird ein Graustufenbild des Spielfeldes ohne Hindernisse gemacht, anschliessend werden die Hindernisse ins Spielfeld gelegt und ein weiteres Graustufenbild erstellt. Aus der Differenz der beiden Bilder wird ein neues Bild erstellt. Um Rauschen zu unterdrücken, werden minimale Bildunterschiede durch Thresholding entfernt. Anschliessend werden wie unter 3.3.3.4 beschrieben, Konturen auf dem resultierenden binären Bild gesucht (Abb. 3.16).

[9] Es werden leicht verwischte Graustufenbilder verwendet, um das rauschen zu unterdrücken.

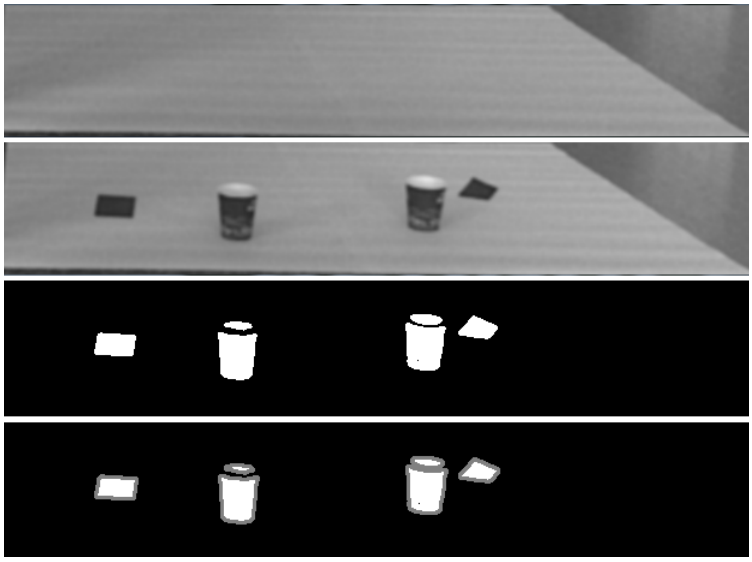

Abbildung 3.16: Suche nach Hindernissen von oben nach unten. Bild ohne Hindernisse, Bild mit Hindernissen, entrauschtes Differenzbild, Konturensuche auf Differenzbild

Anforderung an Hindernisse Wie in Abb. 3.16 ersichtlich, eignet sich dieser Ansatz eher für Flache Hindernisse, da diese keinen „Schatten“ [10] werfen. Die Ungenauigkeit der Tiefendaten (3.2.3.2) führt beim Abbilden zu weiteren Fehlern. Kleine Hindernisse provozieren demnach verhältnismässig grosse Fehler. Weitere Beobachtungen zeigen, dass hohe Hindernisse zusätzliche Fehlerquellen bilden. Wird bei der Abbildung von 2D nach 3D versehens ein Punkt nicht auf dem hohen Objekt, sondern weiter hinten (Abb. 3.17) gemessen, erstreckt sich das Hindernis plötzlich über das halbe Spielfeld. Des Weiteren dürfen Hindernisse kein Licht in Richtung der Kinect reflektieren, da sie ansonsten aufgrund der Helligkeit vom Spielfeld nicht unterschieden werden können.

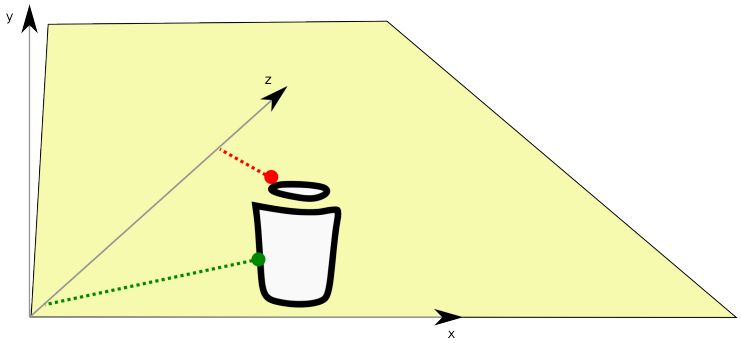

Abbildung 3.17: Ungenauigkeiten von Tiefendaten

Um gute Resultate zu erzielen, sollten deshalb ausschliesslich grosse, flache und nichtreflektierende Hindernisse verwendet werden.

[10] Mit Schatten ist der Bereich hinter dem Hindernis gemeint, der nicht rekonstruiert werden kann.

Korrektur Durch das ungenaue Abbilden auf 3D-Punkte entstehen Unebenheiten, die durch das spätere Aufskalieren in der View weiter verstärkt werden, wie in Abb. 3.18 zu sehen ist (siehe 3.3.5.2).

Abbildung 3.18: Transformierte Hindernisse

Je weniger Punkte abgebildet werden, desto weniger Fehler entstehen. OpenCV stellt eine Funktion bereit, die eine Kontur zu einem Polygon approximiert[11]. Diese kann dazu benutzt werden, um die Fehler der schlecht gemappten Konturen zu glätten. Damit wird versucht, geometrische Figuren als solche zu erkennen (siehe Abb. 3.19).

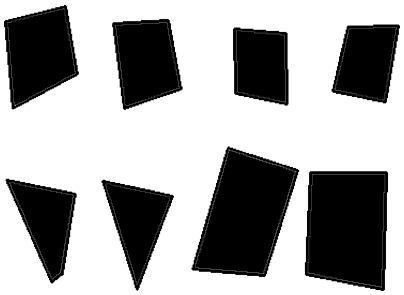

Abbildung 3.19: Transformierte Hindernisse zu Polygonen approximiert (lediglich die Positionen der vier inneren Hindernisse sind äquivalent zu Abb. 3.18)

Konturen Interpretieren Durch Eigenschaften wie Anzahl Ecken, Winkel und Grösse können Konturen klassifiziert werden. Somit ist es möglich, verschiedene Formen voneinander zu unterscheiden und diesen zukünftig besondere Eigenschaften wie Anziehungskraft oder Elastizität hinzuzufügen. Objekte können mit dem Interpreter Pattern [Gamm1995] bequem spezifiziert werden, indem einzelne Spezifikationen kombiniert werden.
Zum Beispiel:

- hat 5 Ecken
- hat eine Mindestfläche
- hat nur Winkel zwischen 60°und 90°
- hat keine konvexen Defekte

[11] `ApproxPoly` (Emgu) Sucht die charakteristischsten Punkte der Kontur, die aus der Distanz der einzelnen Punkte zueinander berechnet werden.[Brad2008]

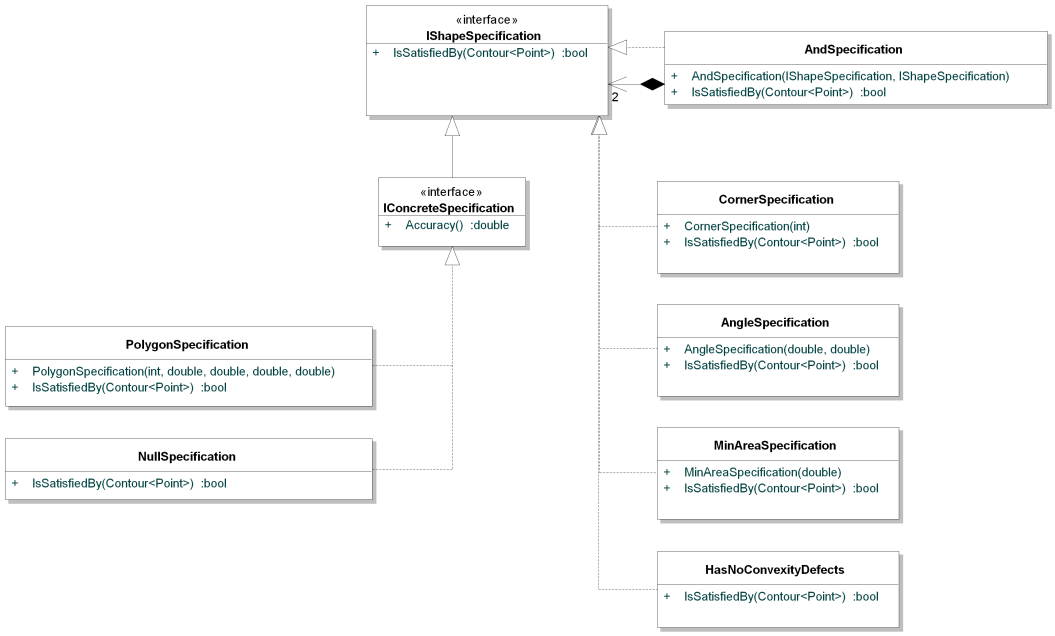

Abbildung 3.20: Nach Spezifikation Filtern

Das Interpreter Pattern wird durch `IConcreteSpecification` erweitert, um die Kontur nach Bedarf als Polygon zu approximieren.

Verbesserungsmöglichkeiten

Mappingproblem Das Problem aus Abb. 3.17 kann korrigiert werden, da die 3D-Daten bekannt sind. Somit kann bei einem höheren Hindernis mittels der `Z`- und `Y`-Werte herausgefunden werden, ob ein Punkt hinter dem Hindernis oder auf dem Hindernis gemappt wurde.

Nur ein Scan Wird mit dem Beamer eine weisse Fläche auf den Boden projiziert, so erkennt die Kinect beim Scannen sehr helle, wenn nicht gar komplett weisse Werte. Aufgrund dieser Umstände muss nicht zuerst ein erstes Bild gemacht werden, da klar ist, wie das Spielfeld in diesem Moment aussieht. Ein einzelner Scan des Spielfeld inklusive der Hindernisse reicht daher.

3.3.3.7 Der Golfschläger

Dank der Skeletonerkennung der Kinect lassen sich die Gelenkpunkte des Spielers als 3D-Daten auslesen.

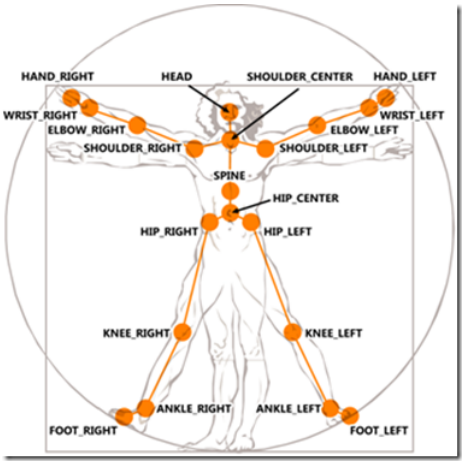

Abbildung 3.21: Ansteuerbare Gelenkpunkte der Kinect [KSDK2012]

Eine Analyse zur Genauigkeit der einzelnen Gelenkpunkte und zur `SkeletonStream`-Parametrisierung (3.2.3) findet sich im Anhang unter 4.1.

Position Wo mit beiden Händen hingezeigt wird, dort soll der Schläger erscheinen. Um dies zu bewerkstelligen, eignen sich intuitiv `Shoulder-`, `Elbow-`, `Wrist-` und `Hand`-Gelenkpunkte am besten. In einem ersten Prototyp wurde, wie in Abb. 3.22 illustriert, die am besten passendste Linie durch die Punkte gelegt. Diese Linie wird bis zum Boden erweitert und bestimmt so die Position des Schlägers. Genauere Informationen dazu im Anhang unter 4.2. Aufgrund der zufriedenstellenden Resultate, die mit dieser Methode erzielt werden, bleibt es bei dieser Implementation.

Winkel Beim Minigolfspielen steht der Spieler oft gerade vor dem Ball, so dass wenn eine Linie durch die beiden Füsse gezogen wird, der Schläger zum Lot dieser Linie steht (Abb. 3.3.3.7).

Gültigkeit Um die Gefahr eines unbeabsichtigten Ballanschlages zu verkleinern, wird festgestellt, ob die Hände nah beieinander liegen, was der typischen Haltung beim Golfen entspricht.

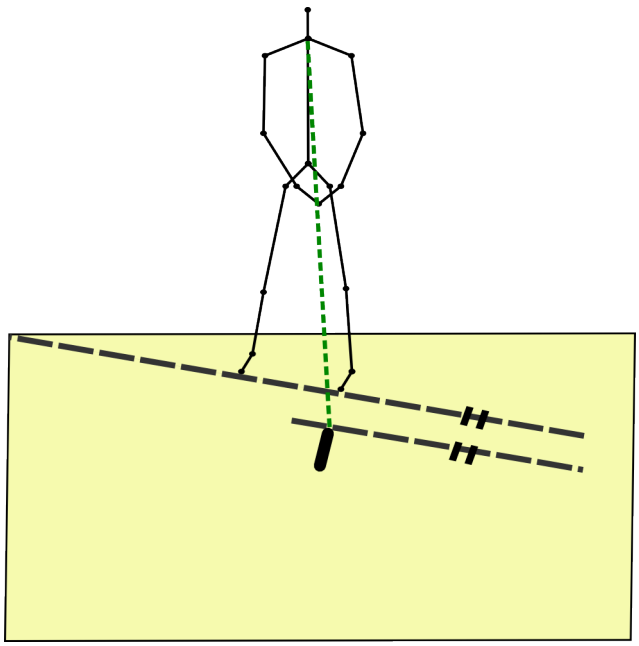

Abbildung 3.22: Winkel und Position des Golfschlägers

3.3.3.8 Resultatoptimierung

Da es möglich ist, dass Funktionen wie z.B. ein Spielfeldscan keine gültigen oder nichtoptimale Daten liefern, ist es sinnvoll, dies auf der Ebene der Datenaufbereitung zu optimieren. Gewünscht ist eine Funktion, die solange rechnet, bis ein annehmbares Resultat gefunden wurde. Dieses Vorgehen wird durch folgende Struktur realisiert:

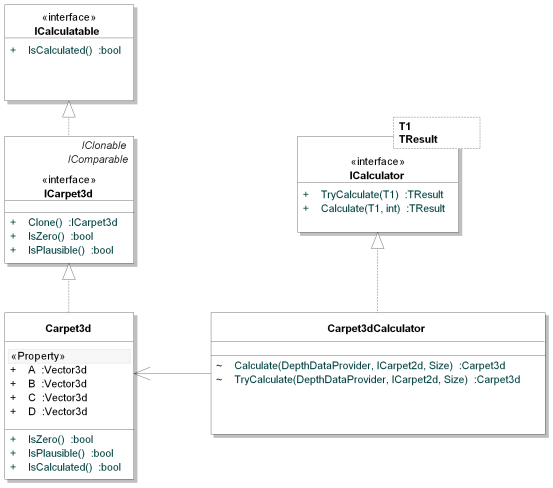

Abbildung 3.23: `ICalculator`, `ICalculatable` Beispiel an `Carpet3d`

Durch das Interface `ICalculatable` wird einer Klasse die Methode `IsCalculated()`

implementiert, die das Kriterium eines annehmbaren Resultats bestimmt. Das Interface `ICalculator` bestimmt die Berechnungsart mit `TryCalculate` für einen einzigen Berechnungsdurchgang und `Calculate` für eine Schleifenvariante.
Da es möglich ist, dass kein Resultat gefunden wird, bleibt das Programm an diesem Punkt stehen und läuft nicht weiter. Daher muss eine Abbruchbedingung für die Schleifenvariante festgelegt werden. Dabei gibt es zwei Ansätze, die verfolgt wurden:

Timeout Um die Laufzeit einer Funktion durch ein Timeout zu beschränken, startet `Thread A` einen `Thread B`, der die entsprechende Funktion ausführt. `Thread A` wartet bis `Thread B` joint und liefert das Resultat zurück oder ein Zeitlimit wird erreicht und eine `TimeoutException` wird geworfen. Im zweiten Fall läuft `Thread B` solange weiter, bis die Berechnung abgeschlossen ist. Dies kann fatale Folgen haben, wenn `Thread B` nie ein Resultat findet. Das kann passieren, wenn beispielsweise kein Spielfeld entdeckt wird. `Thread B` rechnet also weiter und belegt wertvolle Ressourcen. Eine Möglichkeit dies zu umgehen, ist das gewaltsame Beenden des `Thread B`, bevor die Exception geworfen wird. Allerdings kann dies hier zu Memory Leaks führen, da auf „Unmanaged Ressources“ zugegriffen wird.

Beschränkung der Ausführungen Die Angabe der maximalen Anzahl ungültiger Resultate ist eine annehmbare Abbruchbedingung. So wird nicht nach Zeit gemessen, sondern es kann z.B. bereits nach zehn ungültigen Versuchen die Annahme getroffen werden, dass kein plausibles Resultat mehr gefunden werden kann. Dabei muss auch kein `Thread A` auf `Thread B` warten, sondern `Thread A` führt die Schleife bis zur Abbruchbedingung aus. Hierbei muss allerdings darauf geachtet werden, dass `Thread A` an keiner Stelle blockiert wird.

Dieser Ansatz wurde mit der `CalculateUntilValid`-Extensionmethode implementiert, die solange Resultate produziert bis `IsCalculated true` liefert oder ein Grenzwert erreicht wurde. Dieses Konstrukt wird nur für die Spielfeldberechnung von `Carpet2d` und `Carpet3d` verwendet. Für die Berechnung der Golfschlägerposition lohnt sich dieses Konstrukt nicht. Ist der Spieler von der Kinect einmal erkannt, bleibt dies so.

3.3.3.9 Interface

Das nachfolgende Diagramm Abb. 3.24 zeigt die wichtigsten Klassen und Interfaces des Input-Packages und fasst diese kurz zusammen.

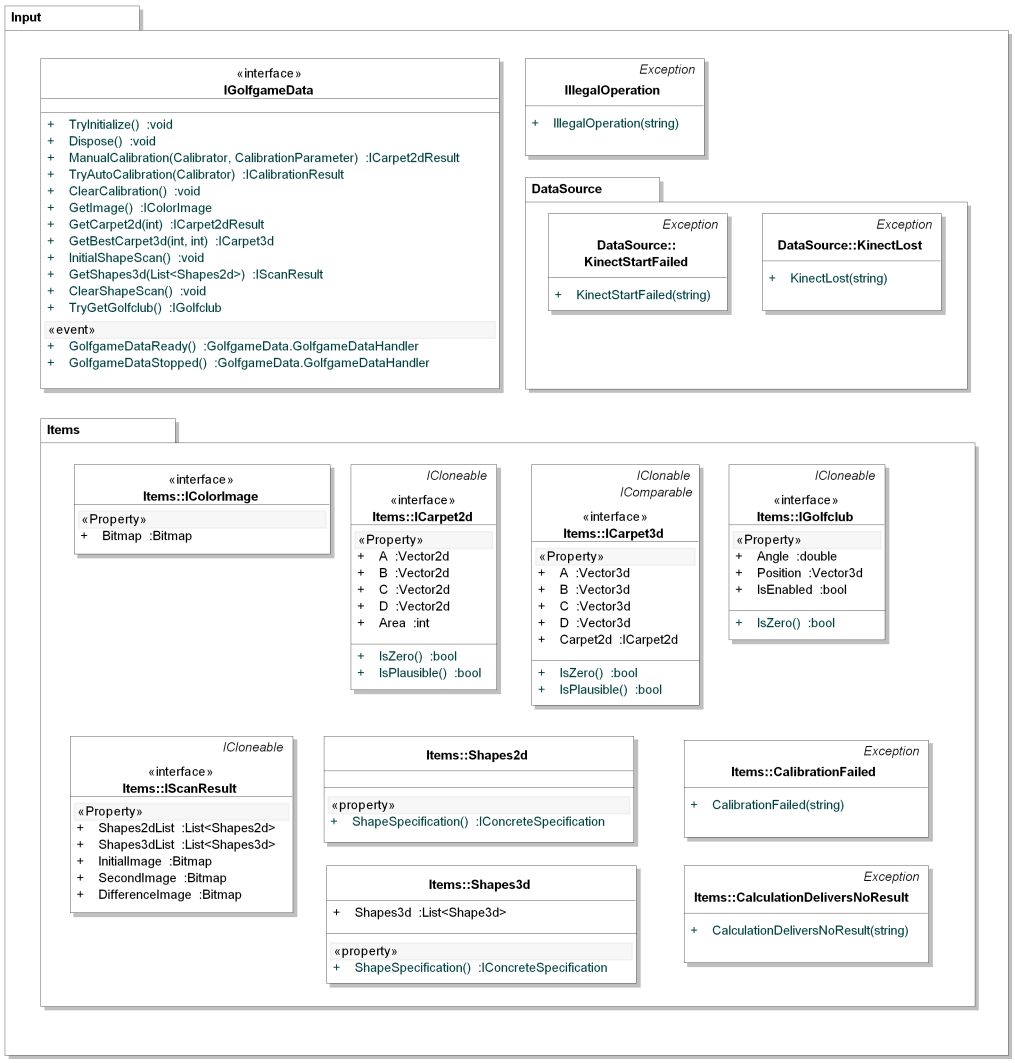

Abbildung 3.24: Input Package

`IColorImage` beinhaltet ein Bild der Kinect.

`ICarpet2d` eine Repräsentation des zweidimensionalen Spielfeldes, das von der Kinect verzerrt erkannt wurde (siehe Abb. 3.11).

`ICarpet3d` eine Repräsentation des dreidimensionalen Spielfeldes, das mithilfe der Tiefendaten ermittelt wurde (siehe Abb. 3.15).

`IGolfclub` repräsentiert die Position, die Ausrichtung und die Freigabe des Schlägers.

`IScanResult` hält die Resultate der Hindernissuche.

`Shapes2d` hält die Koordinaten der auf dem Bild gefundenen Konturpunkte, gefiltert nach der entsprechenden Spezifikation (siehe 3.3.3.6).

`Shapes3d` wie `Shapes2d` aber mit den gemappten 3D-Punkten.

`IllegalOperation` wird geworfen, wenn die aufgerufene Methode im aktuellen `State` (3.3.3.10) nicht zulässig ist. Es wurde bewusst nicht die `InvalidOperationException` verwendet, da diese aus Versehen durch eine andere Behandlung gefangen werden könnte.

`KinectStartFailed` wird geworfen, falls beim Programmstart keine Kinect gefunden wird.

`KinectLost` wird durch Ausstecken der Kinect während des Programmablaufs ausgelöst.

`CalibrationFailed` wird geworfen, wenn die automatische Kalibrierung keine Resultate liefert.

`CalculationDeliversNoResult` wird geworfen, wenn beispielsweise mangels brauchbarer Daten kein `Carpet3d` oder `Carpet2d` gefunden werden kann.

3.3.3.10 Koordination

Der Initialisierungsablauf des Spiels sowie die Modi der Kinect (3.2.3.4) müssen koordiniert werden. Um dieses Problem zu vereinfachen wurde das Statepattern [Gamm1995] implementiert. Die verfügbaren States sind in Abb. 3.25 ersichtlich. Das Statediagramm Abb. 3.26 zeigt die erlaubten Übergänge.

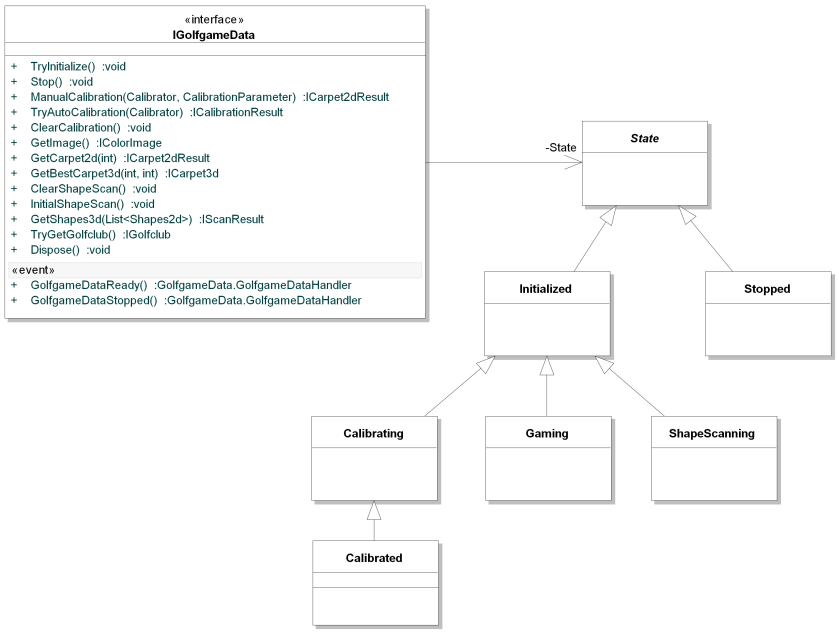

Abbildung 3.25: State Hierarchie

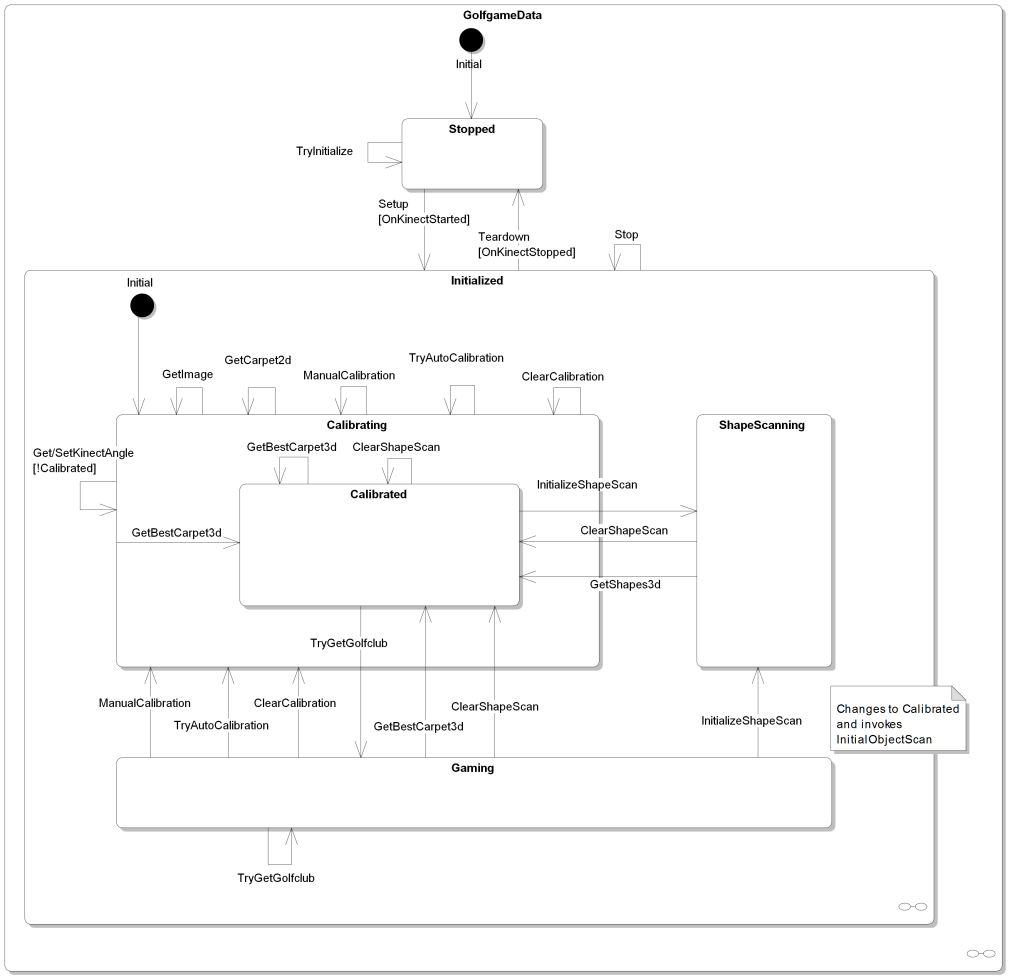

Abbildung 3.26: State Diagramm IGolfgameData

Auf einen unerlaubten Methodenaufruf wird mit einer `IllegalOperation` Exception reagiert. Werden neue Streams der Kinect eingeschaltet, kann es eine Weile dauern. Dies geschieht, wenn auf `Initialized`, auf `Gaming` und von `Gaming` weg gewechselt wird. Wird die Kinect während eines `Initialized`-Zustandes ausgesteckt, wird eine `KinectLost`-Exception geworfen und zu `Stopped` übergegangen.

3.3.4 `Main`

Das `Main`-Projekt enthält den Programmeinstiegspunkt und den Glue-Code, der die verschiedenen Projektkomponenten über Event-Listener miteinander verbindet.
Aufgaben die den Input involvieren und möglicherweise blockieren, wie z.B. die Ermittlung des Spielfelds, werden in separaten Worker-Threads abgearbeitet, um ein temporäres Einfrieren der Applikation zu verhindern.

Abbildung 3.27 zeigt eine Übersicht über die Architektur.

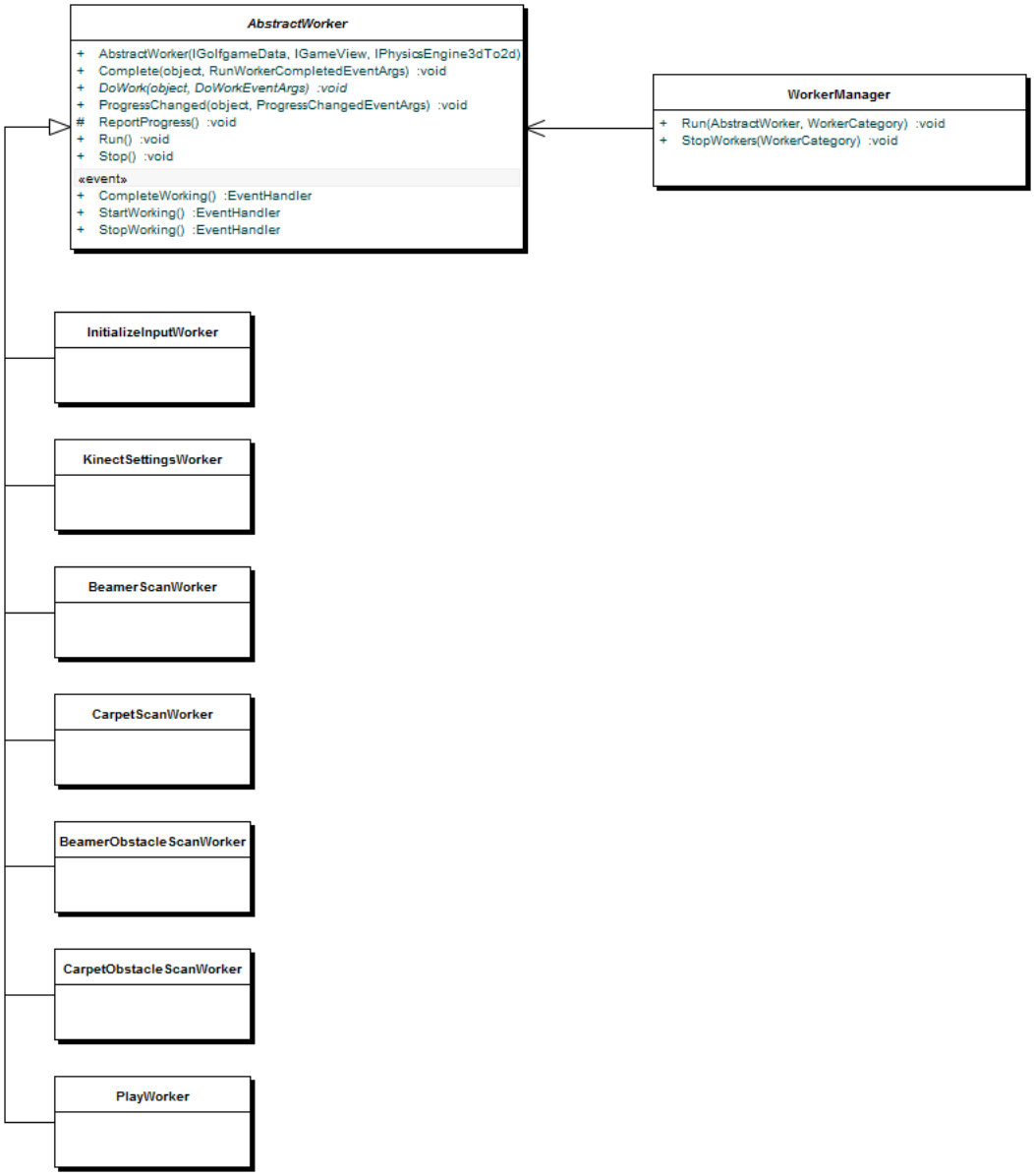

Abbildung 3.27: Übersicht der Worker-Klassen

Ein `AbstractWorker` stellt drei Methoden zur Auftragsabarbeitung zur Verfügung. `DoWork()` enthält den Code, der in einem separaten Thread ausgeführt wird. `ProgressChanged()` wird vom Thread ausgeführt, der den Worker beauftragt hat und kann z.B. für GUI-Updates benutzt werden. `Complete()` wird aufgerufen sobald die Verarbeitung von `DoWork()` terminiert ist. Auch diese Methode wird von dem Thread ausgeführt, der `Run()` auf dem Worker aufgerufen hat.
XNAs Einschränkung, dass Methoden innerhalb der XNA-View nur von dem Thread aufgerufen werden dürfen, der die XNA-Applikation gestartet hat, können dank der `ProgressChanged()`- und `Complete()`-Methode bequem umgangen werden.

3.3.4.1 Threads

Um die Aufgabenaufteilung der Applikation besser zu verstehen, sind im Folgenden alle Threads aufgeführt, die während Ablauf des Programms Code ausführen.

`MainThread` Er wird bei Programmstart erstellt und arbeitet den Konstruktor der `App`-Klasse ab. Dabei initialisiert er die `View` und registriert alle Listener darauf. Danach ist er nur noch für das Rendering des GUIs verantwortlich.

`WorkerThread` Der Code innerhalb der `DoWork`-Methode eines `AbstractWorkers` wird von einem separaten Thread ausgeführt. Die Klasse `PlayWorker`, zum Beispiel, holt in einer Schleife jeweils die aktuellste Schlägerposition vom Input, gibt sie an der Physics-Engine weiter und beauftragt den `MainThread`, den Schläger im GUI neu zu zeichnen.

`KinectThread` Der `Input` registriert sich intern für den `StatusChangedEvent`, um darüber informiert zu werden, wenn beispielweise die Kinect ein- oder ausgesteckt wurde und propagiert das Ereignis weiter an die oberen Schichten. Der registrierte Eventhandler wird in diesem Fall über einen von der Kinect erstellten Background-Thread aufgerufen.

3.3.5 `Adapter`

Wie aus der Architektur in 3.10 ersichtlich ist, besteht die Applikation aus den Hauptkomponenten `Input`, `PhysicsEngine`, `View` und dem Glue-Code (`Main`), der sie miteinander verknüpft.
Bei der Kommunikation zwischen den Komponenten stellt sich das Problem, dass sie in unterschiedlichen Einheiten und Koordinaten rechnen. Folgende Skizze veranschaulicht den Sachverhalt:

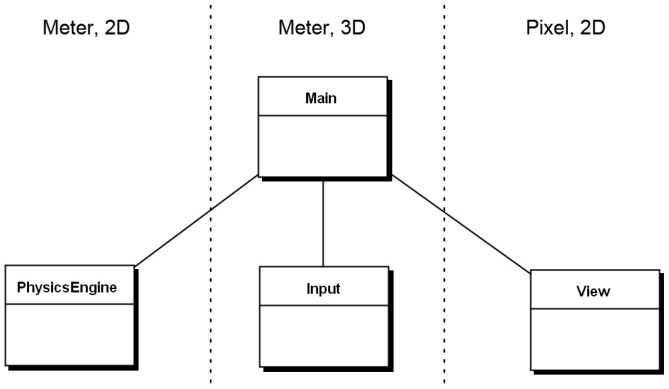

Abbildung 3.28: Schematische Darstellung der Komponenten mit ihren Einheiten

Um Umrechnungen zwischen den Komponenten zu vereinfachen, schalten wir der `PhysicsEngine` und den `View`-Komponenten einen Adapter vor. Dieser bietet gegen aussen dieselbe Schnittstelle an wie sein adaptiertes Objekt, passt aber in Tat und Wahrheit lediglich den Input an und leitet ihn an das adaptierte Objekt weiter. Dies befreit die Komponenten von Umrechnungen und macht das System flexibel, falls bestehende Konversionen geändert oder neue hinzugefügt werden müssen.

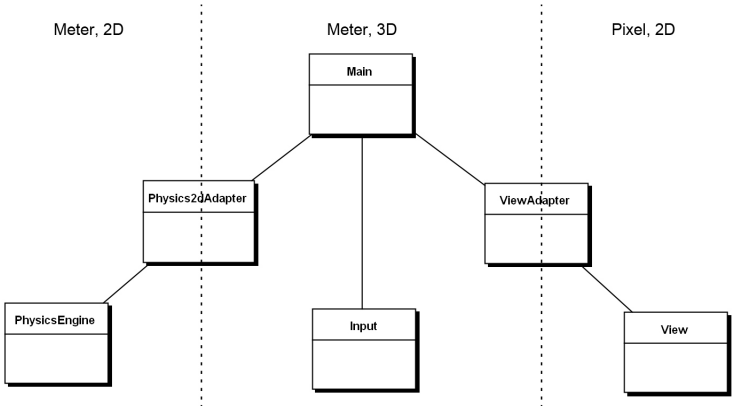

Abbildung 3.29: Schematische Darstellung der Konversion mit Adaptern

3.3.5.1 Vom `Input` zur `PhysicsEngine`

Der Hardware-Abstraktionslayer der Kinect liefert für das Spielfeld, die Hindernisse und Schlägerposition metrische 3D-Koordinaten aus Sicht der Kinect. Die `PhysicsEngine` arbeitet in einem zweidimensionalen MKS (Meter-Kilogramm-Sekunden)-System. Die Einheiten müssen bei der Weitergabe vom `Input` an die `PhysicsEngine` nicht konvertiert werden. Jedoch besteht ein anderes, weitaus komplexeres Problem: Die erhaltenen 3D-Koordinaten müssen auf eine 2D-Ebene projiziert werden.

Dazu wird eine Abbildung von einem 3D- auf einen 2D-Punkt berechnet:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \rightarrow \begin{pmatrix} x \\ y \end{pmatrix} \tag{3.1}$$

Beim Scannen des Spielfelds liefert der `Input` vier 3D-Punkte. Die Basis für diese Punktvektoren bildet das Koordinatensystem der Kinect. Im einem nächsten Schritt werden alle Punkte mit Hilfe einer Basistransformation in das Koordinatensystem des Spielfelds mit Ursprung im unteren linken Punkt A umgerechnet (3.3.3.4).
Die neuen Basisvektoren werden mit vektorgeometrischen Operationen ermittelt. Wie im Folgenden gezeigt wird, reichen dazu schon drei beliebige Punkte aus.

Als Ursprung der neuen Basis verwenden wir Punkt A. Die neuen Basisvektoren können bestimmt werden durch

$$\vec{b_1} = \frac{\vec{AB}}{|\vec{AB}|} = \frac{\vec{B} - \vec{A}}{|\vec{B} - \vec{A}|} \tag{3.2}$$

$$\vec{b_2} = \frac{\vec{AD}}{|\vec{AD}|} = \frac{\vec{D} - \vec{A}}{|\vec{D} - \vec{A}|} \tag{3.3}$$

Zur Vereinfachung werden sie gleich zu Beginn normalisiert.
Die Kinect erkennt Punkte zwar relativ genau, aber nie exakt. Für die beiden Basisvektoren $\vec{b_1}$ und $\vec{b_2}$ bedeutet dies, dass sie nicht orthogonal zueinander sind. Für das weitere Vorgehen ist es jedoch eine Bedingung, dass das Koordinatensystem kartesisch, also orthogonal ist.

Um dies zu erreichen kann der zweite Basisvektor $\vec{b_2}$ mit Hilfe des ersten Basisvektors $\vec{b_1}$ und einem Faktor λ so verschoben werden, dass deren Skalarprodukt 0 ergibt und sie somit orthogonal zueinander stehen.

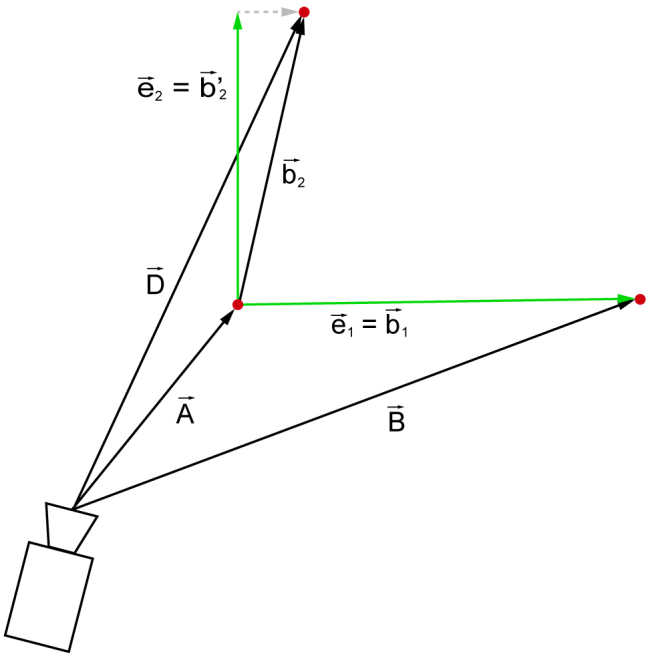

Abbildung 3.30: Koordinatentransformation

Für den gesuchten Vektor $\vec{b_2'}$ ergibt sich die Gleichung

$$\vec{b_2'} = \vec{b_2} + \lambda\vec{b_1} \tag{3.4}$$

Um λ zu ermitteln nutzen wir die Bedingung, dass das Skalarprodukt von $\vec{b_2'}$ und $\vec{b_1}$ 0 sein muss.

$$\begin{aligned} \vec{b_2'}\vec{b_1} &= 0 && (3.5)\\ (\vec{b_2} + \lambda\vec{b_1})\vec{b_1} &= 0 && (3.6)\\ \vec{b_2}\vec{b_1} + \lambda \cdot 1 &= 0 && (3.7)\\ \lambda &= -\vec{b_1}\vec{b_2} && (3.8) \end{aligned}$$

Die endgültigen Basisvektoren $\vec{e_1}$ und $\vec{e_2}$ ergeben sich aus $\vec{b_1}$ und $\vec{b_2'}$. Der dritte Basisvektor $\vec{e_3}$ kann rein mathematisch durch das Kreuzprodukt der anderen beiden Basisvektoren berechnet werden, obwohl dieser an sich nicht mehr benötigt wird, wie weiter unten erklärt ist.

$$\vec{e_1} = \vec{b_1}\ ,\ \vec{e_2} = \vec{b_2'}\ ,\ \vec{e_3} = \vec{e_1} \times \vec{e_2} \tag{3.9}$$

Um nun einen beliebigen, vom Sensor gelieferten Punkt $\vec{P_K}$ als Punkt $\vec{P}$ bezüglich der neuen Basis darzustellen, wird zuerst der Vektor $\vec{OP}$ vom neuen Koordinatenursprung O_K zum Punkt $\vec{P_K}$ und danach das Skalarprodukt mit jedem Vektor der neuen Basis berechnet.

$$\vec{OP} = \vec{P_K} - \vec{O_K} \tag{3.10}$$

$$P_x = \vec{OP} \cdot \vec{e_1} \tag{3.11}$$
$$P_y = \vec{OP} \cdot \vec{e_2} \tag{3.12}$$
$$P_z = \vec{OP} \cdot \vec{e_3} \tag{3.13}$$

Da das Spielfeld eine Ebene ist, kann nun die z-Komponente einfach weggelassen werden und die Projektion von 3D auf 2D ist fertig.

3.3.5.2 Von der `PhysicsEngine` zum Bildschirm

Nach jeder Simulation eines Zeitabschnitts in der `PhysicsEngine` werden alle Spielobjekte auf dem Bildschirm neu gezeichnet. Damit die Körper in der `PhysicsEngine` auf Bildschirmtexturen abgebildet werden können, finden zwei Umrechnungen statt: Zum einen müssen Meter auf Pixel abgebildet werden, zum anderen müssen die Koordinaten in Bildschirm-Koordinaten umgerechnet werden.

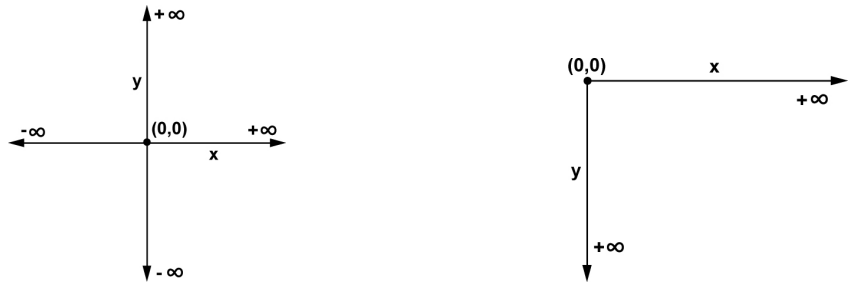

Abbildung 3.31: Vergleich der Koordinaten

Für die Koordinatentransformation wird der Ursprung des `PhysicsEngine`-Koordinatensystems in das Zentrum des Bildes (S_{cx}, S_{cy}) gelegt und die X- und Y-Komponente je mal einem Skalierungfaktor s addiert bzw. subtrahiert. Die Umrechnung eines Punktes aus der `PhysicsEngine` (P_x, P_y) zu einem Bildschirmpunkt (S_x, S_y) erfolgt mit

$$S_x = S_{cx} + P_x \cdot s \tag{3.14}$$
$$S_y = S_{cy} - P_y \cdot s \tag{3.15}$$

Der Faktor s für die Umrechnung von Meter auf Pixel kann leicht mit dem Verhältnis der Maximalgrösse des Spielfelds auf dem Bildschirm zu dem gescannten Spielfeld ermittelt werden.

3.3.5.3 Design

`ScreenAdapter` werden dort eingesetzt, wo Werte der `PhysicsEngine` an die `View` übergeben werden. Die Screen-Klassen `PlayScreen` und `CarpetScanScreen` und

`BeamerScanScreen` brauchen den Input der PhysicsEngine für ihre Anzeige. Deshalb wird diesen Klassen ein Adapter voran gestellt.
Die Umrechnung von `Phyiscs-Engine` zum Bildschirm und umgekehrt geschieht nicht im Adapter selbst, sondern im `PhysicsToScreenUnitConverter`.

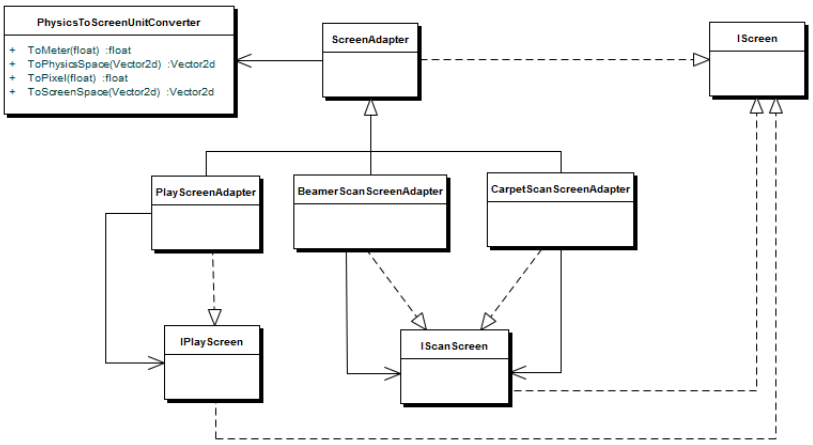

Abbildung 3.32: UML-Diagramm der ScreenAdapter-Architektur

Die View instanziert alle Screens über eine Factory. Dies ermöglicht es, die adaptierten Klassen `PlayScreen`, `CarpetScanScreen` und `BeamerScanScreen` mit einer überschriebenen `ScreenFactory`, der `AdaptedScreenFactory`, zu injizieren.

`PhysicsEngine3dTo2d` Die `PhysicsEngine3dTo2d`-Klasse dekoriert [Gamm1995] die `PhysicsEngine2d` mit all ihren Methoden und ergänzt sie, wie Abbildung 3.33 zeigt mit drei Methoden, die `Vector3d`-Objekte entgegennehmen. Sie bildet mit der Hilfsklasse `Plane3d` die 3D-Vektoren auf 2D-Vektoren gemäss 3.3.5.1 ab und leitet sie weiter an die `PhysicsEngine`.

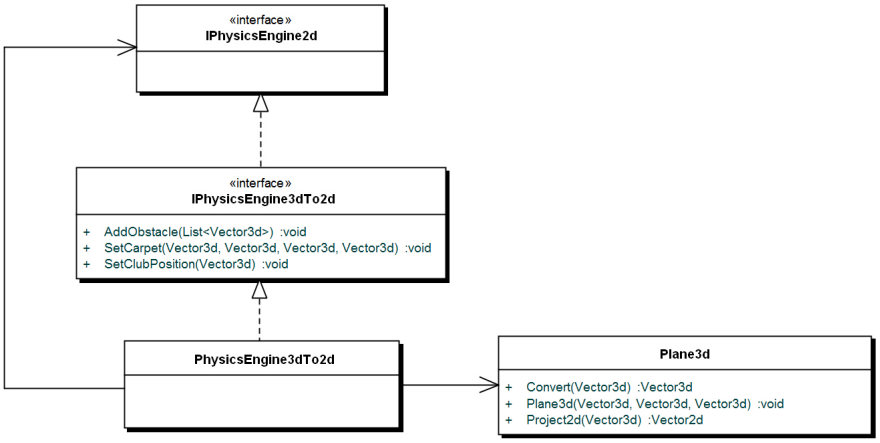

Abbildung 3.33: UML-Diagramm des `PhysicsEngine3dTo2d`-Decorator

3.3.6 `PhysicsEngine`

Die Simulation des Minigolfspiels wurde mit Hilfe der Farseer Physics Engine implementiert. Sie übernimmt die Kollisionsdetektion und berechnet anhand der Eigenschaften aufeinanderprallender Körper die resultierenden Kräfte.

Wie in Tab. 3.3 ersichtlich, wurden die Parameter möglichst realitätsnah gewählt. Die Grösse des Schläger und des Balls stimmt demnach ungefähr mit der Realität überein, wobei Kompromisse zu Gunsten des Spielspass eingegangen wurden.

Körper	**Typ**	**Grösse [m]**	**Dichte [kg·m^{-2}]**	**Haftreibungskoeff.**
Spielfeldrand	Statisch	individuell	-	0
Schläger	Dynamisch	0.03×0.1	10	0.2
Ball	Dynamisch	Radius: 0.02	33.4	0
Loch	Statisch	Radius: 0.05	-	0.2
Hindernisse	Statisch	individuell	-	0.2

Tabelle 3.3: Physikalische Eigenschaften des Minigolfmaterials

Vermerk zur Dichte in Tab. 3.3: In der Farseer Physics-Engine wird die Dichte im Zweidimensionalen mit der Fläche und nicht mit dem Volumen berechnet.

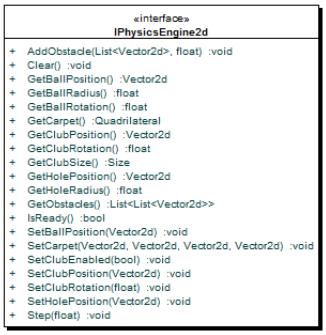

Abbildung 3.34: Interface der `PhysicsEngine`

Abbildung 3.34 zeigt eine Übersicht der angebotenen Funktionen, die von der Phyiscs-Engine implementiert werden.

3.3.6.1 Anschlagsberechnung

Über `SetClubPosition(Vector2d position)` und `SetClubRotation(float angle)` kann von aussen die Position und Ausrichtung des Schlägers geändert werden. In einem Buffer werden die letzten x Positionswerte abgelegt, um bei einer Kollision von Schläger und Ball die Anschlagsgeschwindigkeit zu bestimmen.
Die Geschwindigkeit wird mit folgender Formel berechnet:

$$\vec{v} = \frac{|\Delta\vec{s}|}{\Delta t} \cdot \vec{r} \tag{3.16}$$

Dabei ist $\Delta\vec{s}$ die Differenz zwischen dem aktuellsten und ältesten Positionsvektor im Buffer. Δt ist die verstrichene Zeit zwischen den beiden Positionen. $\vec{r}$ ist der Richtungsvektor des Schlägers, der aus dem aktuellen Winkel des Schlägers gebildet wird.
Damit der Ball mit beiden Seiten des Schlägers angeschlagen werden kann, wird die Geschwindigkeit mit der Seite des Kontakts multipliziert. 1 entspricht der rechten, -1 der linken Seite.
Farseer Physics bietet die Möglichkeit, Impulse auf Körper anzuwenden. Dies wird nun dazu benutzt, den Aufschlag des Schlägers auf den Ball zu übertragen. Für die Berechnung des Impulses wird die ermittelte Geschwindigkeit mit der Masse des Schlägers multipliziert.

$$\vec{p} = m \cdot \vec{v} \tag{3.17}$$

3.3.6.2 Simulation der dritten Dimension

Bei diesem Spiel ist das Spielfeld eine flache Ebene und die Physik wird nur im Zweidimensionalen simuliert. Dadurch ist es beispielsweise nicht möglich, eine Reibung zwischen Ball und Spielebene zu definieren oder ein Loch einzubauen, worin der Ball versenkt werden kann.
Um das Verhalten des Balls trotzdem möglichst realistisch zu gestalten, setzen wir anstatt einer Reibung eine lineare und angulare Dämpfung ein. Da ab einer gewissen minimalen Geschwindigkeit der Widerstand des Bodens zu gross wird, als dass sich der Ball weiter fortbewegen könnte, wird die Geschwindigkeit ab diesem Schwellwert auf 0 gesetzt.

Um die Spannung beim Einlochen etwas zu erhöhen, wird, um die Tiefe des Loches zu simulieren, ein Trick angewendet: Ein Gravitationspunkt, der in der geometrischen Mitte des Lochs liegt. Dieser übt eine Kraft auf den Ball aus, solange er sich innerhalb der Lochtextur befindet. Die Kraft nimmt umgekehrt proportional zur Distanz zwischen Ball und Loch zu. D.h. je näher sich der Ball beim Mittelpunkt des Lochs befindet, desto stärker wird er davon angezogen.
Diese Technik hat einen schönen Nebeneffekt: Wenn der Ball am Lochrand entlang rollt, wird er durch die Anziehung so abgelenkt, dass er für einen Moment dem radialen Verlauf des Lochrands folgt und bei genug grosser Geschwindigkeit den Rand wieder verlässt, anstatt ins Loch zu rollen.

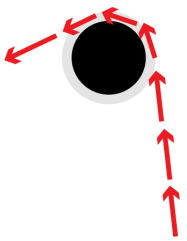

Abbildung 3.35: Möglicher Ballverlauf am Lochrand

3.3.7 `View`

3.3.7.1 XNA

Im Gegensatz zu anderen Grafik-Frameworks wie WPF hat XNA keine nativen GUI-Komponenten wie Buttons, Labels oder Textfelder. Solche Elemente müssen selber erstellt

werden. Nachteilig ist auch, das keine hierarchische Anzeige mögliche ist, die eine relative Positionierung oder vererbte Transparenz erlaubt, wie dies beispielsweise in HTML oder Flash der Fall ist. In der Zeichnungsroutine ist einzig die Textur, absolute Position und die Renderingreihenfolge bestimmbar.

`DisplayObject` Das GUI ist ein wichtiger Bestandteil des Minigolfspiels. Darüber werden diverse Parameter des `Input`-Packages konfiguriert. Neben dem Spiel enthält es selbst noch weitere Benutzerdialoge. Um deren Entwicklung zu vereinfachen und zu beschleunigen, wurde eine `DisplayObject`-Klasse entwickelt, die dem Displaylist-Konzept von Flash nachempfunden ist und die eingangs erwähnten Probleme adressiert.

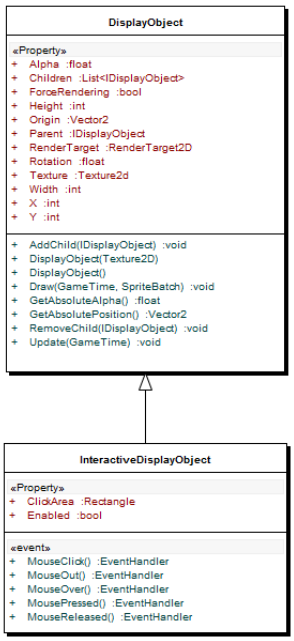

Abbildung 3.36: DisplayObject Hierarchie

Ein `DisplayObject` hat eine Textur[12], Position, Grösse, Alphawert[13] usw. und ist selbst Container für weitere `DisplayObject`-Klassen (Composite Pattern [Gamm1995]). Sie können beliebig tief verschachtelt werden, wobei sich Eigenschaften wie Position und Alpha gemäss ihrer Hierarchie aufaddieren bzw. multiplizieren. Damit lässt sich beispielsweise ein GUI-Element, das aus mehreren Subkomponenten besteht, verschieben, indem einfach die Position der äusseren Komponente geändert wird.

Mit Hilfe der `DisplayObject`-Klasse hat sich die GUI-Entwicklung erheblich vereinfacht. Für das Minigolfspiel wurde die in Tab. 3.4 aufgeführten User-Interface-Komponenten entwickelt.

[12] Grafische Repräsentation
[13] Transparenzwert

Komponente	Aufgabe
`InteractiveDisplayObject`	Basis für Komponenten die Mouse-Events verarbeiten
`Label`	Anzeige von Text
`RadioButton`	Auswahl einer Option
`Slider`	Auswahl eine Wertes in einem bestimmten Bereich

Tabelle 3.4: Eigene User-Interface-Komponenten

`InteractiveDisplayObject` Um mit `DisplayObject`-Klassen interagieren zu können, musste auch die Logik für Events wie `MouseClick`, `MouseOver`, `MouseOut` etc. selber implementiert werden. Ein `InteractiveDisplayObject` kapselt diese Funktionalität. Damit ist es einfach, interaktive GUI-Komponenten wie Buttons oder Slider zu erstellen.

3.3.7.2 Screens

Unter einem `Screen` verstehen wir die grafische Abbildung eines bestimmten Game-Zustands (z.B. Hauptmenü, Einstellungen, Spiel). Um den Wechsel zwischen den verschiedenen Screens zu vereinheitlichen, wurde ein `AbstractScreen` erstellt, der als Basis für alle Screens fungiert. Er bietet nebem den Methoden `Show()` und `Hide()` auch Events für die Ereignisse `AfterShow`, `BeforeShow`, `BeforeHide`, `AfterHide`, die dazu benutzt werden können, Aktionen wie das Polling der Golfschlägerposition im `Input` auszulösen.

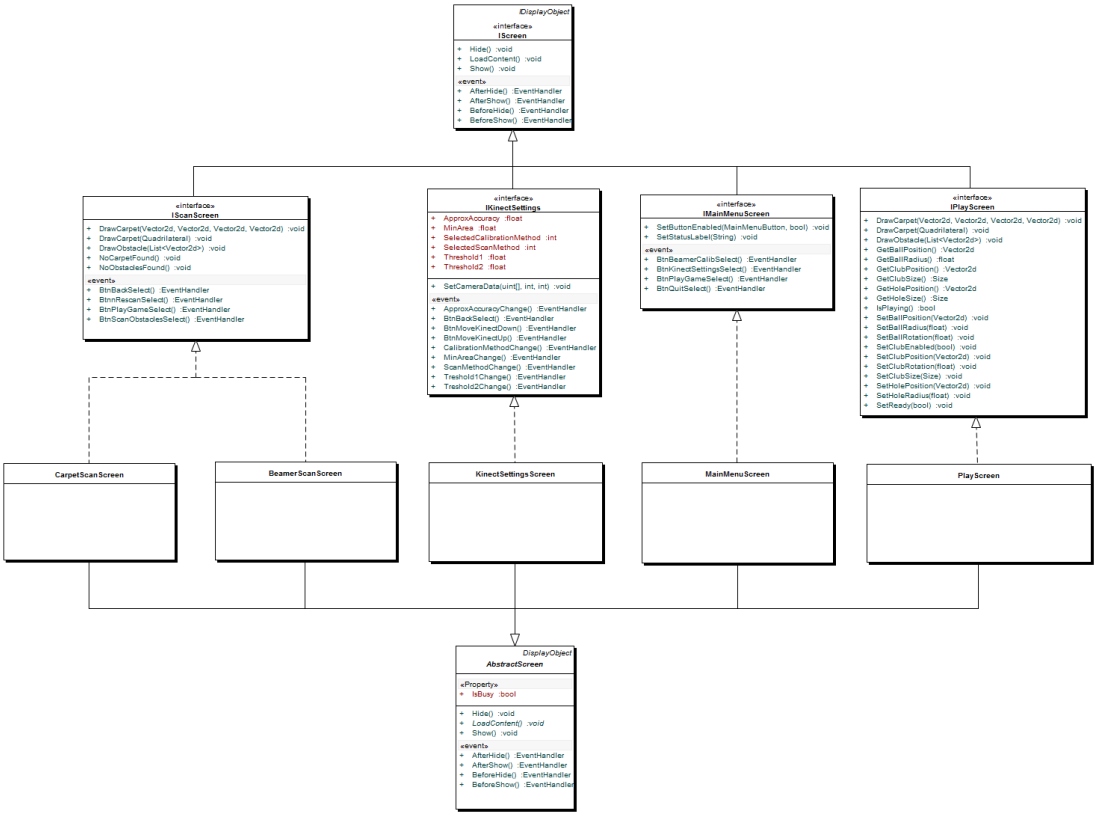

Abbildung 3.37: Verwendete Screen-Klassen mit Interfaces

Das Spiel besteht aus insgesamt fünf Screens, die im Folgenden erläutert werden.

`MainMenuScreen` Hauptmenü und Einstiegspunkt der Applikation.

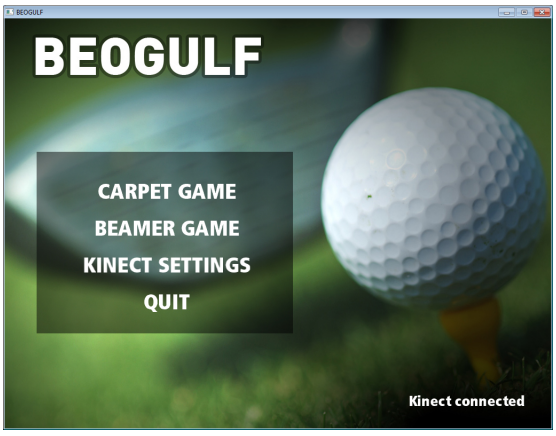

Abbildung 3.38: Screenshot des Hauptmenüs

`BeamerScanScreen` Der `BeamerScanScreen` setzt voraus, dass ein Beamer ein rechteckiges Spielfeld auf den Boden projiziert. Er zeigt eine weisse Fläche an, die dann vom `Input`-Package mithilfe der Kinect als Spielfeld erkannt wird. Danach können beliebige Hindernisse in das Spielfeld gelegt werden, die nach einem weiteren Scan angezeigt werden.

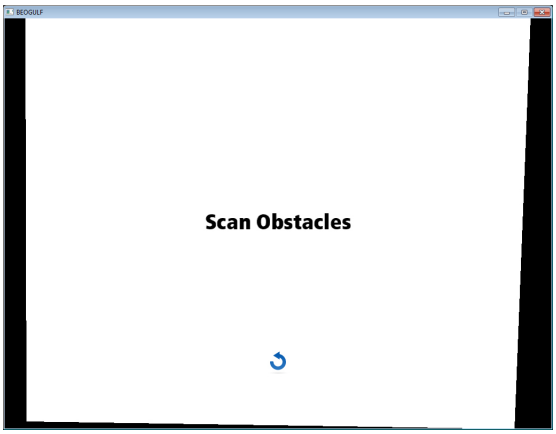

Abbildung 3.39: Screenshot des `BeamerScanScreen`

`CarpetScanScreen` Das Minigolfspiel kann auch ohne ein von oben projiziertes Spielfeld gespielt werden. Voraussetzung dafür ist einzig, dass anstatt des projizierten Spielfelds ein Teppich oder ähnliches auf den Boden gelegt wird, das einen starken Kontrast zum Boden bildet.
Im `CarpetScanScreen` wird dann das gescannte Spielfeld angezeigt. Danach können analog zum `BeamerScanScreen` Hindernisse in das Spielfeld gelegt werden, die nach einem weiteren Scan angezeigt werden.

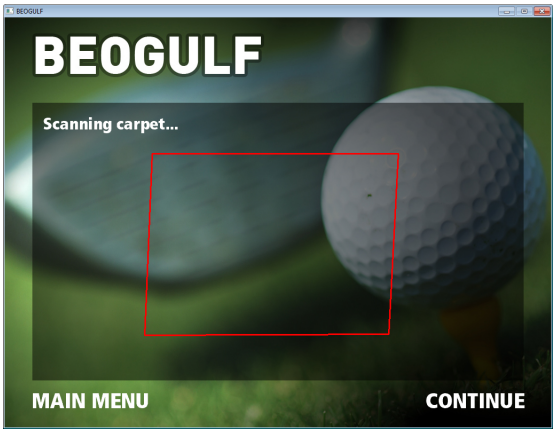

Abbildung 3.40: Screenshot des `CarpetScanScreen`

`KinectSettingsScreen` Die verschiedenen Parameter der Kinect können über den `KinectSettingsScreen` konfiguriert werden. Er zeigt zusätzlich das Kamerabild der Ki-

nect und ermöglicht ihre korrekte Ausrichtung über die Pfeil-Buttons, sodass das Spielfeld und der Spieler gut sichtbar sind.

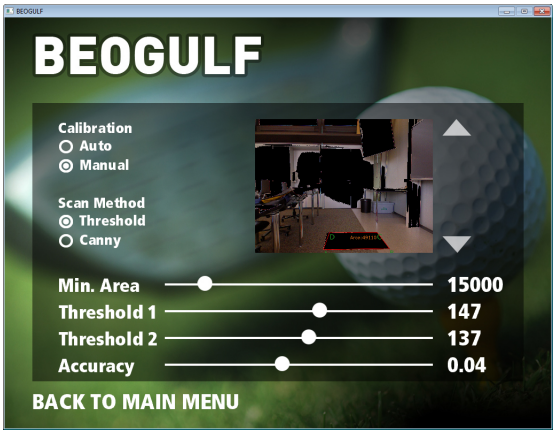

Abbildung 3.41: Screenshot des `KinectSettingsScreen`

PlayScreen Im `PlayScreen` läuft das eigentliche Spiel ab. Er zeigt das gescannte Spielfeld mit den Hindernissen, den Ball, das Loch und der Schläger des Users. Rechts oben signalisiert eine Ampel, wann geschlagen werden darf.

Abbildung 3.42: Screenshot des `PlayScreen`

3.3.7.3 Animationen

Um Animationen mit XNA zu realisieren sind die zwei Hook-Methoden der mitgelieferten Klasse `DrawableGameComponent` zentral: `Draw()` und `Update()`. Beide werden vom Framework mit 60 Hz aufgerufen.
Animationen in XNA werden üblicherweise realisiert, indem die Update-Methode überschrieben wird, um den Wert einer bestimmten Eigenschaft wie z.B. die Position einer Textur neu zu setzen und danach in der überschriebenen Draw-Methode die Textur neu zu zeichnen.

Bei Computerspielen werden die Position und auch andere Eigenschaften einer Textur üblicherweise durch den Userinput oder Output einer `Physics-Engine` bestimmt. Die Werte werden also während der gesamten Spieldauer in einem bestimmten Interval fortlaufend neu gesetzt. Solche Animationen zeichnen sich dadurch aus, dass sie zeitlich nicht begrenzt sind.
Neben dieser Art von Animation müssen aber oft auch GUI-Komponenten wie Menüs, Buttons etc. für eine bestimmte Zeit animiert werden. Beispielsweise wechselt ein Button innerhalb einer definierten Zeit seine Farbe oder auch der Wechsel vom `MainMenuScreen` zu einem anderen Screen hat in der Regel eine fixe Dauer.
Um die Erstellung solcher Animationen zu vereinfachen, wurde eine kleine Animations-Library entwickelt, die es ermöglicht, mittels Reflection beliebige nummerische Eigenschaften eines Objekts zu verändern. Anstatt die Animation in der `Update`- bzw. `Draw`-Methode zu definieren, kann über den `TweenManager`[14] ein beliebiges `DisplayObject` animiert werden. Um beispielsweise ein `DisplayObject` innerhalb von 600 ms von Position $(0, 0)$ nach $(200, 0)$ zu verschieben, reicht folgender Code:

```
TweenManager.Start( new PropertyTween(displayObject, "X", 600, 200));
```

TweenManager Der `TweenManager` ist dafür verantwortlich, jeden aktiven Tween mittels der `Update()`-Methode zu aktualisieren. Überdies erkennt er Konflikte, wenn beispielsweise zeitgleich die selbe Eigenschaft des selben Objekte animiert werden soll und spart so Resourcen.
Sobald ein Tween zu Ende ist oder gestoppt wurde, wird er aus dem Update-Zyklus entfernt und nicht mehr aktualisiert.

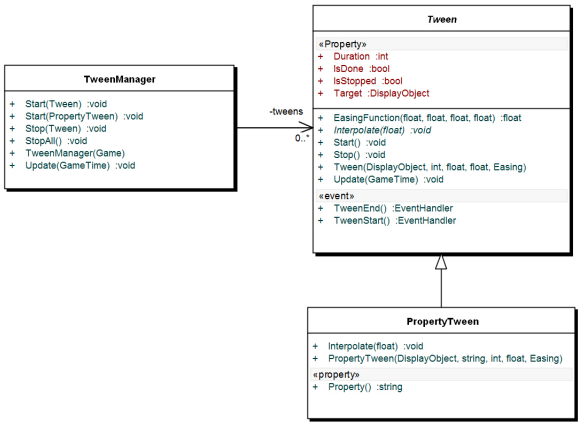

Abbildung 3.43: Architektur der Tweens

3.4 Ergebnisse

Das Ziel dieser Arbeit war es, ein Augmented Virtuality Minigolfspiel zu entwickeln. Das Spielfeld, der Ball, der Schläger und das Loch werden per Beamer auf den Boden projiziert.

[14] Tweening ist ein Verfahren, bei dem Einzelbilder zwischen zwei Schlüsselbildern einer Animation erzeugt werden, um den Eindruck einer flüssigen Veränderung zu erwecken. [Wiki2012b]

Auf das Spielfeld können reale Hindernisse gelegt werden, an denen der virtuelle Ball abprallt (siehe 1.1).

Im Rahmen dieser Bachelorarbeit wurde gezeigt, wie das Spielfeld und alle auf ihm liegenden, flachen Objekte mit der Kinect automatisch erkannt werden können. Der in der Kinect integrierte Tiefensensor und die Kamera machen es in Kombination mit vektorgeometrischer Operationen und Differenzbildern möglich, eine zweidimensionale Repräsentation des Spielfelds zu berechnen und die Koordinaten an eine Physics-Engine weiterzuleiten. Diese bildet das Spielfeld und die Hindernisse intern realitätsgetreu ab. Aufgrund von Messungenauigkeiten der Kinect kann es jedoch je nach Setup des Spiels zu Ungenauigkeiten bei der Umrechnung kommen. Tests ergaben, dass sich die Kinect leicht durch diverse Störfaktoren beeinflussen lässt. Als Hindernisse eignen sich besonders flache, polygonale Objekte, da diese bei zu grosser Störung einfach zu approximieren sind.
Die Implementation setzt voraus, dass der Beamer oberhalb des Spielfelds befestigt wird und senkrecht auf den Boden projiziert, so dass keine Keystone-Effekte auftreten.

Der Spieler kann das Minigolfspiel mit seiner Körperhaltung steuern. Das System erfasst dafür verschiedene Gelenkpunkte, um die Position des virtuellen Golfschlägers zu berechnen. Der Anschlagswinkel wird durch die Beinstellung des Spielers ermittelt. Um Ungenauigkeiten bei der Schlägerberechnung zu korrigieren, wurde die Methode der kleinsten Quadrate angewendet. Sie nähert eine Ausgleichsgerade durch die verschiedenen Gelenkpunkte an und bestimmt so die Position des virtuellen Schlägers.

Für das Minigolfspiel wurde ein grafisches User-Interface mit XNA entwickelt. Es bietet neben der Anzeige des eigentlichen Spiels die Möglichkeit, verschiedene Parameter der Kinect zu konfigurieren. Dabei sind einige wiederverwendbare Klassen entstanden, die die Erstellung von Menüs und Animationen in XNA vereinfachen.

3.5 Schlussfolgerung

Die beschriebenen Ergebnisse und der entwickelte Prototyp zeigen, dass ein Minigolfspiel fürs Wohnzimmer mit einer Kinect realisiert werden kann. Für Produktreife müsste jedoch zusätzlich eine automatische Keystone-Korrektur der Beamerprojektion umgesetzt und die Hinderniserkennung optimiert werden. Daneben gäbe es ein Reihe weiterer Funktionen, die eingebaut werden könnten. Einige Vorschläge sind nachfolgend aufgeführt.

- Anzahl Schläge zählen → Highscore
- Mehrspielermodus
- Speichern von Spielständen und Spielfeldern
- Spielfeldeditor
- Böden mit verschiedenen physikalischen Eigenschaften (z.B. Wiese, Beton, Eis)
- Unebenheit: nach Stoppen des Balles, bewegt er sich manchmal nochmals kurz in eine zufällige Richtung
- Soundeffekte
- Dynamisches Loch: das Loch wird grösser und kleiner oder verschiebt sich kontinuierlich

- Virtuelle Hindernisse, die sich bewegen
- Physikalische Eigenschaften der Hindernisse wie Anziehungskraft, Abstosseigenschaft und Elastizität
- Beschleunigungs- und Verlangsamungsbereiche wie Sandgrube, Wasser und Ölflecken
- Schläger- und Ballwahl mit verschiedenen Eigenschaften
- Ein Kontroller, der den Schläger simuliert und bei Ballanschlag eine Vibration auslöst

Kapitel 4

Anhang

4.1 Skeleton Analyse

Wie Tab. 3.2 zeigt, erlaubt die Kinect eine Konfiguration der Gelenkserkennung, z.B. um das Zittern zu minimieren. Um die Parametrisierung besser zu verstehen und für unsere Bedürfnisse anzupassen, wird im Folgenden die Stabilität der für uns relevanten Gelenkpunkte und die Unterschiede zueinander untersucht.

Durchschnittlich werden alle 33 ms neue Daten geliefert. Um die Stabilität der einzelnen Gelenkpunkte zu messen, wird für 5000 Messpunkte der Abstand zwei aufeinanderfolgender Gelenkpunkte berechnet und in einem Histogramm dargestellt. Der Abstand zur Kinect beträgt ca. 3 m.

Es wurden folgende Parameter gewählt, um die automatische Positionskorrektur der Kinect auszuschalten (Tab. 4.1).

Parameter	Wert
`Correction`	`0.5`
`Prediction`	`0.5`
`Smoothing`	`0.5`
`JitterRadius`	`0 m`
`MaxDeviationRadius`	`0.001 m`

Tabelle 4.1: `TransformSmoothParameters` ohne Korrektur

Die Messung bezieht sich auf typische Bewegungen die beim Golfspielen ausgeführt werden. Abstände sind in cm auf der `X-Achse` angegeben, die Anzahl Vorkommnisse auf der `Y-Achse`.

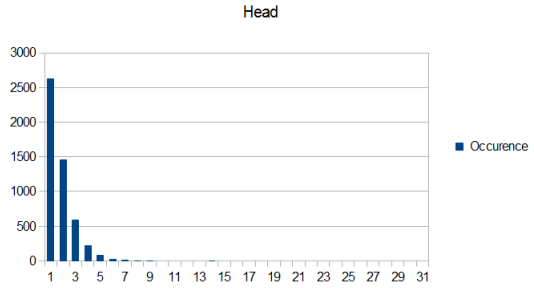

Abbildung 4.1: Kopf

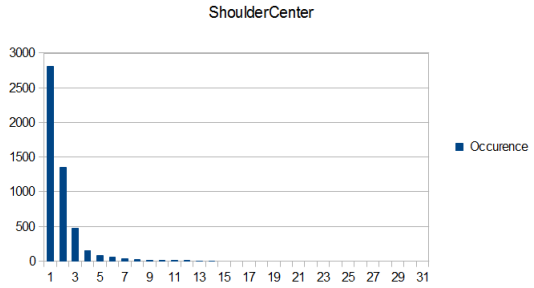

Abbildung 4.2: Schultermitte

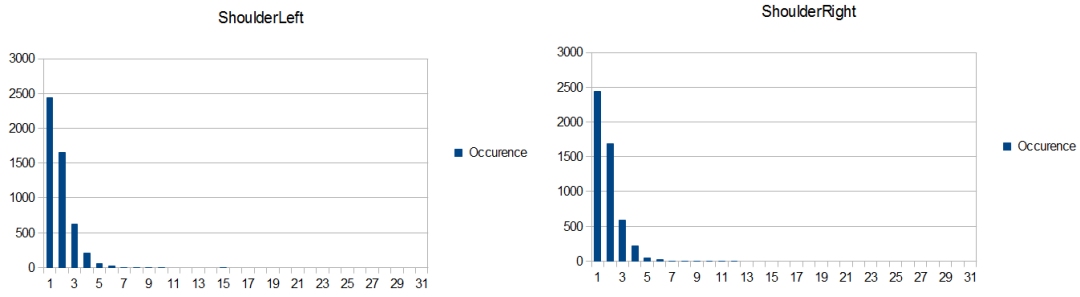

Abbildung 4.3: Schultern

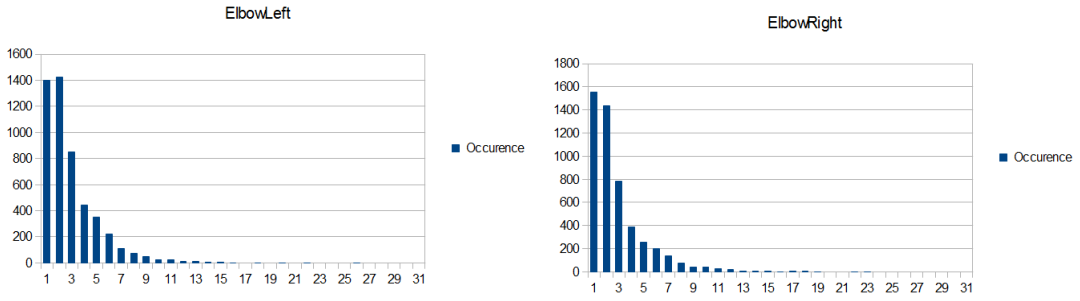

Abbildung 4.4: Ellbogen

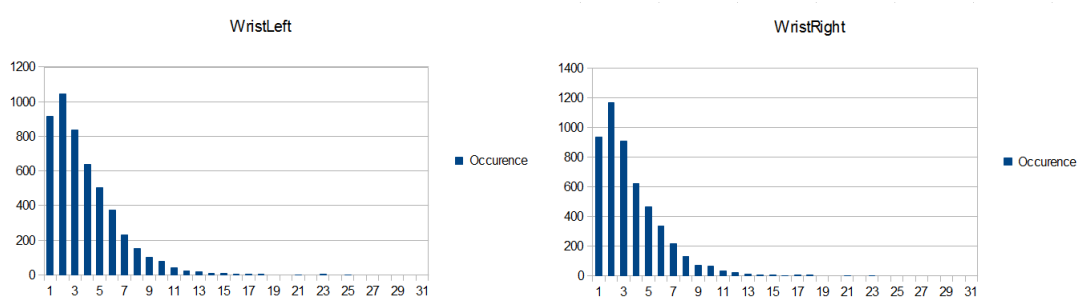

Abbildung 4.5: Handgelenke

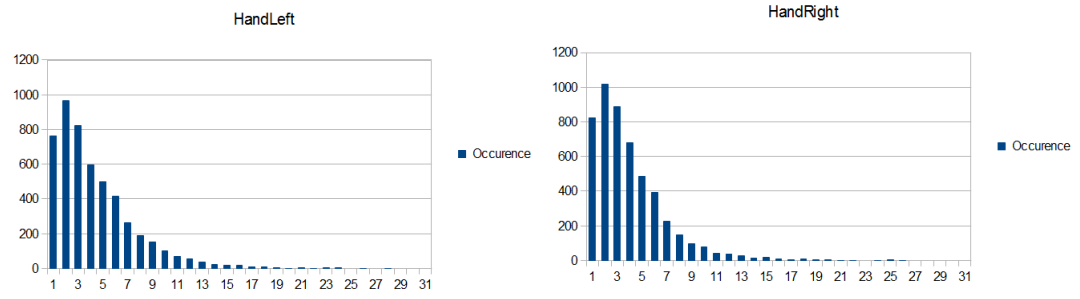

Abbildung 4.6: Hände

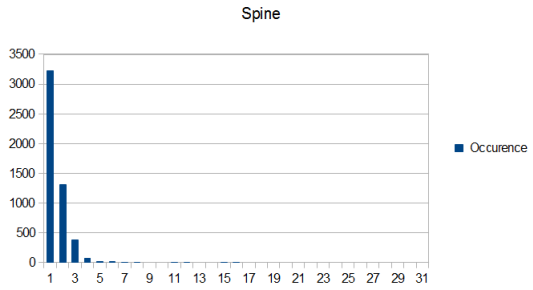

Abbildung 4.7: Rückgrat

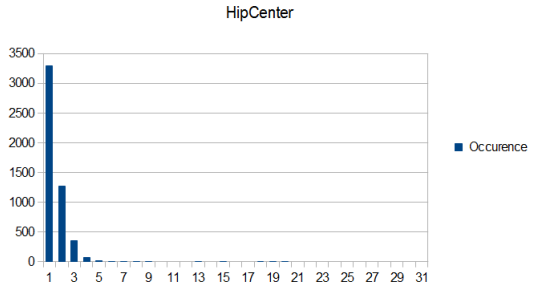

Abbildung 4.8: Hüftmitte

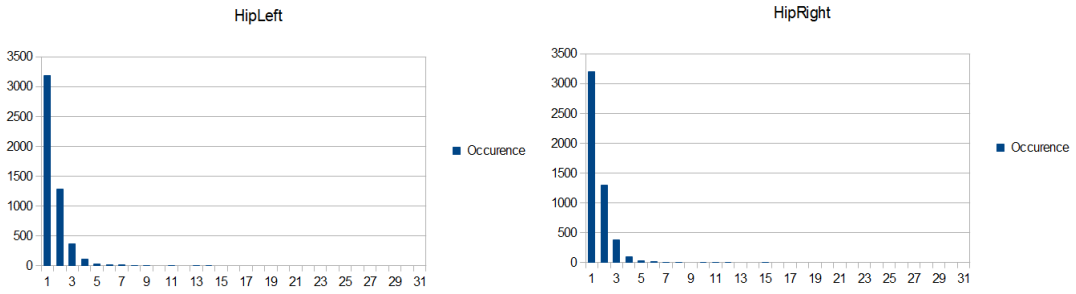

Abbildung 4.9: Hüften

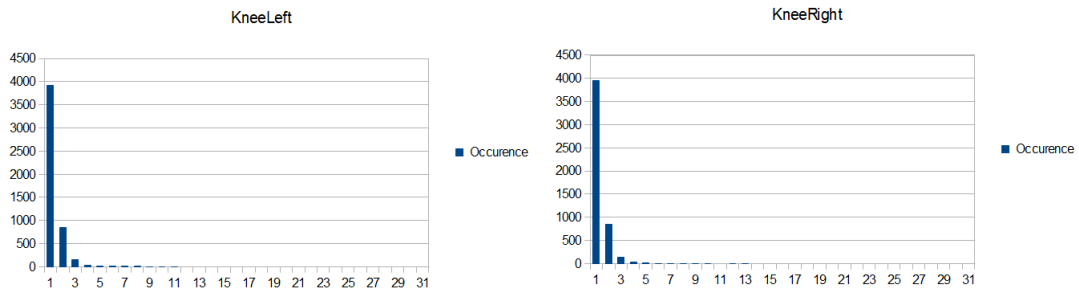

Abbildung 4.10: Knie

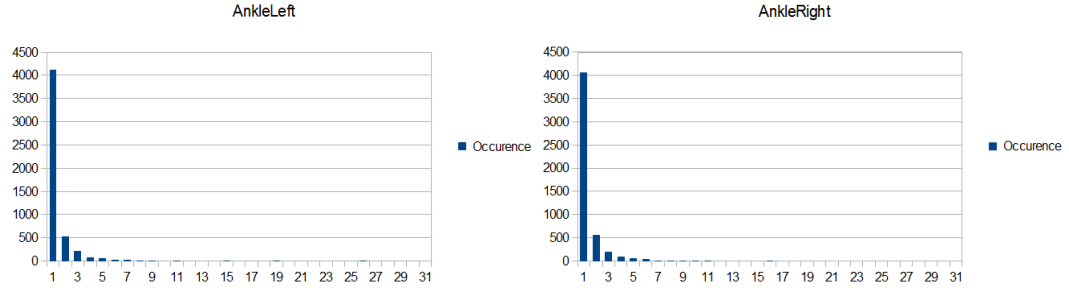

Abbildung 4.11: Fussgelenke

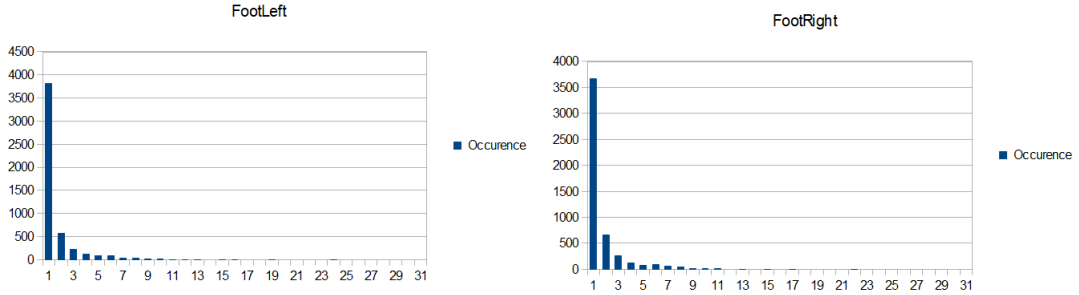

Abbildung 4.12: Füsse

Die Messungen wurden mithilfe der Klasse `SkeletonAnalysis` durchgeführt.

Auswertung Laut [Khos2011] liegt der Fehler von Tiefendaten bei einem Abstand von 2 - 4 m zwischen 5 - 20 mm. Durch das Skeletontracking vergrössert sich der Fehler, wie die obigen, zum Teil sehr wackeligen, Gelenke zeigen. Vor allem die Handgelenke und die Hände weisen hohe Abweichungen von über 5 cm auf. Dies ist insofern problematisch, als dass die Hände sehr wichtig für die Bestimmung des Golfschlägers sind.

Da die Handgelenke im Vergleich zu den Händen etwas stabiler sind, werden diese für die Berechnung der Schlägerposition verwendet.
Weiter fällt auf, dass die Messwerte der rechten Körperhälfte (Hand, Handgelenk, Ellbogen) etwas genauer sind als die links. Dies hat vermutlich damit zu tun, dass die eine Hälfte etwas näher zur Kinect stand als die andere, was typisch ist, wenn für ein Schlag ausgeholt wird.

Nach der Analyse und Tests mit verschiedenen stabilen Punkten wurde entschieden, dass sich für die Positionsbestimmung Schultern, Ellbogen und Handgelenke am besten eignen werden. Diese gestalten die Schlägerführung am intuitivsten.

Parameterbestimmung Für die Bestimmung der `TransformSmoothParameters` wurden zuerst folgende Standardwerte der Kinect benutzt:

Parameter	Wert
`Correction`	`0.5`
`Prediction`	`0.5`
`Smoothing`	`0.5`
`JitterRadius`	`0.04 m`
`MaxDeviationRadius`	`0.05 m`

Tabelle 4.2: `TransformSmoothParameters` Standardeinstellungen

Mit dieser Einstellung war auch nach der Schlägerpositionsberechnung (4.2) ein Zittern sichtbar. Deshalb wurde versucht, den `JitterRadius`, `MaxDeviationRadius` und `Smoothing`-Wert Schritt für Schritt herunterzusetzen, was schliesslich zu den in Tab. 4.3 aufgeführten, als gut empfundenen Werten führte:

Parameter	Wert
`Correction`	`0.5`
`Prediction`	`0.5`
`Smoothing`	`0.2`
`JitterRadius`	`0.025 m`
`MaxDeviationRadius`	`0.015 m`

Tabelle 4.3: `TransformSmoothParameters` Endparametrisierung

4.2 Methode der kleinsten Quadrate

Um eine Linie durch die Gelenkpunkte zu legen, die am besten passt, eignet sich die „Methode der kleinsten Quadrate“. Eine derartige Linie wird auch Ausgleichsgerade genannt. Es wird folgende lineare Modellfunktion gewählt:

$$f(x) = A \cdot x + B \tag{4.1}$$

Der Fehler eines Messwertes gegenüber einer Geraden kann wie folgt berechnet werden, wobei x und y für die Koordinaten des Messwertes stehen:

$$r_1 = A \cdot x_1 + B - y_1 \tag{4.2}$$

Für n Fehler

$$r_1 = A \cdot x_1 + B - y_1 \tag{4.3}$$
$$r_2 = A \cdot x_2 + B - y_2 \tag{4.4}$$
$$r_3 = A \cdot x_3 + B - y_3 \tag{4.5}$$
$$... \tag{4.6}$$
$$r_n = A \cdot x_n + B - y_n \tag{4.7}$$

Gesucht sind nun die Koeffizienten A und B der Ausgleichsgeraden mit der kleinsten Summe der Fehlerquadrate.

$$\min_{A,B} \sum_{i=1}^{n} r^2 \tag{4.8}$$

Die Summenfunktion als Funktion der beiden Variablen A und B, wird partiell nach diesen abgeleitet. Da es sich um die Summe von Quadraten handelt, resultieren daraus zwei lineare Ausdrücke. Die Nullstellen der Ableitungen bilden in diesem Fall je das Fehlerminimum. Durch Setzen der Nullstelle kann so nach der Variable aufgelöst werden, woraus folgende Gleichungen entstehen:

$$A = \frac{\left(\sum_{i=1}^{n} x_i y_i\right) - n \cdot \bar{x}\bar{y}}{\left(\sum_{i=1}^{n} x_i^2\right) - n \cdot (\bar{x})^2} \tag{4.9}$$

und

$$B = \bar{y} - A\bar{x} \tag{4.10}$$

Wobei hier $\bar{x}$ und $\bar{y}$ jeweils das arithmetische Mittel aller x bzw. y Werte bilden. [Wiki2012]

4.2.1 Golfschläger

Für die Berechnung des Golfschlägers wird eine Funktion benötigt, die durch Angabe des y-Wertes die x- und z-Position auf der Ausgleichsgeraden zurück gibt.
Dafür werden mit der oben beschriebenen Methode der kleinsten Quadrate die Variablen A_{yx} und B_{yx} für alle $\begin{pmatrix} y \\ x \end{pmatrix}$ berechnet. Dasselbe für die Variablen A_{yz} und B_{yz} für alle $\begin{pmatrix} y \\ z \end{pmatrix}$.

Nun kann mittels

$$f(y) \mapsto \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} A_{yx} \cdot x + B_{yx} \\ y \\ A_{yz} \cdot z + B_{yz} \end{pmatrix} \tag{4.11}$$

die Position des virtuellen Golfschlägers ermittelt werden.

4.3 Benutzerhandbuch

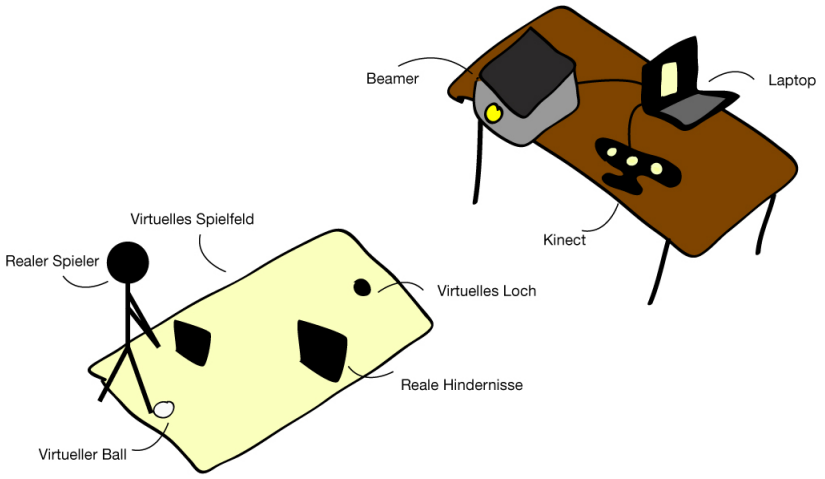

Abbildung 4.13: Schematische Darstellung des Spielaufbaus

Beim Spielaufbau sind folgende Punkte zu beachten:

1. Ein Beamer mit integrierter Keystone-Korrektur wird ca. 1 m ab Boden platziert. Alternativ muss ein Beamer ohne integrierte Keystone-Korrektur von oben senkrecht nach unten auf den Boden projizieren.
2. Die Kinect wird mindestens 1 m über dem Boden platziert
3. Das Spielfeld darf nicht mehr als 4 m von der Kinect entfernt sein
4. Der Boden darf nicht reflektieren
5. Der Raum sollte bei Möglichkeit leicht abgedunkelt werden
6. Der Winkel der Kinect muss so eingestellt sein, dass die Kamera das Bild des Spielfelds und des Spielers komplett einfangen kann
7. Der Spieler kann das Spiel nur gemäss den in Abb. 4.15 abgebildeten Körperhaltungen steuern

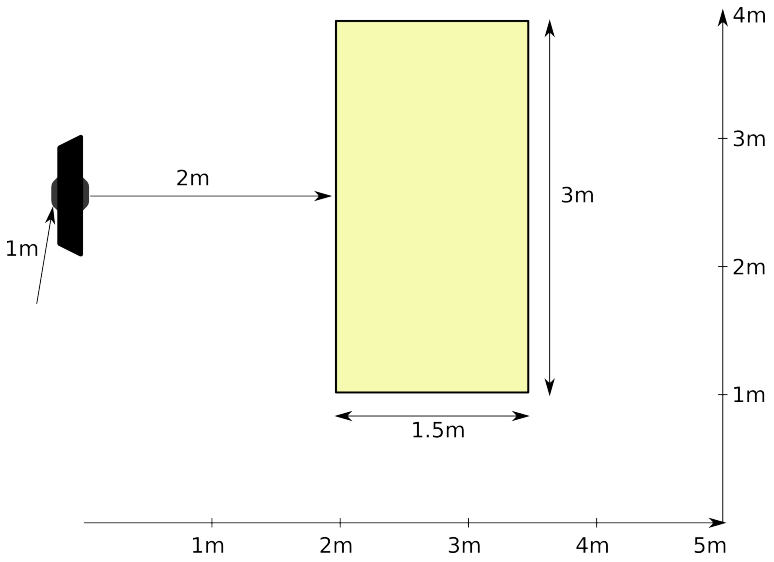

Abbildung 4.14: Optimale Spielbedingung

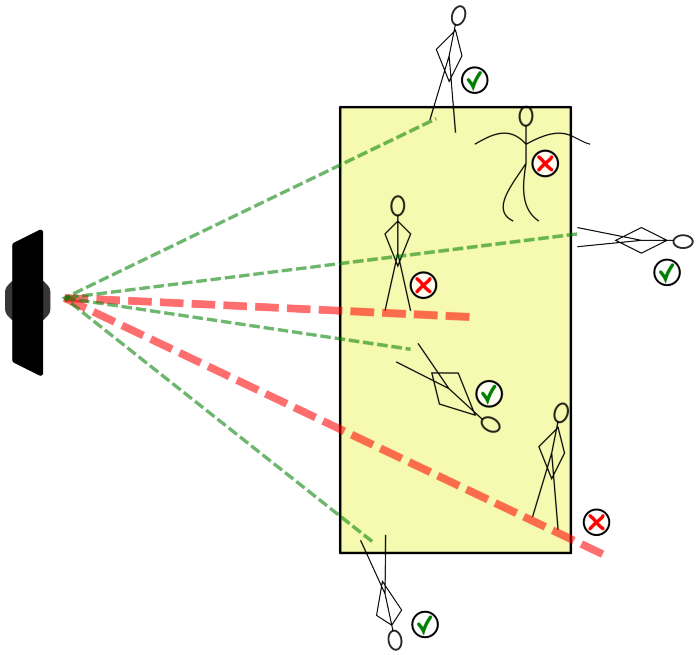

Abbildung 4.15: Optimale Spielhaltung

4.4 Minigolfausflug

Die Parametrisierung der `Physics-Engine` sorgt für das richtige Verhalten des Ballverlaufs. Um dies genauer zu analysieren, besuchten wir zweimal den Minigolfplatz nahe der HSR.

Abbildung 4.16: Eintrittskarten Minigolf

Neben den physikalischen Eigenschaften (Tab. 3.3) wurden weitere Erkenntnisse gesammelt:

Unebenheiten Durch die Unebenheiten der Bahn rollt der Ball manchmal, nachdem er kurz zum Stehen gekommen ist, wieder etwas in eine Richtung zurück. Dies haucht dem eher statischen Spiel etwas Leben ein.

Statisches Spielverhalten Eine Minigolfanlage besteht meist nur aus statischen Komponenten. Ein klassisches Beispiel für eine dynamische Komponente ist die Windmühle, die mit ihren Flügeln das Loch in regelmässigen Abständen versperrt. Beim Augmented Virtuality Minigolf wären diverse solcher virtueller, dynamischer Hindernisse denkbar.

4.5 Persönliche Berichte

4.5.1 Philipp Eichmannn

Nachdem wir im fünften Semester die Studienarbeit bei Prof. Oliver Augenstein erfolgreich abgeschlossen hatten, war für Roman und mich klar, dass wir als Bachelorarbeit wieder eine Arbeit im Computergrafikbereich schreiben wollen.

Prof. Augenstein unterbreitete uns noch vor Ende der Studienarbeit den Vorschlag für ein Augmented Virtuality Minigolfspiel.
Nach Erhalt der Ausgangslage arbeiteten wir uns in die Thematik und neuen Technologien ein. Im Team teilten wir schon zu Beginn die Arbeit auf. Roman widmete sich der Kinect und ich vertiefte mich mit der Physics-Engine und dem Game-Framework. Die Aufgabenstellung beinhaltete eine Vielzahl neuer Themen und dementsprechend intensiv habe ich mich damit während den ersten Woche auseinandergesetzt.

XNA und C# waren komplett neu für mich. Ich hatte zwar bereits einige Erfahrung in der Frontend-Programmierung, jedoch war ich nicht mit dem Workflow einer 3D-Engine wie XNA vertraut. So schienen anfangs auch einfache Aufgaben nur schwer lösbar. Nach einer Einarbeitungsphase kam ich aber gut damit zu recht und konnte das GUI relativ zügig erstellen.
Während Roman mit der Kinect beschäftigt war, versuchte ich zu eruieren, wie die Transformation des Spielfelds vom Boden in unser System und wieder per Beamer zurück auf den Boden am besten gelöst werden könnte. Dieses Problem schien zunächst sehr komplex. Auf Rat von Prof. Augenstein konnte ich dann aber die Fragestellung in einzelne Teilprobleme gliedern, was die Lösungsfindung erheblich vereinfachte.
Die Arbeit mit der Physics-Engine verlief reibungslos. Die Dokumentation und Beispiele waren äussert hilfreich, die richtige Parametrisierung zu finden gestaltete sich hingegen etwas schwierig und zeitaufwendig. Allgemein war das Testing im Vergleich zu einem reinen Softwareprojekt sehr zeitintensiv, was sich schlussendlich auch in der Anzahl aufgewendeter Stunden niederschlug.

Wie schon während der Studienarbeit war auch die Zusammenarbeit mit Roman äusserst angenehm. Wir sprachen uns regelmässig ab und halfen uns gegenseitig, wo Probleme auftauchten. Die Umsetzung des Projekts war intensiv, spannend und hat auch sehr viel Spass gemacht.

4.5.2 Roman Giger

Bereits während der Studienarbeit entschieden wir, dass wir die Bachelorarbeit wieder zusammen machen werden. Die konstruktive und lehrreiche Zusammenarbeit mit Prof. Oliver Augenstein während der Studienarbeit war Anlass dazu, ihn um Betreuung für die Bachelorarbeit zu fragen. Er schlug uns ein sehr spannende Aufgabe vor, bei der wir mit heller Begeisterung zusagten.

Die Einarbeitung in die verschiedenen Themengebiete war sehr spannend. Durch die verschiedenen Technologien gestaltete sich die Aufteilung einfach. Philipp übernahm den Teil der Physics- und Game-Engine und ich begann mich mit der Kinect auseinander zu setzen. Nach etwas Recherche über die Funktionsweise und studieren von Beispielprojekten, begannen wir einen Prototyp anzufertigen, der die verschiedenen Komponenten zusam-

menführte.

Der Prototyp funktionierte soweit, dass mittels Bewegungen die von der Kinect aufgenommen wurden, ein Ball am Bildschirm angeschlagen werden konnte. Nun galt es mit dem Beamer ein rechteckiges Spielfeld auf den Boden zu projizieren und mittels Kinect und deren Tiefensensor eine interne, zweidimensionale Repräsentation anzufertigen. Das Ganze auf einen Wurf hinzubekommen gestaltete sich schwieriger als vorerst angenommen. Prof. Augenstein half uns bei der Lösung dieses Problems, welches dann in Teilprobleme zerlegt wurde. Ab und zu sorgten Treiberprobleme für undefiniertes Verhalten. Probleme mit einem USB 3.0 Anschluss wurden erkannt. Das Problem wurde aber jeweils zuerst im Code gesucht. In solchen Fällen wären Tests sehr praktisch gewesen.
Sobald eine Berechnung länger dauerte als die Kinect benötigt um ein neues Frame zu liefern, kamen neue Probleme auf. Nach einigen Wochen wurde die bisherige Event-basierte Implementation in einen synchronen Programmfluss umgeschrieben. Dies erwies sich als sehr wichtigen Schritt, da fortan die Übersichtlichkeit stieg und Fehlverhalten besser eingegrenzt werden konnten.
Schwierig gestaltete sich das Testing. Anfänglich war es einfach mit der Kinect ein Rechteck zu erkennen. Je weiter das Projekt fortschritt desto schwieriger und aufwändiger gestaltete sich das Testing. Im Nachhinein war es ein Fehler die neuen Funktionalitäten immer manuell zu testen. Möglichkeiten wären vorhanden gewesen, Kinectstreams aufzunehmen und somit eine Kinect zu simulieren. Dies hätte von Anfang an geschehen sollen. Gegen Ende des Projektes wäre der Profit nicht mehr gross gewesen. Ohnehin wären wir aber nicht darum herumgekommen das System wöchentlich aufzubauen um das Spielverhalten zu testen bzw. neue Streams aufzunehmen.

Rückblickend gibt es wieder einige Dinge, die beim nächsten Mal anders gemacht würden. Im Bereich Software-Engineering konnte wieder einiges an Erfahrung gesammelt werden. Verschiedene Design Patterns wie Interpreter, State und Template Methodes kamen zum Einsatz. Vor- und Nachteile gewisser Architekturüberlegungen konnten nachvollzogen werden. Lehrreiche Diskussionen mit Philipp über Designentscheide waren ebenso wertvoll wie die mathematischen Konzepte welche angeeignet werden konnten. Eine weitere interessante Erfahrung war die Arbeit mit einem Sensor wie der Kinect, einem recht performanten Stück Technologie, das nicht alles von sich Preis gibt. Allgemein machte ich die Erfahrung Dinge mehr zu hinterfragen.
Wie der Zeitaufwand in Tab. 4.17 zeigt, wurde einiges an Zeit investiert. Der Einsatz verschiedener Technologien brauchte Einarbeitung, dabei war ebenfalls das richtige Skalieren schwierig. Wird die einfache Lösung implementiert, die nicht lange anhält, dafür nicht viel Zeit kostet oder wird von Anfang an mehr angestrebt, obwohl dies unter Umständen nicht nötig wäre. Das Testing war mitunter ein Grund für den Zeitaufwand des Projektes, passende Räumlichkeiten zu finden begleitete uns ebenfalls über weite Strecken der Arbeit. Die Problematiken, mit denen es sich auseinanderzusetzen galt, waren ab und zu doch etwas komplexer und beanspruchten dementsprechend mehr Zeit, diese zu verstehen und zu lösen. Mit 20h pro Woche wäre ein Vorankommen nur sehr langsam möglich gewesen, zumal das Testen und ausprobieren, sowie die Suche nach passenden Räumlichkeiten recht viel Zeit in Anspruch genommen hat. Ziel war es schliesslich, etwas zu entwickeln das funktioniert und spielbar ist und kein: „Wenn wir jetzt das und das und das noch gemacht hätten dann würde es theoretisch funktionieren.“ Da es sich zusätzlich um eine sehr spannende und lehrreiche Arbeit handelt zeigten wir sehr gerne vollen Einsatz, da wir davon nur profitieren konnten. Die Zusammenarbeit mit Philipp war sehr konstruktiv und hat auch sehr viel Spass gemacht.

4.6 Projektplan

4.6.1 Projektumfang

Das Projekt dauert insgesamt 17 Wochen.

- 14 Wochen während dem Semester ist der Zeit Aufwand mindestens 20h pro Woche
- 1 Woche Osterferien
- 2 Wochen übers Semester hinaus mit einem Zeitaufwand von mindestens 45h pro Woche

Dies ergibt pro Person voraussichtlich einen Mindestaufwand von 370h was für unser Team einen Arbeitsaufwand von 740h entspricht.

4.7 Projektorganisation

4.7.1 Organisationsstruktur

- Philipp Eichmann (phi) Entwicklung / Projektleitung
- Roman Giger (rho) Entwicklung / Stv. Projektleitung

4.7.2 Externe Schnittstellen

Prof. Oliver Augenstein ist der Betreuer dieser Bachelorarbeit.

4.8 Projektverlauf

Die Einarbeitungszeit dauerte drei Wochen. Zuerst wurde Recherche betrieben, um die Technologien festzulegen und die Infrastruktur aufgesetzt. Aufgrund der Technologieentscheide konnte das Projekt anschliessend grob in Kinect, Physics-Engine und GUI unterteilt werden. Und die Arbeit an einem ersten Prototyp konnte beginnen.
Ein Prototyp war relativ schnell erstellt. Mithilfe von Bewegungen, die von der Kinect erkannt wurden, konnte ein Ball am Bildschirm angeschlagen. Die Berechnungen dazu geschahen bereits in der Physics-Engine. Daraufhin wurde ein erster grober Plan über die ganze Projektdauer erstellt (Tab. 4.4), der absichtlich sehr optimistisch gestaltet wurde. Fortan wurde täglich ein kurzes Meeting über den Stand der Dinge und das weitere Vorgehen gehalten.

Woche	Meilenstein	Arbeiten
W8		
W9		
W10		
W11	MS1	Zeitplanung, Design, State-Diagramm, Klassen-Diagramme, Game-Engine Refactoring
W12		Research Kalibrierung, Proof of Concept, Integration OpenCV
W13		Architektur Verbesserung, Kalibrierung abgeschlossen

W14		Hindernisse-Integration
W15		Hindernisse-Integration
W16	MS2	Stabilisierung, Refactoring
W17		Refactoring
W18		Game-Feeling, Sounds, Minigolf Ausflug
W19		Menü, Settings, Kinect-Positionierung / Game Initialisierung
W20		Specials (Gravity inkl. Grafik, animierter Untergrund, Schlägerwahl, Ballwahl, BumperObjekt, Geschwindigkeitsanzeige, Bahn malen)
W21	MS3	Zeitpuffer
W22		Abstract, Summary, Poster
W23		Dokumentation
W24	MS4	Dokukomentation

Tabelle 4.4: Erste Planung

- **Meilenstein 1**: Erster Prototyp
- **Meilenstein 2**: Spiel am Boden spielbar
- **Meilenstein 3**: Spiel mit allen Features implementiert
- **Meilenstein 4**: Abgabe

Der nächste Schritt bestand darin, mit dem Beamer von der Seite her ein Rechteck zu projizieren und dieses mit der Kinect so zu erkennen, dass als interne Repräsentation wieder ein Rechteck vorhanden ist. Dieses Beobachterproblem entpuppte sich komplizierter als anfangs angenommen und sorgte für die erste Verzögerung.

Woche	Meilenstein	Arbeiten
W8		Recherche
W9		Einarbeitung Technologien, Prototyping, Use-Cases
W10		Einarbeitung Technologien, Prototyping
W11	MS1	Zeitplanung, Design, State-Diagramm, Klassen-Diagramme, Anbindung der Physics-Engine, Refactoring
W12		Research Kalibrierung, Proof of Concept, Anbindung der Physics-Engine, Integration OpenCV, Adapter-Design
W13		Planungsende, Architektur, Kalibrierung, Kinect-Wrapper
W14		Threading, Kalibrierung, Architektur
W15		Input State-Machine, Hinderniserkennung, Kalibrierung
W16	MS2	Integration Physics-Engine, Wechsel auf synchronen Programmablauf, Hindernisse
W17		Stabilisierung, Refactoring (Timeout)
W18	MS3	Hindernisintegration, Dokumentation, Zwischengespräch

W19		GUI-Architektur, Optimierung Spielfelderkennung, Analyse Golfschläger-Zittern, Logging-Tool Evaluierung, Fehlersuche
W20		Stabilisierung Golfschläger, GUI-Implementation, Main-Architektur, Input Refactoring Timeout to MaxAttempts
W21		Minigolfausflug, Parametrisierung, Autokalibration, GUI-Implementation, Optimierung Physics-Engine
W22	MS4	GUI-Implementation, Abstract, Management Summary, Poster, Code-Cleaning, Testing, Code-Dokumentationstool, Planung Präsentationsaufbau
W23		Dokumentation, Bugfixes, Letzte Tests - CodeFreeze
W24	MS5	Dokumentation

Tabelle 4.5: Projektverlauf

- **Meilenstein 1**: Erster Prototyp: mittels Bewegung den Schläger am Bildschrim bewegen und Ball anschlagen (Kinect, Physics-Engine)
- **Meilenstein 2**: Spielfeld auf Bildschirm rechteckig (Beobachterproblem)
- **Meilenstein 3**: Spiel auf Bildschirm mit richtigen Spielfeldproportionen möglich
- **Meilenstein 4**: Spiel auf Boden möglich, Kalibrierungsvorgang mit neuem GUI, Feature-Freeze
- **Meilenstein 5**: Abgabe

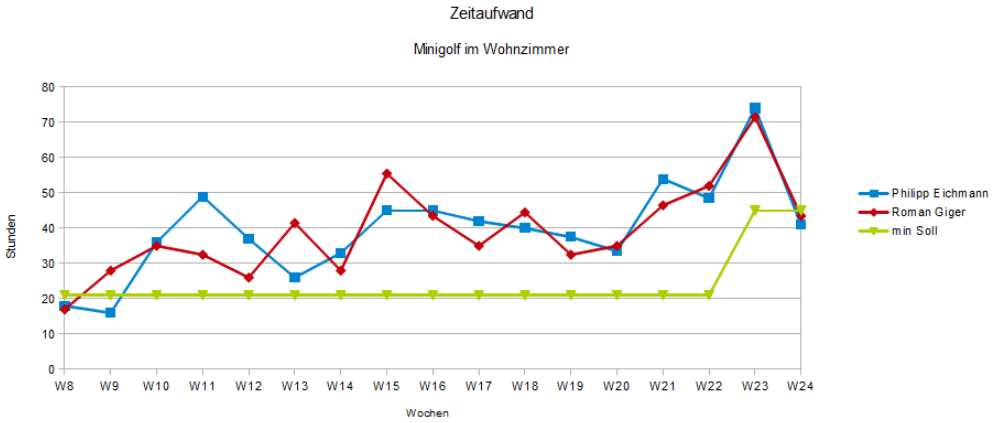

Abbildung 4.17: Zeitaufwand

Name	Zeitaufwand
Philipp Eichmann	675h
Roman Giger	667h
Gesamtaufwand	1342h

Tabelle 4.6: Zeitaufwand

Dazu mehr in den Persönlichen Berichten (4.5).

4.9 Infrastruktur

4.9.1 Hardware

Anzahl	Hardwarebezeichnung	Beschreibung/Zweck
1x	Beamer	Projiziert das Spielfeld auf den Boden
1x	Kinect	Ermöglicht das Scannen des Spielfeldes und des Spielers
1x	Virtueller Server	Ubuntu Server mit Redmine und git-Repository
2x	Workstation	Mit Entwicklungsumgebung und Dokumentationstools
2x	Notebook	Siehe Workstation

Tabelle 4.7: Hardware Infrastruktur

4.9.2 Software

Zweck	Software
Entwicklungsumgebung	Visual Studio 2010
Refactoring and Productivity Tool	Resharper
Kinect API	Microsoft Kinect SDK
GameEngine	XNA
Physics-Engine	Farseer Physics Engine
Image Processing Library	Emgu (C# OpenCV Wrapper)
Versionierungsmanagement	git
Logging Framework	log4net
Code Dokumentation	Sandcastle
Projektdokumentation	LaTeX
Bilderbearbeitung	Adobe Design Collection, Inkscape
Modellierung	Enterprise Architect 8
Zeiterfassung	Redmine
Ticketingsystem	Redmine

Tabelle 4.8: Software Infrastruktur

4.10 Qualitätsmassnahmen

4.10.1 Design by Contract und Codedokumentation

Design by Contract Falls folgende Regeln nicht erfüllt werden, muss dies explizit im XML-doc vermerkt sein.

- Eine Funktion gibt nie `null` zurück
- Eine Funktion darf nie mit `null` als Parameter aufgerufen werden
- Funktionsaufrufe, die Andere voraussetzen, haben entsprechende Vermerke im XML-doc

Diese Contracts dienen dazu, Fehler zu reduzieren und wiederkehrende Validierungen wie `null`-Checks zu vermeiden.

Code Dokumentation Um aus den XML-docs eine Codedokumentation zu erstellen, wird `Sand Castle` verwendet. Auch hierfür wurden Regeln definiert:

- XML-doc befindet sich im `interface` sofern eines vorhanden ist, ansonsten im `class`-File
- Simple, selbsterklärende Funktionen müssen nicht dokumentiert werden
- Werden Einheiten verwendet, muss die Masseinheit angegeben werden
- Werden nicht selbsterklärende Zahlenwerte als Parameter verwendet, müssen diese dokumentiert werden
  - `internal int Width = 1024;` selbsterklärend
  - `internal float Accuracy = 0.5f;` nicht selbsterklärend
- Wirft eine Methode eine `Exception`, wird dies im XML-doc mit entsprechendem Tag vermerkt
- Referenziert eine Methode auf eine andere, wird dies mit dem `seealso` im XML-doc vermerkt

Zur Übersichtlichkeit wurde pro Projekt ein `Constants.cs`-File erstellt. Darin befinden sich alle Konstanten, Settings und Magic-Numbers. Eine Ausnahme sind GUI-Konstanten für das Layout. Im `Util`-Projekt werden allgemein verwendbare Funktionen zur Verfügung gestellt. In den `Util`-Files der einzelnen Projekte werden projektspezifische Funktionen zentralisiert.

Memory Management Die CLR[1] übernimmt das Garbage Collecting. Nicht davon Betroffen sind Unmanaged Ressources. Objekte, die das `IDisposable`-Interface implementieren können mit dem `using` Konstrukt aufgerufen werden.

```
using(IDisposable disposable = new DisposableObject()){
  disposable.Read();
  ...
}
```

[1] Common Language Runtime ist die Laufzeitumgebung von .NET

Ein `using`-Block ruft am Ende automatisch `Dispose()` auf. Daher wird überall wo es möglich ist, mit `using` gearbeitet. Objekte die am Ende aufräumen müssen, implementieren entsprechend `IDispoable`.
Weitere Memory-Leaks können entstehen, wenn Eventhandler nicht deregistriert werden. So besitzt der Publisher eine Referenz auf den Subscriber und hält diesen am Leben. Problematisch wird dies, wenn der Publisher viel länger als der Subscriber lebt. Daher muss dafür gesorgt werden, dass sich die Subscriber jeweils wieder deregistrieren, wenn sie nicht mehr verwendet werden.

4.10.2 Sitzungen

Einmal wöchentlich findet eine Besprechung mit dem Betreuer statt. Protokolle werden keine geführt.

4.10.3 Coderichtlinien

Coderichtlinien entsprechen der Standardeinstellung des Resharpers mit Ausnahme, dass bei `2d` und `3d` Bezeichnungen, das „d“ nach der Zahl klein geschrieben wird wie z.B. `Carpet2d`

4.10.4 Unit-Tests

Testabdeckung ist keine geplant.

4.10.5 Versionsverwaltung

Dokumente und Quellcode werden in geeigneter Struktur im Versionsverwaltungssystem abgelegt. Dies ermöglicht zentralen Zugriff, eine History und vermeidet ein separates Backup.

4.11 Use Cases

UC0: Setup (implementiert)

Use Case Name	Setup
Scope	Minigolf im Wohnzimmer
Level	User Goal
Primary Actor	Spieler
Description	Um Minigolf zu spielen muss, der Spieler die Spielumgebung einrichten
Stakeholders and Interests	Der Spieler will Minigolf spielen. Dazu muss er die Hardware einrichten, diese mit dem Computersystem verbinden und die Software starten
Preconditions	• Benötigte Hard- und Software ist vorhanden • Installationsanleitung ist vorhanden
Success Guarantee	Das System empfängt das Kinectsignal und liefert ein Ausganssignal an den Projektor
Main Success Scenario	1. Der Spieler montiert Projektor oberhalb der Spielfläche und platziert die Kinect gemäss Installationsanleitung. Er schliesst die Kinect an das Computersystem mit der Minigolfsoftware an. 2. Der Spieler startet die Software 3. Die Software wird initialisiert und signalisiert dem Spieler die Bereitschaft
Frequency of Occurence	Immer vor Systemstart
Extensions	**a.* Jederzeit, falls die Verbindung zur Kinect abbricht, signalisiert die Software dem Spieler, dass die Kinect angeschlossen werden muss.
Special Requirements	• Projektor • Kinect • Computer und Minigolfsoftware • genügend Platz vorhanden • abgedunkelter Raum

Tabelle 4.9: UC0: Setup

UC1: Neues Spielfeld erstellen (teilweise implementiert)

Use Case Name	Neues Spielfeld erstellen
Scope	Minigolf im Wohnzimmer
Level	User Goal
Primary Actor	Spieler
Description	Der Spieler betritt die Minigolfeinrichtung und legt Hindernisse auf das projizierte Spielfeld. Die Software erkennt die Objekte und generiert daraus ein virtuelles, zweidimensionales Spielfeld.
Stakeholders and Interests	Der Spieler möchte eine neues Spiefeld mit Hindernissen erstellen. Er will dabei seiner Kreativität freien Lauf lassen und eine unlimitierte Anzahl flacher Objekte beliebiger Form verwenden.
Preconditions	• UC0: Setup
Success Guarantee	Kinect hat alle Hindernisse im projizierten Feld erkannt und deren Form und Positionen berechnet.
Main Success Scenario	1. Die Software projiziert eine rechteckiges Spielfeld auf den Boden und markiert den Startpunkt sowie das Loch 2. Der Spieler legt beliebige Objekte auf das Spielfeld 3. Die Software erkennt mittels der Tiefendaten von der Kinect die Form und Position aller Hindernisse und markiert sie mit einer projizierten Beschriftung oder Umrandung
Frequency of Occurence	So oft der Spieler will

Extensions	*3a.* Falls die Software nicht alle Hindernisse detektiert, erkennt dies der Benutzer anhand der fehlenden projizierten Markierung. Siehe auch 3d. *3b.* Die Software erkennt, dass die platzierten Hindernisse es verunmöglichen, das Loch zu treffen und signalisiert dies dem Spieler. *3c.* Nachdem die Software ein Spielfeld erkannt hat, hat der Spieler die Möglichkeit, dieses zu speichern und zu einem späteren Zeitpunkt wieder zu laden. Dabei müssen die gespeicherten Hindernisse nicht erneut auf das Spielfeld gelegt werden, sondern sie werden aufs Spielfeld projiziert. *3d.* Falls die Software keine Objekte erkennt, wird dies dem Benutzer signalisiert. Worauf er sich entscheiden kann, ob dies gewollt ist oder nicht. Im zweiten Fall muss er u.U. grössere bzw. besser erkennbare Objekte wählen. *3e.* Nachdem das System die Hindernisse erkannt hat, kann der Spieler das erfasste Spielfeld benennen und speichern, um es zu einem späteren Zeitpunkt wieder zu laden.
Special Requirements	• Gegenstände gemäss Spezifikation als Hindernisse
Open Issues	• In welcher Form werden die Spielfelddaten gespeichert? • Können Spielfelder gelöscht werden? • Wie werden ganze Parcours (ca. 15 Spielfelder) erfasst?

Tabelle 4.10: UC1: Neues Spiel erstellen

UC2: Spielfeld laden (nicht implementiert)

Use Case Name	Spielfeld laden
Scope	Minigolf im Wohnzimmer
Level	User Goal
Primary Actor	Spieler
Description	Über ein Menü hat der Spieler die Möglichkeit, entweder bereits erfasste Spielfelder zu laden, die er gemäss UC1 Extension 5e gespeichert hat, oder solche, die bereits mit der Software mitgeliefert wurden.
Stakeholders and Interests	Der Spieler möchte Spielfelder, die er bereits erfasst hat und persistiert wurden, benutzen, um Minigolf darauf zu spielen.
Preconditions	• UC0: Setup • Es wurde mindestens ein Spielfeld gemäss UC1 erstellt oder es werden Spielfelder mit der Software mitgeliefert
Success Guarantee	Die Software projiziert anhand der gespeicherten Spielfelddaten die Hindernisse auf das Spielfeld.
Main Success Scenario	1. Der Spieler wählt im Hauptmenü den Punkt „Laden“ 2. Das System zeigt eine Auswahl der gespeicherten Spielfelder an 3. Der Spieler selektiert ein Spielfeld 4. Das System lädt die Spielfelddaten und projiziert die Hindernisse inkl. virtueller Ball und Zielloch auf das Spielfeld 5. Spiel ist bereit, Spieler kann mit UC3 weiterfahren
Frequency of Occurence	So oft der Spieler will
Extensions	**a.* Jederzeit, falls die Verbindung zur Kinect abbricht, signalisiert die Software dem Spieler, dass die Kinect angeschlossen weden muss. *2a.* Neben der Auswahl an selbsterstellten Spielfelder können sich auch Default-Spielfelder in der Liste befinden.
Special Requirements	-
Open Issues	• Wie sieht die Spielfeldliste aus? Wird ein Thumbnail dazu angezeigt?

Tabelle 4.11: UC2: Spielfeld laden

UC3: Spiel spielen (teilweise implementiert)

Use Case Name	Spiel spielen
Scope	Minigolf im Wohnzimmer
Level	User Goal
Primary Actor	Spieler
Description	Das System projiziert einen virtuellen Minigolfball auf ein gemäss UC1 erstelltes Spielfeld. Der Spieler schlägt mit einem realen Minigolfschläger den Ball an, um ihn ins virtuelle Loch zu treffen. Je nach Anschlagsgeschwindigkeit und Winkel berechnet das System den Weg des Balles und projiziert die Animation auf das Spielfeld. Trifft der Ball auf ein Objekt, verläuft der weitere Weg nach den üblichen physikalischen Gesetzen.
Stakeholders and Interests	Der Spieler möchte während des Spiels eine möglichst realistische Simulation des realen Minigolf erleben. Es muss ihm möglich sein, den Wegverlauf nach physikalischen Gesetzen abzuschätzen.
Preconditions	• UC0: Setup • Es wurde ein Spielfeld gemäss UC1 erstellt oder ein Spielfeld bzw. Parcours geladen
Success Guarantee	System hat Minigolfschlägerbewegung erkannt, den Wegverlauf des Balls berechnet und projiziert die Ballanimation auf das Spiefeld.
Main Success Scenario	1. Der Spieler erstellt ein neues Spielfeld oder wählt ein bereits gespeichertes 2. Der Spieler schlägt mit dem Minigolfschläger den virutellen Ball an 3. Das System berechnet den Wegverlauf des Balls und projiziert die Ballanimation auf das Spielfeld 4. Falls Ball zum Stillstand gekommen ist und sich noch nicht im Loch befindet, gehe zu Punkt 2 5. Das System registriert den Treffer und zeigt eine Statistik zum Spiel an (Anzahl Schläge, durchschnittliche Anschlagsgeschwindigkeit, etc).
Frequency of Occurence	So oft der Spieler will

Extensions	*4a.* Falls die maximale Anzahl Schläge erreicht ist, wird das Spiel für diese Bahn beendet. *4b.* Um den Ball erneut anzuschlagen, muss sich der Spieler auf dem Spielfeld bewegen. Problematisch dabei ist der Körper und der Schattenwurf des Spielers. Dieser stört einerseits die Hinderniserkennung (falls diese auch während des Spiels in Betrieb ist) und verdeckt u.U. die auf das Spielfeld projizierten Elemente wie Ball, Loch und Hindernisse.
Special Requirements	• Minigolfschläger
Open Issues	• Wie beendet man ein Spiel? Wie kommt man zurück zum Menü?

Tabelle 4.12: UC1: Spiel spielen

Abbildungsverzeichnis

Literaturverzeichnis

[Brad2008] Gary Bradski and Adrian Kaehler. *Learning OpenCV - computer vision with the OpenCV library.* O'Reilly Media, Inc., Sebastopol, CA, 1. aufl. edition, 2008.

[Cann1986] John Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1986.

[Free2010] Barak Freedman, Alexander Shpunt, Meir Machline, and Yoel Arieli, 2010.

[Gamm1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison Wesley, San Francisco, CA, 1. aufl. edition, 1995.

[Iraj2011] Mohammad Saber Iraj and Ali Yavarii. Skin color segmentation in fuzzy ycbcr color space with the mamdani inference. *American Journal of Scientific Research*, 2011.

[KSDK2012] Microsoft. *Kinect for Windows Software Development Kit.* Micorsoft Corporation.

[Kar2011] Abhishek Kar. Skeletal tracking using microsoft kinect. Technical report, Department of Computer Science and Engineering, IIT Kanpur, 2011.

[Khos2011] K. Khoshelham. Accuracy analysis of kinect depth data. Technical report, ITC Faculty of Geo-information Science and Earth Observation, University of Twente., 2011.

[Kim2006] Itaru Kitahara Tomoji Toriyama Hansung Kim, Ryuuki Sakamoto and Kiyoshi Kogure. Robust foreground segmentation from color video sequences using background subtraction with multiple thresholds. Technical report, Knowledge Science Lab, ATR, Keihanna Science City, Kyoto, 619-0288, Japan, 2006.

[MICR] Paul E. Nothnagle, William Chambers, and Michael W. Davidson. Genicam standard. `http://www.microscopyu.com/articles/stereomicroscopy/stereointro.html`. [Online; accessed 14-June-2012].

[Sims1996] Kristian Simsarian and Karl-Petter Akesson. Windows on the world: An example of augmented virtuality. Technical report, Swedish Institute of Computer Science, 1996.

[Sukt2000] Rahul Sukthantar, Robert G. Stockton, and Matthew D. Mullin. Automatic keystone correction for camera-assisted presenstation interfaces. Technical report, The Robotics Institute, Carnegie Mellon University, Pittsburgh, 2000.

[Viol2001] Paul Viola and Michael J. Jones. Robust real-time face detection. Technical report, Microsoft Research, Redmond, WA 98952, USA, 2001.

[Wiki2012] www.wikipedia.org. Methode der kleinsten quadrate. http://de.wikipedia.org/wiki/Methode_der_kleinsten_Quadrate, 2012. [Online; accessed 12-June-2012].

[Wiki2012b] www.wikipedia.org. Tweening. http://de.wikipedia.org/wiki/Tweening, 2012. [Online; accessed 14-June-2012].

Glossary

Antropometrie Die Lehre der Vermessung des menschlichen Körpers und derer Anwendung.

Augmented Virtuality Das Erweitern der virtuellen Welt mit Hilfe von realen Gegenständen oder Spielern wird Augmented Virtuality genannt [Sims1996]. Ein weit bekannterer Begriff bildet den Gegensatz, die Augmented Reality, bei der die Reale Welt mithilfe von virtueller Information erweitert wird.

Imageprocessing Aus einem Bild werden ein neues Bild oder Charakteristiken daraus gewonnen.

Keystone Trapezförmige Verzerrung eines projizierten Bildes.

SDK Software Development Kit

Skalarprodukt Das Skalarprodukt zweier normalisierter Vektoren befindet sich im Wertebereich $\mathbb{W} = [-1, 1]$. Stehen die Vektoren rechtwinklig zueinander so ist das Skalarprodukt 0.

Skeletal Tracking Automatisches Bestimmen und Nachführen des Skeletons mittels Sensordaten.

Skeleton Ein abstraktes Skelett des menschlichen Körpers mit den wichtigsten Gelenkpunkten.

Skin Color Segmentation Dabei wird versucht alle Teile des Bildes zu Filtern auf denen keine Hautfarbe vorhanden ist. Damit lassen sich irrelevante Bereiche für Trackings ausblenden. [Iraj2011]

Tween Verfahren, bei dem Einzelbilder zwischen zwei Schlüsselbildern einer Animation erzeugt werden, um den Eindruck einer flüssigen Veränderung zu erwecken. [Wiki2012b]

Unmanaged Ressources Im Gegensatz zu den Managed Ressouces, werden die Unmanaged Ressouces vom Garbage Collector nicht aufgeräumt.