Teil I

Technischer Bericht

Kapitel 1

Einleitung

In den letzten paar Jahren hat sich im Bereich der Unterhaltungselektronik viel verändert. So war es früher selbstverständlich Videospiele per Controller zu steuern. Diese Art von Interaktion weist in Hinsicht auf die Spielsteuerung Limitationen auf. Der Anwender kann die virtuellen Spielfiguren lediglich mittels Joystick und ein paar wenigen Knöpfen kontrollieren. Durch die Anzahl der Eingabeelemente eines Controllers wird der Benutzer in der Vielfalt seiner Aktionsmöglichkeiten eingeschränkt. Ausserdem wirkt diese Art von Steuerung unnatürlich und ist daher nicht intuitiv. Beispielsweise muss der Spieler bei einem Tennisspiel wissen, welcher Knopf den gewünschten Schlag ausführt. Bei gleichartigen Spielen kann die Tastenbelegung je nach Hardware- oder Software-Hersteller voneinander abweichen.

Heute wird nebst der klassischen Steuerung mittels Controller ein alternatives Eingabekonzept, die Gestensteuerung, verfolgt. Es gibt bereits einige Hersteller, welche die dafür benötigte Hardware bzw. Software liefern. Eines der Produkte, welches Gesten erkennen kann, ist die Kinect von Microsoft. Sie wurde speziell für die Spielkonsole Xbox entwickelt, um den Spielern eine neue Dimension von Spielsteuerung anzubieten. Die Kinect nimmt Spieler und deren Bewegungen auf und integriert diese direkt in Spiele. So kann ein Spieler beispielsweise durch die Bewegung seines Arms einen Tennisball anschlagen. Dies ermöglicht eine natürlicheres und intuitives Spielerlebnis. Zudem bietet die Gestensteuerung dem Benutzer viel mehr Aktionsmöglichkeiten als ein Controller. Das mühsame Erlernen von Tasten-Aktion Zuweisungen entfällt.

Bei dieser Verwendung dient die Kinect lediglich zur Erkennung menschlicher Bewegungen. Die virtuelle Welt wird dem User wie bisher über ein Ausgabegerät, Fernseher oder Beamer, vermittelt. Die Kinect ist von dieser virtuellen Welt entkoppelt.

Was wäre aber, wenn die Kinect nebst den Gesten von Benutzern auch diese virtuelle Welt wahrnehmen würde?

Durch diese Überlegung offenbart sich ein neues Konzept von Augmented Virtuality für Spielinteraktionen. Dabei sieht die Kinect die Projektionen eines Beamers, welche meist verzerrt sind. Diese projizierte Fläche kann nebst den Gesten des Spielers in ein Spiel integriert werden. Ausserdem kann der Beamer das Spielgeschehen beinflussen, indem er unterschiedliche Bilder ausgibt. Aufgrund dieser Begebenheit wäre es denkbar beispielsweise eine Minigolf-Bahn als Spielfeld mittels Beamer auf dem Boden darzustellen. Ein virtueller Ball, welcher ebenfalls vom Beamer projiziert wird, kann von einem Spieler mittels natürlichen Gesten angeschlagen werden. Die Hindernisse der Bahn könnten durch reale Gegenstände, wie z.B. eine Schachtel, auf die Bahn gelegt werden. Die Kinect erkennt diese neuen Objekte und nimmt sie in die virtuelle Welt auf.

Dieses Setup bringt einige Problematiken mit sich. Einerseits müssen diverse Koordinatensysteme untereinander synchronisiert werden. Andererseits müssen Spieleentwickler in der Lage sein, diese komplexe Umgebung nutzen zu können. Die Kalibration der unterschiedlichen Koordinatensysteme haben wir bereits in unserer Studienarbeit im Frühjahr 2013 behandelt.

Da das Prinzip in dieser Form nicht auf dem Markt vorhanden ist, gibt es noch keine Programmbibliothek für Spielentwickler, welche diese Konstellation unterstützt. Aus dieser Erkenntnis leitet sich das Ziel, eine einheitliche Platform für Spieleentwickler zu realisieren, ab.

Um die Funktionalität der Platform demonstrieren zu können, wurde zusätzlich ein kleines Spiel als Proof of Concept umgesetzt. Das Spiel unterstützt Augmented Virtuality und zeigt dadurch neue Möglichkeiten für Spiele. Der Studiengang Informatik der HSR pflegt eine Sammlung von mehreren Informatik Projekten als Ausstellungsmaterial für ihre Informationsanlässe in Form des Projekts "Informatik zum Anfassen". Das Spiel könnte die Vielfalt der bisherigen Projekte erweitern. Die Konzepte, welche sich hinter dem Spiel verstecken, könnten den Besuchern der Anlässe präsentiert werden.

Kapitel 2

Anforderungsanalyse

In der Anforderungsanalyse wird der Rahmen der Bachelorarbeit festgelegt. Es werden funktionale und nichtfunktionale Anforderungen erarbeitet und erste Entscheidungen getroffen.
Ausgehend von der Aufgabenstellung lässt sich die Arbeit in zwei Teilprojekte unterteilen:

- Bibliothek für die Nutzung einer Kinect-Beamer-Umgebung
- Demo-Anwendung für das Projekt "Informatik zum Anfassen"

Da ein Spiel für ein solches Setup prädestiniert ist, wird der Schwerpunkt der Bibliothek auf die Unterstützung von Spielentwicklungen gelegt. Dies schliesst aber andere Anwendungen nicht aus.

2.1 Bibliothek

Die Bibliothek hat folgende Funktionalitäten anzubieten:

- Die Steuerung der Hardware, namentlich Kinect und Beamer, erfolgt über die Bibliothek
- Die Bibliothek ist in der Lage einen Punkt, der vom Beamer projiziert wird, mit der Kinect zu erkennen.
- Die Spielfläche, welche vom Beamer projiziert wird, ist mit einer geeigneten Methode zu abstrahieren.
- Es sollte möglich sein auf die virtuelle Abstraktion zu zeichnen. Das Ergebnis ist in der Beamerprojektion sichtbar.
- Die resultierende Zeichnung sollte winkeltreu sein. Ein Rechteck sollte auch als Rechteck in der Spielfläche erscheinen.

- Ein reales und flaches Objekt kann von der Kinect erkannt werden, sobald es innerhalb der Spielfläche liegt.

Ergänzend werden die nichtfunktionalen Anforderungen gelistet:

- Die Schnittstellen sind dokumentiert.
- Der Autokalibrierungsprozess ist in unter 10 Sekunden abgeschlossen.
- Einzelne Komponenten sind getestet.
- Extension Points bieten dem Spielentwickler flexible Erweiterungsmöglichkeiten.
- Die Architektur unterstützt Erweiterungen und Anpassungen innerhalb der Bibliothek.

2.1.1 Kinmeration

Die Bibliothek wurde Kinmeration genannt. Es ist eine Wortmischung aus den Wörtern Kinect, Beamer und Kalibration. In den nachfolgenden Kapiteln wird Kinmeration als synonym für die Bibliothek verwendet.

Abbildung 2.1: Logo von Kinmeration

Auf dem Logo ist eine verzerrte Spielfläche abgebildet. Die Pfeile sollen die Synchronisation zwischen den unterschiedlichen Koordinatensystemen bildlich darstellen.

2.1.2 Abgrenzung

Folgende Punkte werden in dieser Arbeit weggelassen:

- Gestensteuerung der Kinect
- Erkennung von dreidimensionalen Objekten mit Schattenwurf

2.2 Demo-Anwendung

Eines der Hauptziele unserer Bachelor-Arbeit war eine ansprechende Demo-Anwendung im Rahmen von "Informatik zum Anfassen" zu entwickeln. Diese sollte bei Informationsanlässen der HSR als Aushängeschild für die Fachhochschule dienen. Deshalb sollte die zu präsentierende Demo-Anwendung einen hohen Wiedererkennungswert und eine gewisse Anziehungskraft für Besucher aufweisen. Folgende Ideen wurden während der Analyse besprochen und bewertet.

2.2.1 Reale Sandbox

Beschreibung:
Als Basis für diese Demo-Anwendung dient ein echter Sandkasten (Rahmen und Sand) aus dem Baumarkt. Dieser wird mit Hilfe des Beamers von oben ausgeleuchtet. Das heisst, dass das vom Beamer projizierte Feld genau so ausgerichtet wird, dass es den gesamten Sandkasten abdeckt. Beim Start der Demo-Anwendung wird der aktuelle Stand des Sandes gemessen und als Referenz gespeichert. Danach hat der Besucher die Möglichkeit mittels einer Schaufel ein Loch zu graben oder einen Hügel aufzuschütten. Die Kinect wird dazu benötigt, um jeweils den aktuellen Stand des Sandes auszumessen. Damit ist es sehr einfach zu bestimmen, ob ein Loch oder ein Hügel vorliegt. Diese werden dann mit unterschiedlichen Farben eingefärbt.

Vorteile:

- hervorragender Eyecatcher für Besucher
- hohe Interaktionsmöglichkeit
- baut auf unserer API auf

Nachteile:

- hoher Setup-Aufwand (Aufbau von Sandkasten und Abfüllen mit Sand)
- Logistik stellt ein Problem dar (viel Material muss bei Infoanlässen transportiert werden)
- Testumgebung ist nicht einfach realisierbar (Halterung für Beamer müsste konstruiert werden)

Fazit:
Aufgrund der oben aufgeführten Einschränkungen eignet sich diese Variante weniger für unsere Bachelorarbeit. Die Sandbox würde sich eher als eigenständige, eventuell sogar interdisziplinäre Arbeit eignen. Die Idee an sich ist unserer Meinung nach sehr gut.

2.2.2 Virtuelle Sandbox

Beschreibung:
Da uns die Idee einer realen Sandbox sehr gefallen hat, aber dennoch viele Probleme mit sich bringt, haben wir uns einen anderen Ansatz ausgedacht. Dieser sollte die Problematik mit dem zusätzlichen Material umgehen, indem der Sand "virtualisiert" wird. Dadurch sind lediglich ein Beamer und eine Kinect sowie einige kleinere Objekte notwendig.

Vorteile:

- weniger Setup-Aufwand als bei der echten Sandbox
- baut auf unserer API auf

Nachteile:

- wirkt nicht so authentisch wie die echte Sandbox

Fazit:
Die virtuelle Sandbox würde die Problematik der realen Sandbox umgehen. Die Implementation wäre dadurch durchaus vorstellbar. Leider wirkt diese Variante nicht so interessant wie die echte Sandbox.

2.2.3 Minigolf

Beschreibung:
Bei diesem Spiel wird ein virtueller Ball, welcher vom Beamer projiziert wird, durch ein Minigolf-Spielfeld mit realen Hindernissen gespielt.

Vorteile:

- geringer Setup-Aufwand
- baut auf unserer API auf

Nachteile:

- Gesten-Steuerung muss zusätzlich umgesetzt werden (mehr Zeitaufwand)
- Eine Game Engine muss entwickelt werden (mehr Zeitaufwand)

Fazit:
Die Umsetzung könnte aufgrund der Gesten-Steuerung und das Entwickeln einer Game Engine unerwartete Probleme aufwerfen. Das Problem hierbei wird sein, dass wir zu wenig Zeit zur Verfügung haben um dies realisieren zu können.

2.2.4 Hindernis-Parkour

Beschreibung:
Die ursprüngliche Idee war es in einem ersten Schritt auf dem Spielfeld (Beamerfläche) ein Labyrinth aus realen Objekten aufbauen zu können. Anschliessend könnten Besucher ein virtuelles Objekt aus dem Labyrinth heraus führen. Die Idee wurde in einer Besprechung mit Oliver Augenstein noch ausgebaut und verfeinert, sodass es denkbar wäre, dass eine Maus (virtuelles Objekt) einen Käse (reales Objekt) im Labyrinth suchen muss.

Vorteile:

- geringer Setup-Aufwand
- baut auf unserer API auf

Nachteile:

- Game Engine wird benötigt

Fazit:
Ohne die zusätzliche Verfeinerung mit dem Maus-Käse-Prinzip, wäre der Hindernis-Parkour wahrscheinlich eher uninteressant für Besucher. Das Maus-Käse-Spiel hat das Potential zum Eyecatcher für die unterschiedlichen Zielgruppen (Kinder, Schüler etc.) zu werden. Die Umsetzung der Game Engine könnte aber dazu führen, dass die Zeit sehr knapp wird.

2.2.5 Evaluation und Fazit

Anhand der vorangegangenen Analyse der unterschiedlichen Ideen konnten wir die Auswahl auf zwei Favoriten reduzieren. Nach der ersten Runde standen noch die virtuelle Sandbox sowie der Hindernis-Parkour zur Diskussion. Die Entscheidung, welche der beiden Ideen wir umsetzen möchten, fiel uns nicht leicht, da beide ihre speziellen Reize bieten. Deshalb nutzten wir eine Nutzwertanalyse um eine Auswahl zu erleichtern. Wir haben uns die wichtigsten Kriterien, welche das Produkt aufweisen sollte, notiert und gewichtet (1-5, wobei 5 der obersten Priorität entspricht). Anschliessend konnten wir mit Hilfe von Punkten von 1-10 beurteilen, wie gut die einzelnen Ideen unsere Kriterien erfüllen. Dabei entspricht 10 dem höchsten Erfüllungsgrad. Aus der Auswertung ging der Hindernis-Parkour ganz knapp als Sieger hervor.

		Ideen für "Informatik zum Anfassen"			
		Virtual Sandbox		*Hindernis-Parkour*	
Entscheidungskriterium	**Gewicht**	Erfüllt	G * E	Erfüllt	G * E
Eyecatcher Potential	5	7	**35**	8	**40**
API Unterstützung	4	7	**28**	7	**28**
Portabilität	1	5	**5**	5	**5**
Innovation	3	4	**12**	7	**21**
Erweiterbarkeit	2	7	**14**	4	**8**
	Total		**94**		**102**

Abbildung 2.2: Nutzwertanalyse zur Auswahl der Ideen

2.3 Visualisierung der Algorithmen

Nebst der Demo-Anwendung und der Bibliothek verfolgte unsere Bachelor-Arbeit ein weiteres Ziel, das Darstellen einiger unserer erarbeiteten Algorithmen. Dies soll vor allem die Demo-Anwendung im Rahmen des Projekt ”Informatik zum Anfassen” unterstützen. Die Visualisierung von Algorithmen dient vor allem dazu , den Besuchern erklären zu können, wie unsere Anwendung im Hintergrund arbeitet. Für die Umsetzung dieses Ziels gibt es zwei Möglichkeiten, von denen wir uns auf eine fokussieren sollten.

1. Calibration Wizard

Die erste Möglichkeit ist ein sogenannter Calibration Wizard, welcher es dem Benutzer ermöglicht einzelne Schritte unserer Kinect-Beamer-Kalibration visualisieren und erklären zu können. Der Calibration Wizard würde somit vor allem die Algorithmen, welche wir in unserer Studienarbeit [LV13] entwickelt haben, behandeln. Der Fokus liegt dabei auf der eigentlichen Kalibration der Kinect-Beamer-Umgebung.

2. Demo-Anwendung

Als zweite Möglichkeit bietet sich die Demo-Anwendung, welche wir während der Bachelor-Arbeit ausarbeiten, an. Dabei stehen verschiedene Algorithmen im Vordergrund. Einerseits ist es denkbar einige Algorithmen unserer Studienarbeit [LV13] zu präsentieren. Andererseits können wir die Algorithmen erläutern, welche wir neu entwickeln, um die Demo-Anwendung realisieren zu können.

2.3.1 Evaluation und Fazit

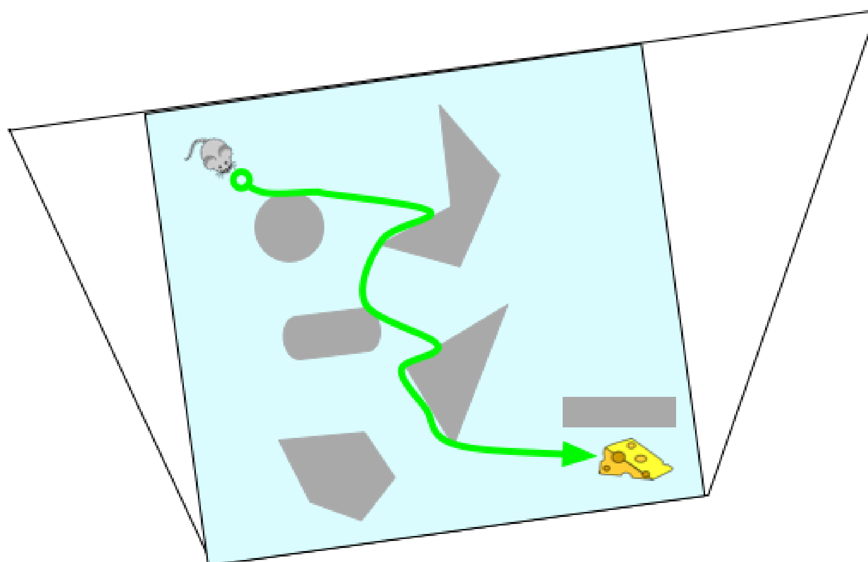
Während der Elaboration-Phase haben wir beide Möglichkeiten analysiert. Schliesslich haben wir uns für die Visualisierung der Demo-Anwendung entschieden, da diese unserer Meinung nach für die Besucher von ”Informatik zum Anfassen” spannender ist. Vor allem das Darstellen der Algorithmen, welche sich hinter dem Hindernis-Parkour verstecken, sind für die Besucher interessant. Die Kalibration könnte unter Umständen für einige Zielgruppen zu technisch oder mathematisch sein.

Obwohl wir auch den Calibration Wizard angedacht haben, wollen wir im Bericht nicht weiter darauf eingehen und unseren Fokus auf die Demo-Anwendung legen. Mockups und Ideen für den Calibration Wizard sind im Anhang C vorzufinden.

2.4 aMAZEingMouse

2.4.1 Kurzübersicht

Dem Start der Demo-Anwendung geht eine vollständige, automatische Kalibration der Beamer-/Kinect-Umgebung voraus. Sobald das Spiel gestartet wurde, wird eine virtuelle Maus irgendwo auf dem Spielfeld platziert. Nun haben Besucher die Möglichkeit reale Gegenstände als Hindernisse auf das Feld zu legen. Des Weiteren liegt es in der Verantwortung der Besucher ein spezielles reales Objekt, den Käse, zu positionieren. Dies ist zugleich der Startschuss für die virtuelle Maus den Käse zu suchen. Während der Käsesuche weicht die Maus geschickt den Hindernissen aus bis sie den Käse gefunden hat.



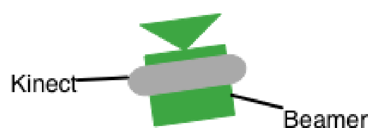


Abbildung 2.3: Schematischer Aufbau des Maus-Käse-Spiels

2.4.2 Modi

2.4.2.1 Spiel-Modi

1) Minimaler Modus
Die oben beschriebenen Schritte Maus-Projektion, Hindernis-Platzierung und Käse-Positionierung werden sequentiell durchgeführt. Verschiebung der Hindernisse oder des Käses sowie weiteres Hinzufügen oder Entfernen von Hindernissen ist in diesem Modus nicht erlaubt. Zudem werden die Hindernisse mit einer einheitlichen Farbe, welche für die Objekterkennung ideal ist, von uns zur Verfügung gestellt. Persönliche Gegenstände oder Kleidungsstücke werden nicht unterstützt.

1b) beliebige Hindernisse
Eine erste Erweiterung des minimalen Modus wäre die Unterstützung von beliebig farbigen Hindernissen, welche von den Besuchern ausgewählt werden. Einzige Einschränkung wären gelb-farbige Gegenstände, da diese noch nicht vom Käse-Objekt unterschieden werden können.

2) eingeschränkter Live Modus
In diesem Modus wäre es denkbar, dass die Hindernisse während sich die Maus auf der Suche nach dem Käse befindet verschoben werden dürfen. Auch das Hinzufügen und Entfernen von Hindernissen ist möglich. Einzige Einschränkung ist die Position des Käses, welche nicht verändert werden darf.

3) ultimativer Live Modus
Was im eingeschränkten Live Modus noch nicht möglich war, wird nun erlaubt. Der Käse darf zur Laufzeit bewegt werden. Dies bedeutet, dass die Maus unter Umständen ständig neue Wege in Betracht ziehen muss, was das Spiel auf die Spitze treibt.

3b) Katz-und-Maus-Spiel
Um das Spiel noch realistischer und authentischer zu gestalten, wäre eine Erweiterung denkbar. Die Hände der Besucher würden dabei die zentrale Rolle spielen. Diese könnten als Katzen interpretiert werden, welche die Maus unbedingt vermeiden möchte. Die Maus würde dadurch vor Händen der Besucher zurückschrecken und einen grossen Bogen um diese machen.

2.4.2.2 Debug/Demo-Modus

Dieser Modus ist speziell für die Präsentation der umgesetzten Algorithmen, welche sich hinter der Käsesuche verbergen, gedacht. Dabei ist es unter anderem möglich, dass man die erkannten Hindernisse einfärben kann. Des Weiteren lässt sich die der gewählte Weg der Maus visualisieren. Je nach dem wie wir den Käsesuch-Algorithmus implementieren, bieten sich noch einige weitere Möglichkeiten der Visualisierung an. Es wäre auch denkbar mehr als einen Algorithmus zu realisieren. Dann würde es sehr eindrücklich sein, die unterschiedlichen Ansätze visuell vergleichen zu können. Der Demo-Modus wird vom Moderator bedient und erläutert.

2.4.3 Mockups: aMAZEingMouse

Unsere Demo-Anwendung wird, wie im Konzept vorgestellt, primär auf der Beamerfläche gespielt. Dennoch ist es unserer Meinung nach sinnvoll, auf dem Notebook-Screen oder auf dem eigentlichen Monitor des Computers ein Hauptmenü mit minimalen Funktionen anzubieten. Die Funktionen beschränken sich im Wesentlichen auf das Starten bzw. Beenden des Spiels sowie einem Demo-Modus, welcher aktiviert oder deaktiviert werden kann. Der Demo-Modus dient dazu, den Besuchern einige unserer Algorithmen, welche wir im Rahmen des Spiels entwickelt und implementiert haben, präsentieren zu können.

Die Demo-Anwendung startet im sogenannten Main Screen und bietet dort lediglich die Möglichkeit das Spiel zu starten oder aber auch zu beenden, falls die Anwendung ungewollt geöffnet wurde.

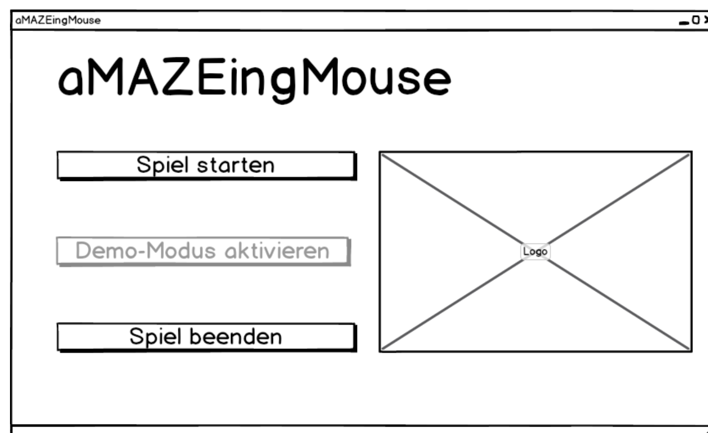


Abbildung 2.4: Demo-Anwendung - Main Screen

Sobald das Spiel erfolgreich gestartet wurde, können Besucher auf dem vom Beamer projizierten Bild das Spiel ausprobieren. Im Anwendungsfenster wird nun ein zusätzlicher Button "Demo-Modus aktivieren" freigeschaltet. Dieser ermöglicht es dem Moderator spannende Informationen bezüglich der verwendeten Algorithmen direkt im Beamerbild einblenden zu können.

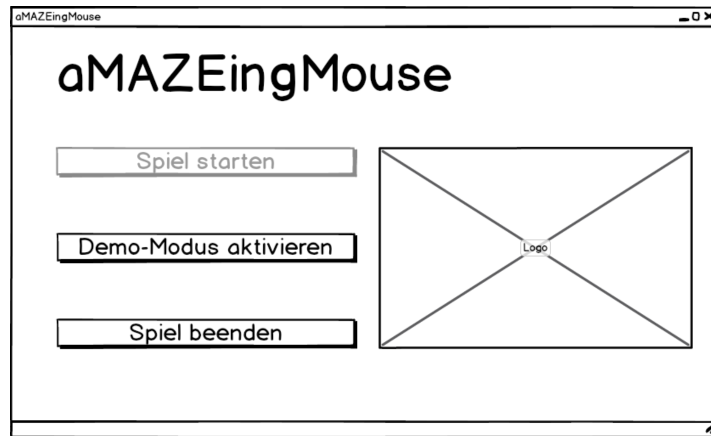


Abbildung 2.5: Demo-Anwendung - Demo-Modus Aktivierung

Natürlich kann der Demo-Modus auch wieder deaktiviert werden. Zudem kann das Spiel jederzeit beendet werden.

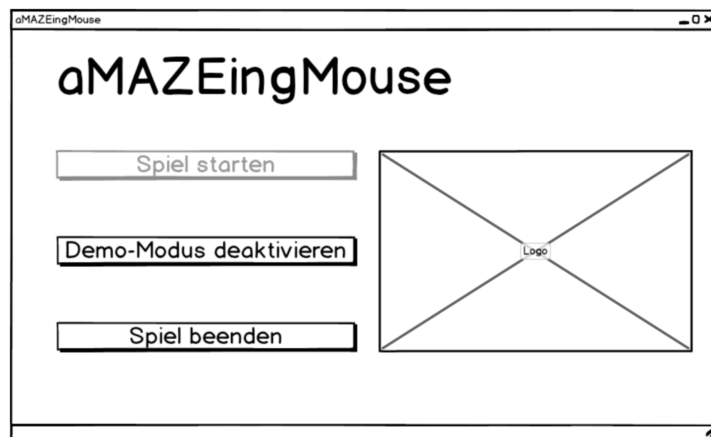


Abbildung 2.6: Demo-Anwendung - Demo-Modus beenden

2.4.4 Funktionale Anforderungen

”aMAZEingMouse” sollte am Ende des Projekts Folgendes können:

- Reale Objekte werden korrekt als Spielobjekte erkannt.
- Die virtuelle Maus ist in der Lage den physischen Käse zu finden.
- Die virtuelle Maus kann während ihrer Suche nach dem Käse Hindernissen ausweichen.
- Es ist möglich einen Demo-Modus mit einigen unterschiedlichen Visualisierungen von Algorithmen zu starten.
- Die Maus startet ihre Suche, sobald das Käseobjekt ins Spielfeld gelegt wurde.
- Der Live-Modus, also das Verschieben von Hindernissen und Käse während einer laufenden Käsesuche, ist nicht zwingend zu implementieren.

2.4.5 Nichtfunktionale Anforderungen

Die Anwendung muss auf grossen Screens gut lesbar sein, da meist viele Leute gleichzeitig am Stand angesprochen werden. Die Benutzeroberfläche muss zudem responsive sein. Lange Warte-/Blockierzeiten sind nicht akzeptabel.

Eine geführte Präsentation der Kalibration bzw. der Algorithmen wird innerhalb von 5-7 Minuten durchgeführt. Dies entspricht in etwa der durchschnittlichen Zeit, die ein Besucher erfahrungsgemäss höchstens an einem Stand verbringt. Dabei wird das Demo-Spiel von den Benutzern durchschnittlich 3 Minuten benutzt.

Die Anforderungen bezüglich des Projekts ”Informatik zum Anfassen” wurden in einem Meeting am 25.09.2013 mit Prof. Dr Markus Stolze, Studiengangleiter Informatik der HSR, ausgearbeitet.

Kapitel 3

Entwurf

Das nachfolgende Kapitel geht auf die wichtigsten Konzepte und Überlegungen ein, welche während der Arbeit ausgearbeitet wurden. Sie bildet zudem die Grundlage für das nachfolgende Realisierungskapitel.

3.1 Kinmeration

Kinmeration wird hauptsächlich in Spielentwicklungen verwendet. Somit werden die Konzepte und die Architektur auf diese ausgerichtet. Eine Umgebung in der Kinmeration genutzt wird könnte wie folgt aussehen:

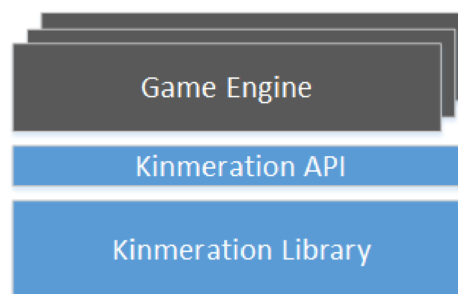


Abbildung 3.1: Systemumgebung von Kinmeration

Eine GameEngine hat mittels API Zugriff auf die Services, welche Kinmeration anbietet. Die angeboteten Services sind aber unabhängig von der GameEngine, somit spielt es keine Rolle um was für eine GameEngine es sich handelt. Trotzdem soll es möglich sein aus GameEngine Sicht auf die Daten von Kinmeration aufzubauen und diese zu bearbeiten oder zu erweitern.

3.1.1 Schichtenmodell

Um die Komplexität zu vereinfachen, wurde ein Layer-Prinzip wie im Buch [FB96] beschrieben angewandt. Die Bibliothek wird in zwei horizontale Schichten unterteilt.

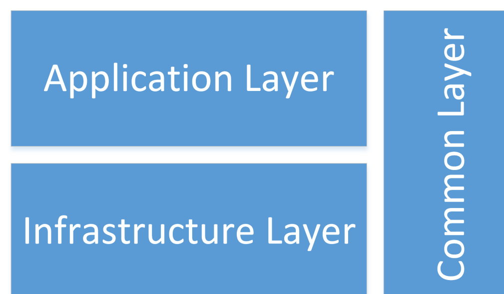


Abbildung 3.2: Logische Schichten von Kinmeration

ApplicationLayer
Der ApplicationLayer beinhaltet high level Services, wie beispielsweise die Abstraktion der Spielfläche. Er baut auf den Services des InfrastructureLayers auf.
InfrastructureLayer
Die Ansteuerung der Hardware, in diesem Fall Kinect und Beamer, wird in diesem Layer realisiert. Dabei werden die Daten aufbereitet und in interne Datenstrukturen verwaltet, welche wiederum über eine Serviceschnittstelle abgerufen werden können.
CommonLayer
Dem CommonLayer kommt eine spezielle Rolle zu. Er enthält sämtliche Datenstrukturen, die sowohl vom ApplicationLayer als auch vom InfrastructureLayer genutzt wird. Ausserdem sind generische Algorithmen, welche auch in anderen Projekten genutzt werden können, diesem Layer zugeordnet.

3.1.2 Kalibration

Die Kalibration umfasst das Synchronisieren der verschiedenen Koordinatensysteme, welche in der Studienarbeit [LV13] beschrieben sind. Einfach gesagt, muss ein Punkt in jedem Koordinatensystem identifizierbar sein.

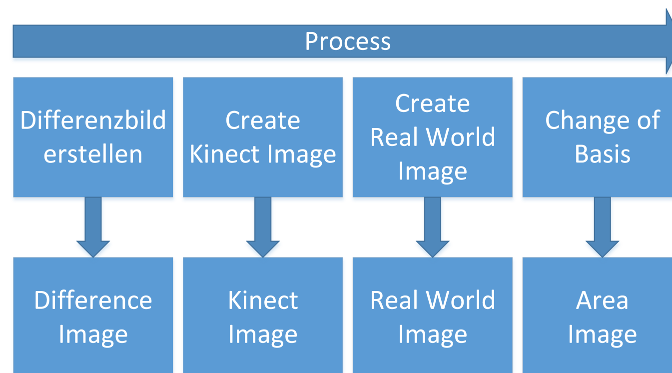


Abbildung 3.3: Kalibrationsprozess mit Ergebnis jedes Schrittes

Algorithmen und der Kalibrierungsprozess konnte aus der Studienarbeit übernommen werden. Die grösste Änderung liegt in der Repräsentation der Resultate von den einzelnen Schritten. Diese wurden vereinheitlich.

3.1.3 Images

Die Kinect SDK von Microsoft liefert die Daten der Kinect in einem Bitstrom an. Diese Repräsentationsart ist ungeeignet, da ein zweidimensionales kartesisches Koordinatensystem innerhalb eines eindimensionalen Arrays verborgen wird.
Eine einfachere Repräsentation bietet ein Image Ansatz mit folgenden Eigenschaften:

- variable Auflösung
- jeder Punkt ist über zwei Parametern addressierbar

Somit werden die Datenen des Farb- und Tiefenstreams der Kinect in ein Image umgewandelt. Aufbauend auf diesem Datencontainer können nun Algorithmen die Daten bearbeiten.

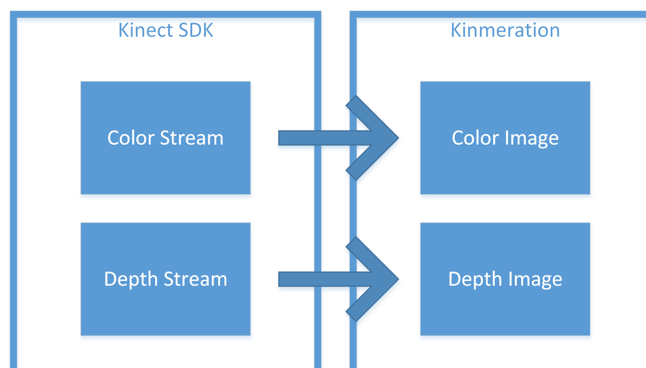


Abbildung 3.4: Stream zu Image

Die Auflösung, in welcher die Kinect ein Bild aufnimmt, kann bei der Initalisierung festgelegt werden. Für den Kalibrationsprozess ist es wichtig, dass Farb- und Tiefenbild die gleiche Auflösung aufweisen.
Die Vereinigung von Farb- und Tiefendaten ist ein Spezialfall. Die Kinect SDK bietet aber die Möglichkeit an die Tiefendaten auf die Farbdaten auszurichten. Das Resultat wird innerhalb von Kinmeration in ein Kinect Image gespeichert.

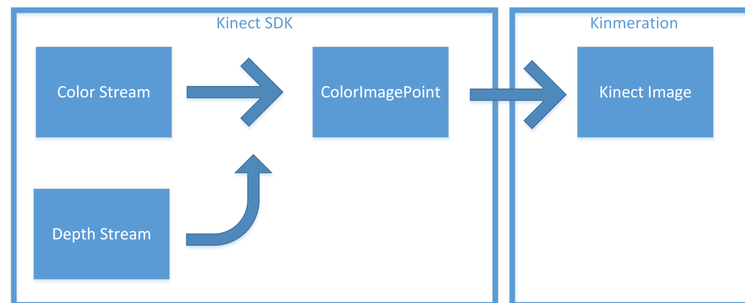


Abbildung 3.5: KinectImage als Vereinigung von Color und Depth Stream

RealWorld-Image, Difference-Image und Area-Image sind virtuelle Images, welche bei der Kalibration erzeugt werden. Die Auflösung dieser Images richtet sich nach der Auflösung des Color Images.

3.1.4 Punkte

Die Repräsentation eines Punktes in einem bestimmten Koordinatensystem muss ebenfalls beachtet werden:

Punkt	Einheit	Koordinatensystem
ColorPoint	Pixel	2D kartesisch
DepthPoint	Pixel	2D kartesisch
DifferencePoint	Pixel	2D kartesisch
KinectPoint	Pixel/Millimeter	3D nicht kartesisch
RealWorldPoint	Millimeter	3D kartesisch
AreaPoint	Millimeter	2D kartesisch

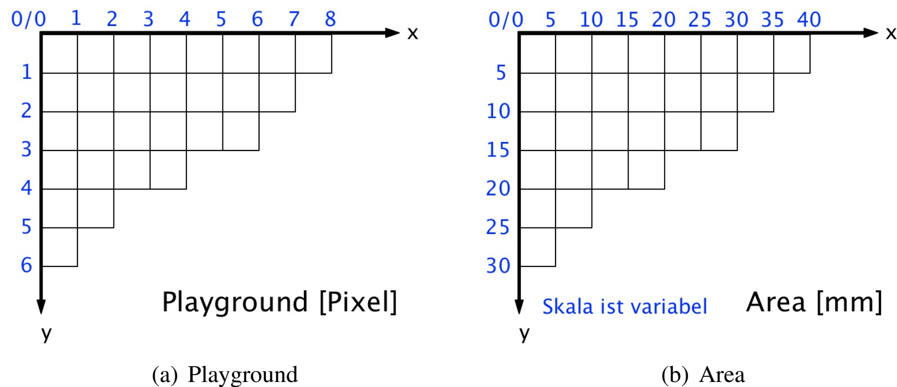
Tabelle 3.1: Punkte

3.1.5 Koordinatensystem

Mit der Kombination eines Images und einem bestimmten Punkt, kann eine einheitliche Repräsentation verschiedener Koordinatensysteme sichergestellt werden. Beispielsweise kann nun in einem ColorImage ein ColorPoint an der Position (1,1) genommen werden. Der AreaPoint im AreaImage an der selben Position entspricht genau dem gleichen Punkt, einfach in einem anderen Koordinatensystem.

3.1.6 Playground

Die Playground ist eine Abstraktion der Spielfläche. Sie basiert auf einem zwei dimensionalen kartesischen Koordinatensystem. Die Idee besteht darin, die Area in ein Raster zu unterteilen. Jedes einzelne Quadrat kann dann wiederum addressiert werden.



(a) Playground (b) Area

Abbildung 3.6: Playground vs. Area

Die Playground basiert auf dem Area Koordinatensystem. Da durch den Basiswechsel ein Eckpunkt zum Koordinatenursprung wird, können negative Positionen entstehen. Um die Playground trotzdem als Image behandeln zu können, wird ein Offset um die Area gelegt. Somit funktioniert die Addressierung weiterhin.

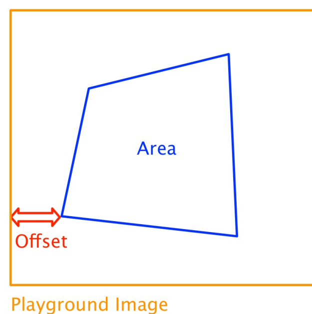


Abbildung 3.7: Playground umschliesst Area

Umrechnung

Für die Umrechnung zwischen Area-Koordinaten (mm) und Playground-Koordinaten (Pixel) wird einen Skalierungsfaktor α benötigt. Der Skalierungsfaktor gibt an, wie gross ein Pixel in Millimeter ist.

Die Umrechnung eines Punktes von Area zu Playground funktioniert wie folgt:

$$\begin{aligned} x_P &= x_A/\alpha \\ y_P &= y_A/\alpha \end{aligned} \tag{3.1}$$

Dynamische Bestimmung
Der Skalierungsfaktor kann mit folgender Formel dynamisch bestimmt:

$$\alpha = \sqrt{\frac{area}{width * height}} \tag{3.2}$$

Dabei sind *width* und *height* einerseits die Dimensionen des Playgrounds in Pixel und andererseits ist *area* die Fläche der Area in mm^2. Die Berechnung von *area* geschieht mittels der Flächenformel für beliebige Vierecke. [Wik13b]

$$area = \frac{1}{2}\sqrt{|\vec{e}|^2|\vec{f}|^2 - (\vec{e} \cdot \vec{f})^2} \tag{3.3}$$

In obiger Formel stehen *e* und *f* für die beiden Diagonalen des Vierecks.

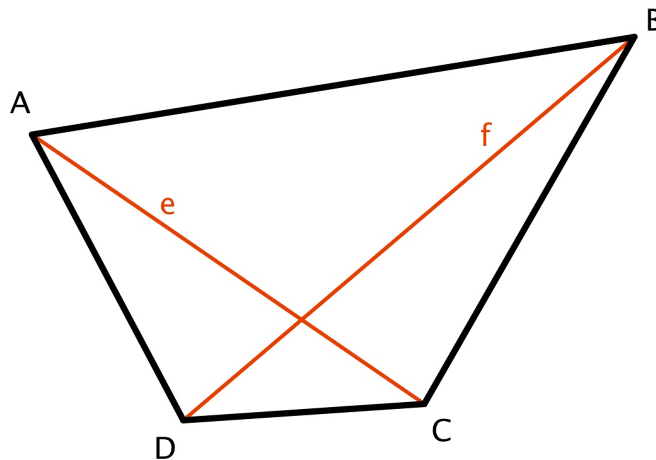


Abbildung 3.8: Diagonalen im Viereck

3.1.7 Varianten von Playgrounds

Damit der Umgang mit dem Playground für die Bibliothek-Benutzer möglichst flexibel wird, werden mehrere unterschiedliche Varianten von Playgrounds geplant. Diese erlauben es, eine optimale Spielfläche für das zu entwickelnde Spiel zu finden und anschliessend einfach verwenden zu können.

1) MaximalPlayground
Die MaximalPlayground ist der mächtigste Vertreter der Playgrounds-Familie. Er stellt ein 1:1-Mapping zur gesamten Area-Fläche dar. Daher umfasst er jeden Punkt der Area-Fläche und erlaubt so das Zeichnen aller möglichen Punkte.

2a) SquarePlayground
Dieser Playground bietet dem Benutzer ein möglichst grosses Quadrat innerhalb des verwendbaren Bereichs (Area-Fläche) an.

2b) RectanglePlayground
Der RectanglePlayground liefert ein möglichst grosses Rechteck innerhalb der Area-Fläche zurück.

2c) CirclePlayground
Es wird ein Kreis mit möglichst grossem Radius innerhalb der Area-Fläche gesucht und dem Entwickler zur Verfügung gestellt.

3) RotatedPlayground
Dieser spezielle Playground bietet die Möglichkeit einen gefundenen, geometrischen Playground nach Wunsch zu drehen. Das führt dazu, dass der Playground in der Area anders zu liegen kommt. Der Entwickler muss sich aber nicht um diese Rotationsproblematik kümmern.

4) ScaledPlayground
Der ScaledPlayground ermöglicht es dem Bibliothek-Benutzer einen Playground nach seinen Wünschen bezüglich Höhe und Breite zu erstellen.

Kombinationen

Der Entwickler hat die Möglichkeit die unterschiedlichen Playgrounds so zu kombinieren, wie er es für sein Spiel bzw. seine Anwendung benötigt. Die nachfolgende Abbildung 3.9 illustriert mögliche Kombinationen der geplanten Playgrounds. Dabei bedeuten die Pfeile, dass ein bestimmter Playground-Typ auf einem anderen Playground-Typ aufbauen darf. Dadurch entstehen Ketten von Playgrounds, welche sehr flexibel zusammengestellt werden können. Wie man sieht, stellt der `MaximalPlayground` den Ursprung aller Unterformen von Playgrounds dar. Dies macht Sinn, da er zugleich die grösste Fläche aufweist.

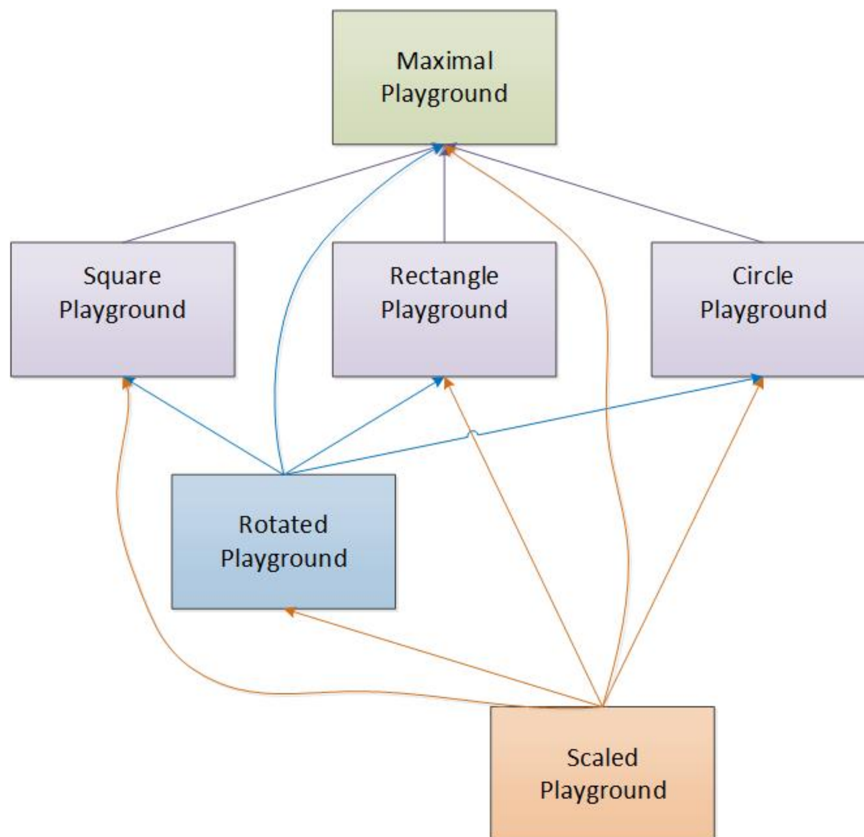


Abbildung 3.9: Kombinationen der Playground-Varianten

3.1.8 Objektdetektierung

Der Nachteil des Kalibrierungsprozesses ist, dass lediglich ein 1:1-Mapping basierend auf der Auflösung der Kinect-Kamera vorgenommen wird. Die Playground weist hingegen eine flexible Auflöung auf. In Situationen bei der die Playground-Auflösung grösser als die der Kinect ist, entstehen Lücken während des 1:1-Mappings. Aufgrund dieser Erkenntnis war es notwendig, die Objektdetektierung um Korrekturparameter zu erweitern.

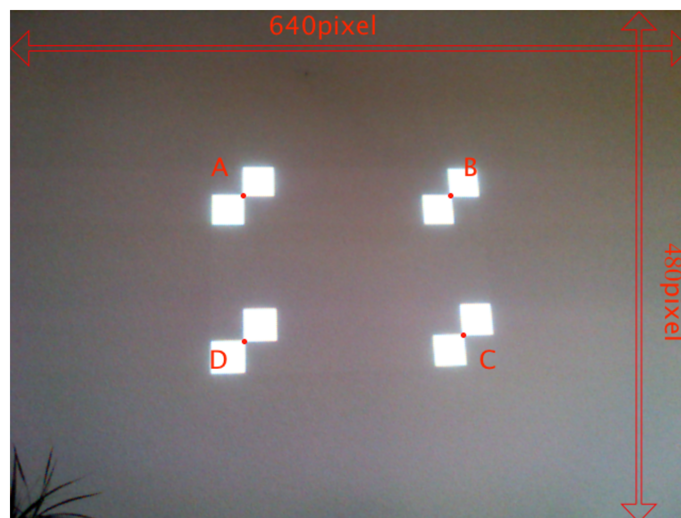


Abbildung 3.10: Dimensionen des Farbbildes (gesehen von der Kinect)

Abbildung 3.11: Dimensionen der Playground

Das Mapping von Farb-Koordinaten zu Playground-Koordinaten erweist sich als schwierig, da eine kleine Anzahl von Farb-Punkten (Abbildung 3.10) in die hochauflösende Playground (Abbildung 3.11) übersetzt werden müssen. Das Problem ist, dass nicht jeder Playground-Punkt von einem Farb-Punkt aus erreicht werden kann. Bei einem 1:1-Mapping wird jeder der 640x480 Punkte der Kinect einem Playground-Punkt zugeteilt. Das Hauptproblem dieses 1:1-Mappings ist, dass erkannte Objekte, welche in Kinect-Punkten vorliegen, mit einer sehr geringen Pixeldichte in der Playground dargestellt werden. Dies wiederum führt beim Zeichnen auf den Beamer dazu, dass die Objekte nicht komplett ausgefüllt visualisiert werden. Es ist also notwendig, dass ein Kinect-Punkt mehrere Playground-Punkte ansprechen kann.

Eine weitere Hürde stellt die dynamische Playground dar. Das Erstellen einer Playground liefert stets unterschiedlich grosse Playgrounds, da je nach Positionierung des Beamers und der Kinect die Grösse des Spielfeld varieren kann.
Die nachfolgende Abbildung veranschaulicht das Problem, dass erkannte Objekte (blaues Rechteck, rechts) nicht komplett ausgefüllt gezeichnet werden können.

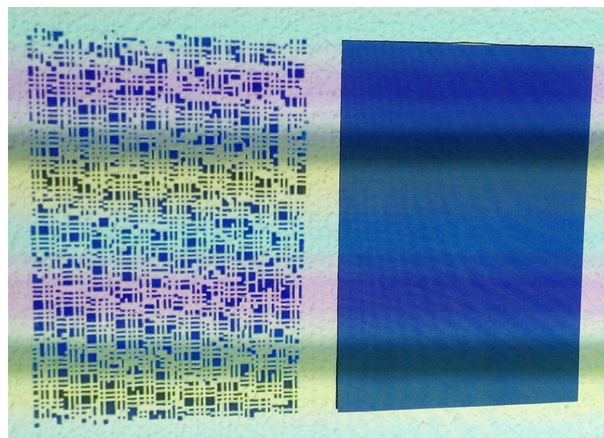


Abbildung 3.12: Problematik 1:1-Mapping

Lösung

Die Position der Kalibrationspunkte (A,B,C,D) können wir im Bild, welches die Kinect aufgenommen hat, genau bestimmen. Nach der Kalibration und dem erstellen einer Playground, sind die Koordinaten dieser Punkte auch in der Playground bekannt. Anhand der unterschiedlichen Seitenlängen vom Kinect Bild verglichen zu der Playground, lassen sich zwei Skalierungsfaktoren (χ und ψ) ableiten. Diese beiden Faktoren helfen einem Kinect-Punkt mehrere Playground-Punkte zuzuweisen. Das Errechnen der unterschiedlichen Seitenlängen und das anschliessende Vergleichen liefern zugleich eine dynamische Lösung, welche mit verschieden grossen Playgrounds funktioniert. So ist es beispielsweise möglich, dass in einem Setup ein 1:4-Mapping bestimmt wird und in einem grösseren Setup ein 1:8-Mapping vorliegt.

χ: Skalierungsfaktor in x-Richtung
ψ: Skalierungsfaktor in y-Richtung

Die Berechnung erfolgt mit den nachstehenden Formeln:

distColorAB = Distanz von Punkt A nach B im Kinect Bild
distColorBC = Distanz von Punkt B nach C im Kinect Bild
distColorCD = Distanz von Punkt C nach D im Kinect Bild
distColorDA = Distanz von Punkt D nach A im Kinect Bild

distPlayAB = Distanz von Punkt A nach B in der Playground
distPlayBC = Distanz von Punkt B nach C in der Playground
distPlayCD = Distanz von Punkt C nach D in der Playground
distPlayDA = Distanz von Punkt D nach A in der Playground

$$\begin{aligned} \chi &= \left(\frac{\frac{distColorAB + distColorCD}{2}}{\frac{distPlayAB + distPlayCD}{2}} \right) \\ \psi &= \left(\frac{\frac{distColorBC + distColorDA}{2}}{\frac{distPlayBC + distPlayDA}{2}} \right) \end{aligned} \tag{3.4}$$

Nach der Berechnung von χ und ψ werden diese noch einem Ceiling unterzogen, d.h. auf eine ganzzahlige Zahl aufgerundet.

Das komplett ausgefüllte Objekt (links) beweist die Funktionsweise der erarbeiteten Lösung.

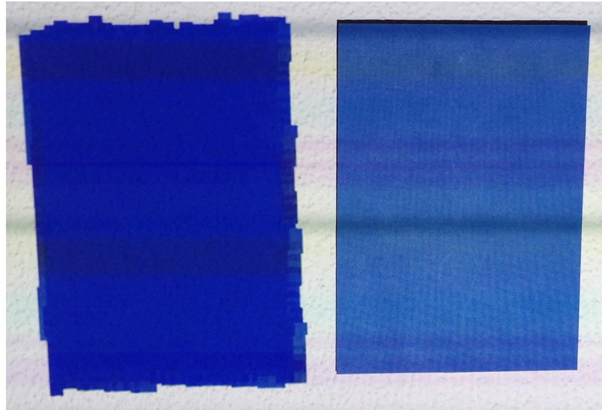


Abbildung 3.13: Kinect/Playground-Mapping: Lösung durch dynamisches Mapping

3.2 aMAZEingMouse

3.2.1 Marketing

Bereits während der Anforderungsanalyse haben wir uns Gedanken darüber gemacht, wie wir unser Spiel nennen und präsentieren wollen. In der Entwurfsphase kam als Nächstes ein Logo dazu, welches unser Spiel eindeutig identifiziert. Auch der Name verrät schon einige Charakteristiken des Spiels. ”Maze” ist englisch und bedeutet übersetzt ”Labyrinth”. Eine ”tolle, verblüffende” Maus spielt die Hauptrolle in unserem Spiel. Vor allem hinsichtlich dem Projekt ”Informatik zum Anfassen”, welches unterschiedliche Zielgruppen ansprechen soll, eignet sich der Name und das Logo hervorragend.

Abbildung 3.14: Logo aMAZEingMouse

3.2.1.1 Aufbau

Das Spiel stellt ein Proof of Concept für ”Kinmeration” dar, da wir durch ein funktionierendes Spiel zeigen können, dass die Bibliothekt für Spielentwicklungen genutzt werden kann. Aufgrund dessen haben wir beschlossen, den Aufbau bzw. die Architektur des Spiels möglichst einfach zu halten. Der Hauptteil bildet die Game Engine, welche die ganze Spiellogik enthält.

Die Algorithmen, welche für das Spiel benötigt werden, sind ebenfalls ein wesentlicher Teil von aMAZEingMouse. Unser Fokus liegt vor allem auf dem Entwickeln von funktionierenden Algorithmen.

Benutzeroberfläche und Fehlerbehandlung sind weitere wichtige Punkte. Wir wollen in diesen beiden Bereichen eine grundlegende stabile Lösung liefern. Bei genügend Zeit sind zusätzliche Erweiterungen denkbar.

3.2.2 Interaktion aMAZEingMouse - Kinmeration API

Das Spiel "aMAZEingMouse" muss mit der Kinmeration API interagieren, was im nachfolgenden Sequenzdiagramm visualisiert wird. Das Sequenzdiagramm beschreibt das Beispiel eines kompletten Durchlaufs des Spiels.

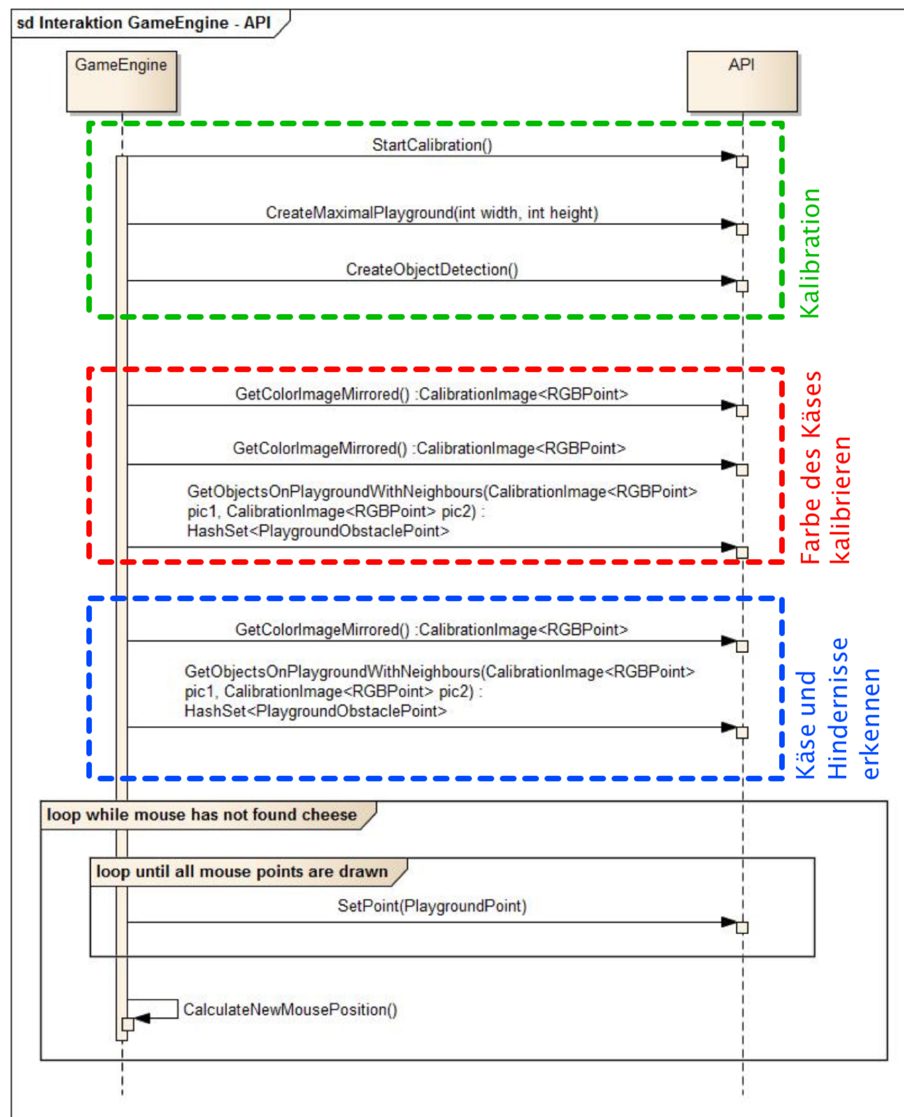


Abbildung 3.15: Interaktion GameEngine - API

3.2.3 Spiel-Modus

In einem ersten Schritt wird die Kalibration von Kinect und Beamer vorgenommen. Anschliessend wird das eigentliche Spiel "aMAZEingMouse" initialisiert. Dialoge weisen den Benutzer darauf hin, welche Aktion als Nächstes ansteht. Zu Beginn muss die Farbe des Käse-Objektes kalibriert werden. Dabei wird der Benutzer dazu aufgefordert das gewünschte Käse-Objekt ins Spielfeld zu legen. So wird die Referenzfarbe für das Erkennen des Käses bestimmt. Danach kann der Benutzer nach Belieben ein Labyrinth bestehend aus realen Objekten zusammenstellen. Dabei muss beachtet werden, dass die Hindernisse andere Farben aufweisen müssen als das kalibrierte Käse-Objekt. Nebst den Hindernissen platziert der Benutzer zugleich auch den Käse im Spielfeld. Nun kann die Maus dazu veranlasst werden, sich auf die Käsesuche zu begeben. Mit Hilfe des entwickelten Algorithmus findet die Maus den Käse und das Spiel ist beendet. Auf Wunsch des Benutzers kann dann ein neues Labyrinth aufgebaut und die Käsesuche der Maus neu gestartet werden.

3.2.4 Demo-Modus

Ein erweiterter Modus, der sogenannte Demo-Modus, bietet dem Benutzer die Möglichkeit diverse Visualisierungen während des laufenden Spiels einzublenden. Dabei ist es beispielsweise möglich die erkannten Hindernisse einzufärben. Des Weiteren können die berechneten Ecken des Spielfelds eingeblendet werden. Dieser Demo-Modus kann zu jeder Zeit während des Spiels aktiviert werden, unabhängig davon, ob die Maus läuft oder das Spiel pausiert wurde. Der Demo-Modus kann einfach per Knopfdruck beendet werden.

Der Demo-Modus ist vor allem für die Ausstellung "Informatik zum Anfassen" spannend, da dort Besucher sehen können, was genau hinter der eigentlichen Käsesuche steckt. Der Moderator kann diese Zeit nutzen Informationen über Algorithmen oder andere Details zu vermitteln.

3.2.5 Hauptmenü

Der Fokus der Spielentwicklung lag vor allem auf dem Ausarbeiten der Spiellogik und die dazu nötigen Algorithmen. Zudem war es wichtig, dass das Spiel auf "Kinmeration" basiert, um dessen Funktionalität und Verwendbarkeit prüfen zu können. Aus diesen Gründen haben wir uns dafür entschieden, die Benutzeroberfläche vorerst einfach zu gestalten.

Es soll ein einziges Hauptmenü realisiert werden, welches dem Benutzer erlaubt alle Schritte zu steuern. Das Hauptmenü umfasst zwei Teilbereiche. Die obere Hälfte bietet alle Funktionen, die für das eigentliche Spielen von "aMAZEingMouse" benötigt werden. Die untere Hälfte bildet den Demo-Modus ab. Dort ist es möglich während des Spiels zusätzliche Informationen auszugeben. Die Steuerung läuft jeweils über dieses Fenster. Spielausgaben erfolgen mit Ausnahme weniger Dialoge ausschliesslich auf dem Beamer.

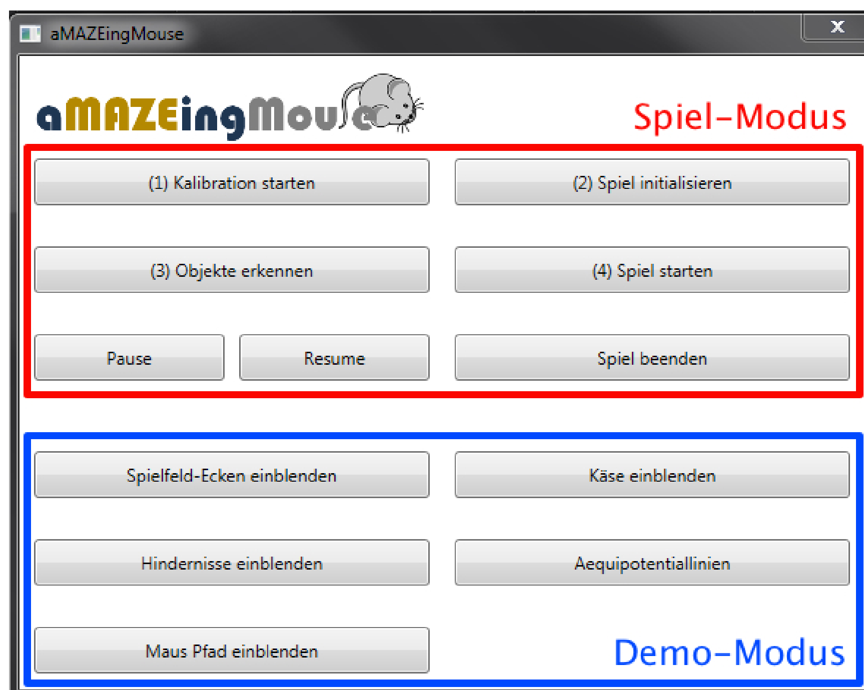


Abbildung 3.16: Hauptmenü aMAZEingMouse

3.2.6 Spielobjekte

Um unser eigentliches Spiel umsetzen zu können, benötigen wir unterschiedliche Typen von Spielobjekten. Beispielsweise haben die verschiedenen Spielobjekte andere physikalische Eigenschaften, wie in Abschnitt 3.2.9 beschrieben. Die Positionen der Spielobjekte werden als Pixel-Koordinaten in einer Playground angegeben. Jede Position erhält zusätzlich einen Spielobjekt-Typ zugeteilt. So wissen wir stets wo sich Maus, Hindernisse und Käse innerhalb der Playground befinden.

Innerhalb der Game Engine werden alle Spielobjekte als Instanzen der Klasse `GamePoint` abstrahiert. Die GamePoints lassen sich 1:1 in PlaygroundPoints umwandeln, da sie sich lediglich durch ein zusätzliches Property, *Type*, unterscheiden.

Die nachfolgende Tabelle erläutert kurz die einzelnen Spielobjekte:

Spielobjekt	**Beschreibung**
Maus	Das zentrale Spielobjekt von aMAZEingMouse. Es stellt eine virtuelle, dynamische, projizierte Maus dar, die sich beliebig auf der Playground bewegen kann. Ihr Ziel ist es, den Käse zu finden.
Käse	Nebst der Maus ein wichtiges Spielobjekt, da er das Ziel der Maus darstellt. Der Käse ist im Gegensatz zur Maus ein reales Objekt, welches vom Spieler ins Spielfeld gelegt wird. Zudem ist der Käse ein statisches Objekt.
Hindernis	Die Hindernisse sind ebenfalls reale, statische Objekte, welche innerhalb des Spielfelds platziert werden können. Sie bringen Spannung ins Spiel, da die Maus jetzt andere Wege suchen muss, um an den Hindernissen vorbei zu kommen.
virtuelles Hindernis	Die virtuellen Hindernisse können nicht direkt vom Benutzer beeinflusst werden. Sie dienen dazu, dass die Maus komplexere Hindernisse umgehen kann. Im Abschnitt 3.2.9 werden diese Spielobjekte genauer beschrieben.

Tabelle 3.2: Spielobjekte

3.2.7 Game States & Debug States

Zu einer Game Engine gehören unter anderem Game States, welche die Zustände der Game Engine repräsentieren. Zustandsübergänge werden einerseits durch Inputs des Benutzers und andererseits durch die Game Engine selbst eingeleitet. Jeder Game State bzw. Debug State hat seine eigene Funktion und ist für die Ausführung von bestimmten Operationen verantwortlich. Die States wiederspiegeln dabei die Funktionen der beiden Modi, Spiel-Modus und Demo-Modus.

Game Zustand	Beschreibung
Started	Applikation wurde gestartet und wartet auf User Input.
Calibrating	Kalibration von Kinect und Beamer wurde ausgelöst und wird ausgeführt.
InitializingGame	In einem ersten Schritt wird ein Playground für das Spiel eingerichtet. Anschliessend wird die Objekterkennung sowie das Spiel initialisiert.
DetectingObstacles	Die vom Benutzer platzierten Objekte werden detektiert und in Spiel-Punkte umgewandelt.
StartingGame	Die Suche nach dem Käse wird eingeleitet.
Gaming	Die nächsten Schritte der Maus werden berechnet und anschliessend gezeichnet.
Stopped	Die Maus hat den Käse gefunden. Das Spiel wartet auf User Input.
Paused	Das Spiel wird pausiert.

Tabelle 3.3: Game States für Spiel-Modus

Debug Zustand	Beschreibung
None	Debugging ist ausgeschaltet.
DebuggingEdges	Die Eckpunkte der erkannten Playground werden eingeblendet.
DebuggingCheese	Der Mittelpunkt des erkannten Käse-Objektes wird angezeigt. (Ziel der Maus)
DebuggingObstacles	Alle erkannten Hindernisse werden eingefärbt.
DebuggingPotential	Ausgehend von der aktuellen Position der Maus werden die Äquipotentiallinien , welche in diesem Moment auf sie wirken, eingeblendet.
DebuggingMousePath	Der bisher zurückgelegte Weg der Maus wird eingezeichnet.

Tabelle 3.4: Debug States für Demo-Modus

Das folgende Diagramm soll den Zusammenhang der einzelnen Zustände visualisieren (blau = Game States, orange = Debug States). Dabei befindet sich das Spiel zu jedem Zeitpunkt in einem GameState und in einem DebugState.

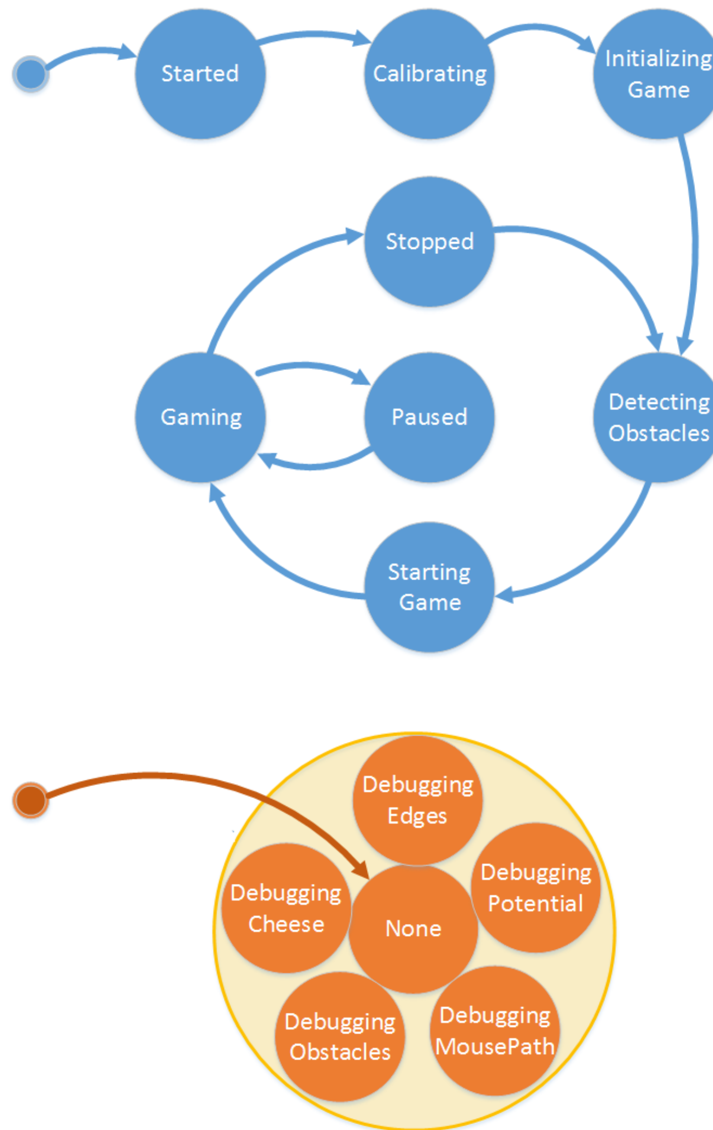


Abbildung 3.17: Game States und Debug States

Die oben beschriebenen GameStates werden während des Spiels durchlaufen. Der Benutzer kann durch seine Inputs das Spiel nach seinen Belieben steuern. Damit dies möglich ist, haben wir eine einfache Game Loop implementiert. Die Game Loop prüft in fest definierten Zeitintervallen in welchem Zustand sie sich momentan befindet und leitet dann die nötigen Operationen ein. Durch User Inputs wird der Zustand der Game Loop verändert, was dazu führt, dass die gewünschten Funktionen ausgeführt werden.

Dabei ist zu beachten, dass Game States und Debug States parallel und unabhängig voneinander angesteuert werden können. Dies macht den Demo-Modus sehr flexibel. Es ist möglich, dass Debug Informationen nebst dem eigentlichen Spiel eingeblendet werden können, ohne dass das Spiel pausiert werden muss. Solange das Spiel noch nicht fertig initialisiert ist, befindet sich der Debug State im ”None”-State. Hat die Maus ihre Suche begonnen, so ist es jederzeit möglich in den gewünschten Debug State zu wechseln. Von jedem Debug State aus ist es möglich in einen beliebigen anderen Debug State überzugehen.

3.2.8 Object Identification

Die erste Implementation der Objekt Identifikation fällt sehr einfach aus. Wir haben uns darauf festgelegt die Objekt Identifikation lediglich durch einen Farbabgleich durchzuführen. Dabei werden beim Generieren des Differenzbildes alle Punkte, welche sich zwischen den beiden Bildern geändert haben, markiert. Um nun die Spielobjekte unterscheiden zu können, muss die Farbe der markierten Punkte bestimmt werden. Anhand jener Farben kann dann entschieden werden, um welchen Typ von Spielobjekt es sich handeln sollte. Das Identifizieren von Objekten könnte zu einem späteren Zeitpunkt mit Hilfe von weiteren Techniken, wie beispielsweise einer Kantendetektion, noch weiter präzisiert werden. Solche Erweiterungen sind aber aufgrund des Zeitbudgets vorerst nicht geplant.

Um eine Farbunterscheidung realisieren zu können, ist es notwendig die benötigten Farben im RGB-Raum zu analysieren. Auch die verschiedenen Grade der einzelnen Farbtöne muss berücksichtigt werden, da beispielsweise nicht jedes Gelb die gleichen RGB-Werte liefert.

Die einfachste Form der Unterscheidung bestimmt Offsets für das Gelb (Käse) sowie für das Blau (Hindernisse). Dass Hindernisse blau sein müssen, wurde in einem nächsten Schritt noch überarbeitet, damit auch Objekte mit anderen Farben als Hindernisse verwendet werden können.

Spielobjekt	Farbe	RGB-Raum [R][G][B]
Käse	Gelb	[245-255][230-255][0-205] dabei gilt: $R \geq G \geq B$
Hindernis	Blau	[0-185][0-242][100-255] dabei gilt $R \leq G \leq B$

Tabelle 3.5: RGB-Farbraum der Spielobjekte

Die RGB-Bereiche der einzelnen Farben bzw. der Grade der Farben wurden anhand von RGB-Farbtabellen, von denen es reichlich im Internet zu finden gibt, zusammengestellt. Problematisch an diesem Ansatz ist, dass äussere Einflüsse, wie z.B. unterschiedliche Hintergründe, unterschiedliche Lichtverhältnisse etc., nicht berücksichtigt werden.

1. Verbesserung: Referenzwert bilden

Beim Testen der Object Identification haben wir sehr schnell bemerkt, dass die Kinect schwächere und teilweise leicht verfälschte Farbwerte liefert als wir zuerst angenommen haben. Dies hat unter anderem mit den externen Einflüssen, welche auf das Bild wirken, zu tun. Zudem wurde uns bewusst, dass wir für das Erstellen der Differenzbilder zur Objekterkennung nicht die Farbe Weiss als Referenz

nehmen sollten. Weiss ist eine sehr grelle Farbe, was dazu führt, dass sie die Objekterkennung bzw. die Sensoren der Kinect in jenem Bereich stört.

Aufgrund dieser beiden Erkenntnisse haben wir die ObjectDetection so angepasst, dass wir als Referenzfarbe Schwarz verwenden. Diese lieferte unter den getesteten Bedingungen (unterschiedliche Hintergrundfarben und Lichtverhältnisse) die besten Werte. Des Weiteren bieten wir nun dem Benutzer die Möglichkeit in einem weiteren Initialisierungsschritt des Spiels den Referenzwert der Farbe Gelb (Käse) bestimmen zu lassen. Mit Hilfe des bestimmten Referenzwertes und einem fixen Threshold lässt sich die Entscheidung, ob es sich um Käse oder Hindernisse handelt, einfacher treffen. Dies bringt auch den Vorteil mit sich, dass Hindernisse in unterschiedlichen Farben genutzt werden können. Lediglich Gelb bleibt dem Käse-Objekt vorbehalten.

2. Verbesserung: Standardabweichung miteinbeziehen

Bis anhin haben wir lediglich den Mittelwert aller RGB-Werte der erkannten Pixel des Käse-Objektes gebildet und diesen als Referenzwert gespeichert. Für das Identifizieren des Käses während dem Spiel haben wir dann die neu erkannten RGB-Werte der Pixel mit dem Referenzwert verglichen. Dabei haben wir eine Toleranz in Form eines statischen Thresholds pro Farbwert zugelassen. Es stellte sich schnellhaft heraus, dass dieser statische Threshold nicht für alle möglichen Setups optimal ist, da teilweise andere Lichtverhältnisse herrschen und andere Einflüsse auftreten.

Deshalb wollten wir einen dynamischen Threshold, welcher sich an das Setup anpassen kann, verwenden. Dazu berechnen wir bei der Bestimmung des Referenzwertes nebst dem Mittelwert zudem die Standardabweichung der gefundenen RGB-Werte der Pixel [Wik13a]. Nun werden die im nächsten Schritt erkannten Pixel mit dem Referenzwert unter Berücksichtigung der Standardabweichung σ als Offset verglichen. Das daraus gewonnene Ergebnis leitet dann die Entscheidung ein, ob der Pixel zu einem Käse-Objekt oder zu einem Hindernis gehört.

$$\sigma = \sqrt{\left| \left(\frac{1}{n} \sum_{i=1}^{n} x_i \right)^2 - \frac{1}{n} \sum_{i=1}^{n} x_i^2 \right|} \tag{3.5}$$

σ stellt in obiger Formel den Threshold dar. x entspricht den Farbwerten der gefundenen Pixel des Differenzbildes. σ wird jeweils für die Farbwerte R, G und B berechnet. So gibt es für jede Komponente des RGB-Farbwertes einen unterschiedlichen, dynamischen Threshold. Dadurch konnte die Präzision der ObjectIdentification verbessert werden.

3.2.9 Algorithmus: Käsesuche

Damit die Maus das von der Kinect erkannte Käse-Objekt finden kann, braucht es einen ausgeklügelten Algorithmus. Der Algorithmus muss die Maus auf dem Weg zum Käse möglichst optimal an den Hindernissen vorbei lenken können.

3.2.9.1 Physikalisches Konzept

Unserem Algorithmus liegt ein physikalischer Ansatz zugrunde. Ladungen mit gleichem Vorzeichen stossen sich ab. Hingegen ziehen sich Potentiale mit unterschiedlichen Vorzeichen an. Dieses einfache Prinzip haben wir auf unsere verschiedenen Spielobjekte abgebildet. Die Maus soll vom Käse angezogen werden, aber von den Hindernissen abgestossen werden. Daraus können wir Folgendes definieren:

Spielobjekt	Ladung
Maus	positiv (+)
Käse	negativ (-)
Hindernis	positiv (+)
virtuelles Hindernis	positiv (+)

Tabelle 3.6: Ladungen der Spielobjekte

Die unterstehende Skizze soll das Konzept visuell unterstreichen. Die Maus wird am Hindernis abgestossen (grüne Pfeile). Sobald sie dem Hindernis ausgewichen ist, steuert sie aufgrund der Anziehungskraft direkt auf den Käse zu.

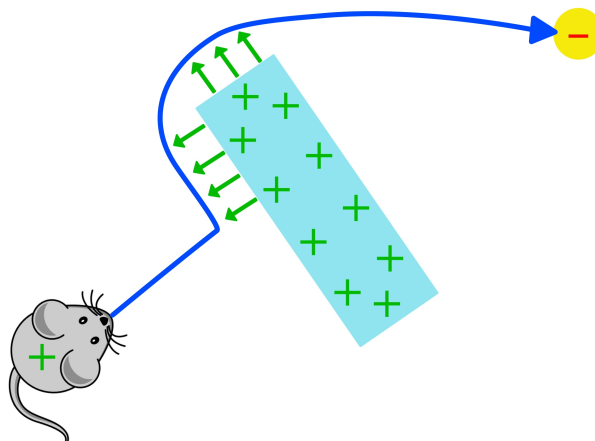


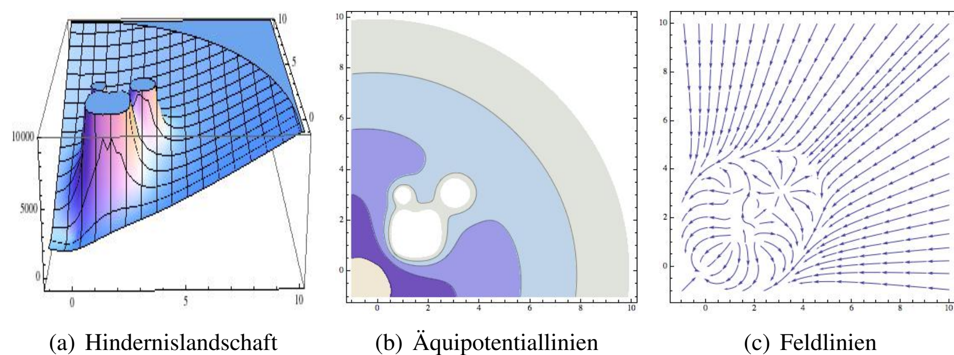
Abbildung 3.18: Suchalgorithmus - physikalisches Konzept

In Abbildung 3.19 stellen drei Bilder die mathematische Funktionsweise unseres Konzepts dar. Alle Bilder basieren auf dem selben Beispiel. Dabei befindet sich der Käse in der unteren linken Ecke. Die Maus startet von oben rechts. Es wurden einige Hindernisse, durch die Erhöhungen in Bild(a) erkennbar, zwischen Maus und Käse platziert.

Bild (a) zeigt die Hindernislandschaft. Wie man deutlich sieht, werden die Hindernisse als Erhöhungen dargestellt. Dies soll für die Maus ein nicht betretbarer Bereich darstellen. Man kann sich die Maus als eine Kugel vorstellen, welche zum tiefsten Punkt hinunter rollt. Der tiefste Punkt wird immer durch den Käse repräsentiert. Die Hindernisse lenken die Maus in eine Richtung ab, damit sie zum Käse gelangt. Wie man sich denken kann, gibt es spezielle Hindernisse, wie z.B. eine 'U'-Form, aus der die Maus nicht mehr hinauskommen kann. Diese Problematik wird in den nachfolgenden Abschnitten behandelt.

Bild (b) illustriert die sogenannten Äquipotentiallinien. Die Farben stellen die unterschiedlichen Potentiale dar, welche an einer bestimmten Position auf die Maus wirken. Hindernisse werden mit der Farbe Weiss markiert.

Bild (c) veranschaulicht die Feldlinien, welche zugleich den gewählten Richtungen der Maus entsprechen. So kann man sehr gut sehen, wie die Maus den einzelnen Hindernissen ausweicht und welchen Weg sie einschlägt.



(a) Hindernislandschaft (b) Äquipotentiallinien (c) Feldlinien

Abbildung 3.19: Physikalisches Konzept

3.2.9.2 1. Phase: Weg zum Käse finden

In der ersten Phase der Implementation des Algorithmus ging es lediglich darum, die Maus zum Käse hin zu bewegen. Dabei wurden noch keine Hindernisse beachtet. Die nachfolgende Grafik illustriert die Koordinaten-Komponenten, welche für die Potentialberechnung benötigt werden.

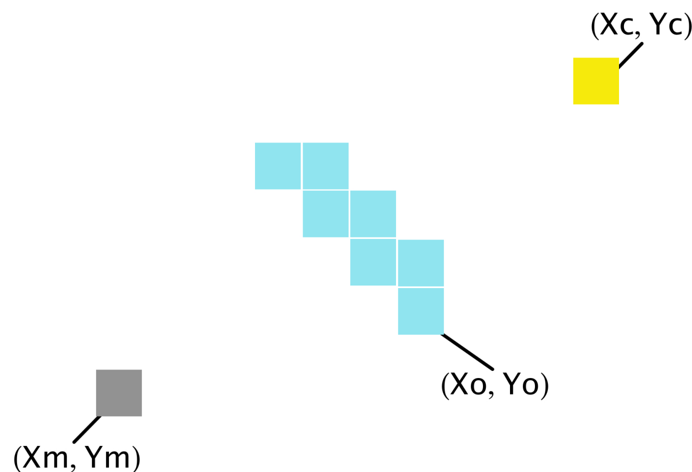


Abbildung 3.20: Koordinaten für mathematische Berechnungen

Die Formel für dieses Szenario ist im Prinzip sehr einfach. Es wird lediglich das Potential des Käses, welches auf die Maus wirkt, beachtet. In den nachfolgenden Formeln repräsentiert x_m, y_m die aktuellen Koordinaten der Maus und x_c, y_c die des Käses.

Der Vorfaktor β kann dazu verwendet werden, um die Anziehung des Käses anzupassen. Dabei entspricht ein grösseres β einer grösseren Anziehung.

$$
\begin{aligned}
x_d &= \frac{\partial}{\partial x_m}\left(\beta * \sqrt{(x_m - x_c)^2 + (y_m - y_c)^2}\right) \\
y_d &= \frac{\partial}{\partial y_m}\left(\beta * \sqrt{(x_m - x_c)^2 + (y_m - y_c)^2}\right)
\end{aligned} \tag{3.6}
$$

Die Ergebnisse x_d und y_d bilden die Komponenten des Richtungsvektors bzw. der an der spezifizierten Mausposition x_m, y_m wirkende Feldvektor.

3.2.9.3 2. Phase: Hindernissen ausweichen

Die zweite Phase befasste sich mit dem Ausweichen an einfachen Hindernissen. In dieser Phase wurde die Formel um die Potentiale der Hindernisse erweitert, da diese nun auf die Maus wirken sollen. Dabei stellen die Potentiale der Hindernisse jeweils eine kurzreichweitig wirkende Kraft dar, welche stärker als die Anziehungskraft des Käses wirkt, wenn die Maus sich in unmittelbarer Nähe des Hindernisses befindet. Die Anziehungskraft des Käses wird als weitreichweitig wirkende Kraft bezeichnet. Im Gegensatz zur 1. Phase müssen die Potential der Hindernisse dazugerechnet werden. In den Formeln stehen x_m, y_m wieder für die aktuellen Koordinaten der Maus und x_o, y_o für die Position der Hindernis-Pixel.

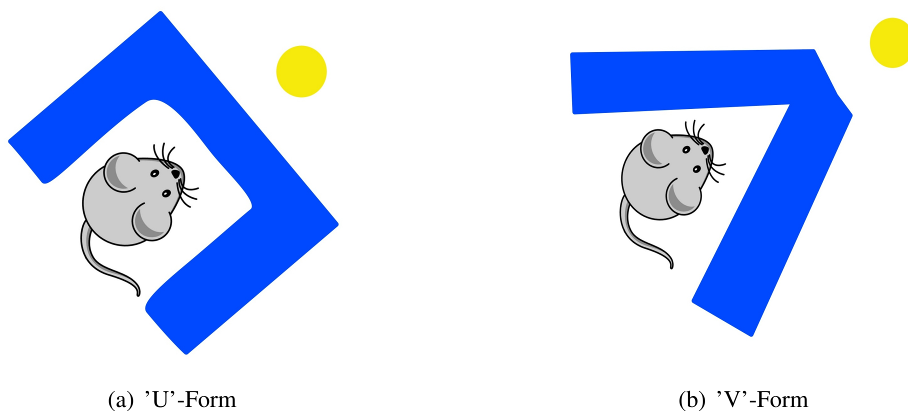
γ und der Exponent δ können dazu genutzt werden, zu definieren wie nahe sich die Maus an die Hindernisse hin bewegen darf. Dabei gilt, je grösser γ ist, desto grösser bleibt der Abstand zwischen Maus und Hindernis. Zudem bedeutet ein grösseres δ, dass die Maus sich näher an das Hindernis heran traut. Es bedarf einiger Experimente um optimale Werte für γ und δ zu finden.

$$
\begin{aligned}
x_d &= \frac{\partial}{\partial x_m}\left(\frac{\gamma}{\sqrt{(x_m - x_o)^2 + (y_m - y_o)^2}^{\delta}}\right) \\
y_d &= \frac{\partial}{\partial y_m}\left(\frac{\gamma}{\sqrt{(x_m - x_o)^2 + (y_m - y_o)^2}^{\delta}}\right)
\end{aligned}
\tag{3.7}
$$

Die Werte x_d und y_d werden für alle Playground-Punkte, welche ein Hindernis darstellen, aufsummiert und anschliessend zum mit Formel 3.6 berechneten Käsepotential summiert. Das Resultat dieser Summe liefert den Richtungs- bzw. Feldvektor, welcher an der Mausposition x_m, y_m unter Berücksichtigung von Hindernissen wirkt.

3.2.9.4 3. Phase: spezielle Hindernisse überwinden

In der dritten Phase war die Herausforderung, dass es einige Hindernisformen gibt, welche der Maus Schwierigkeiten bereiten. Zu dieser Kategorie gehören Hindernisse in Form eines 'U' oder eines 'V'. Diese haben die Besonderheit, dass die Maus in den tiefsten Punkten (lokale Minima) der Hindernisse stecken bleibt und nie wieder hinaus findet. Für diese Fälle musste der Algorithmus intelligenter werden.



(a) 'U'-Form

(b) 'V'-Form

Abbildung 3.21: Beispiele komplexerer Hindernisse

Um diese komplexeren Hindernisse umgehen zu können, haben wir uns für einen Backtracking-Ansatz entschieden. Dabei werden alle Bewegungen der Maus in einem Stack abgelegt. Trifft die Maus auf ein lokales Minimum, aus welchem sie nicht mehr hinaus kommt, setzen wir sie um Schritt für Schritt zurück. Gleichzeitig platzieren wir an der Position des lokalen Minimums ein virtuelles Hindernis, welches nun die Maus aus dem Hindernis hinaus drängt. Wir verwenden virtuelle Hindernisse, welche nur ein Pixel gross sind, dafür aber eine grössere Abstossungskraft besitzen als reale Hindernisse. Das bedeutet, dass wir eine zusätzliche Unterscheidung der physikalischen Begebenheiten der beiden Hindernis-Typen machen mussten. Mathematisch verwenden wir für jedes `virtualObstacle` die Formel 3.7 für "normale" Hindernisse. Der einzige Unterschied ist, dass bei virtuellen Hindernissen für γ und δ andere Werte als bei den "normalen" Hindernissen" gewählt werden. Dies hat zur Folge, dass die Maus von virtuellen Hindernissen stärker abgestossen wird als von den anderen Hindernissen.

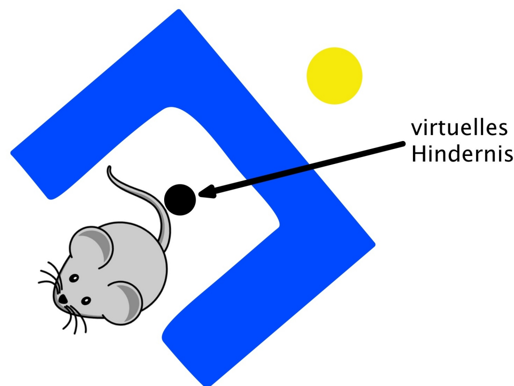


Abbildung 3.22: Backtracking - Beispiel

Insgesamt werden also für das Berechnen des Feldvektors an einer Mausposition x_m, y_m jeweils das Käsepotential (Formel 3.6) mit der Summe aller Hindernisse (Formel 3.7) und der Summe aller virtuellen Hindernisse (Formel 3.7) verrechnet.

3.2.9.5 Optimierung:

Eine der von uns eingebauten Verbesserungen ist, dass auf die Maus nur noch Potentiale, welche sich innerhalb einer bestimmten Entfernung befinden, wirken. Dies hat einerseits einen Vorteil in der Berechnung, da weniger Potentiale in die Rechnung einfliessen. Andererseits können wir so Fehler vermeiden, dass beispielsweise ein sehr grosses Potential (z.B. ein virtualObstacle) innerhalb des gesamten Spielfelds die Maus beeinflussen kann. Vor dieser Optimierung hatten wir das Problem, das häufiges Backtracking so viele virtualObstacles hinterlässt, sodass die Maus von diesen sehr stark verdrängt wurde. Die Käsesuche wurde dadurch verfälscht.

Um die Anziehungskraft des Käses ein wenig stärker zu gestalten, addieren wir zur Bildung des Käsepotentials folgende zwei Summanden:

- Ergebnis von der eigentlichen Formel 3.6 für das Käsepotential
- Ergebnis der Formel 3.7 mit negativem γ und kleinem δ

Dadurch zieht der Käse die Maus stärker an als sie von irgendwelchen Hindernissen abgestossen wird. Dies verhindert, dass ein hinter dem Käse liegendes Hindernis die Maus nicht davon abhalten kann, den Käse zu erreichen.

3.3 Technologieentscheide

- Visual Studio 2012 (Update 3) & Resharper 8
- Kinect SDK 1.8
- Kinect Developer Tools & Kinect Studio 1.8
- Redmine
- TFS (Team Foundation Service) & GIT
- WriteableBitmapEx
- NLog

Da unsere Bachelorarbeit auf unserer Studienarbeit [LV13] des vergangenen Semesters basiert, können wir die Technologieauswahl sehr kurz halten.

Wie bisher verwenden wir als Entwicklungsumgebung das Visual Studio von Microsoft. Um Kinect-Funktionalitäten ansprechen zu können, benötigen wir wiederum die Kinect SDK, welche ebenfalls von Microsoft angeboten wird. Zu Beginn unserer Arbeit wurde gerade die neue Version 1.8 veröffentlicht. Obwohl Version 1.8 keine neuen Features enthält, welche für uns interessant wären, kommt diese zum Einsatz. Auch das dazu gehörige Kinect Studio und die Kinect Developer Tools wurden auf die neuste Version angehoben.

Bereits gegen Ende unserer Studienarbeit haben wir uns entschlossen für das Zeitmanagement ein professionelleres Tool als Excel zu verwenden. In einem vergangenen Projekt haben wir bereits Redmine kennen gelernt. Deshalb haben wir uns schnellhaft darauf geeinigt. Redmine läuft auf einem vServer, welcher freundlicherweise von der HSR zur Verfügung gestellt wird.

Nebst Redmine kam eine weitere neue Technologie hinzu. Der TFS, Team Foundation Service, von Microsoft bietet sowohl eine GIT-Anbindung, um unseren bisherigen Code verwalten zu können, als auch ein einfaches Web-GUI zur Unterstützung unserer angestrebten SCRUM-Vorgehensweise. Der TFS ist bis zu einer Teamgrösse von 5 Personen kostenlos, was uns sehr entgegenkommt.

Kapitel 4

Realisierung

Die Realisierung beleuchtet verschiedene Sichten der Architektur von den beiden Applikationen ”Kinmeration” und ”aMAZEingMouse”.

4.1 Allgemein

Als Projektcontainer wurde eine übliche Visual Studio Solution gewählt. Die Solution wurde wie folgt aufgebaut:

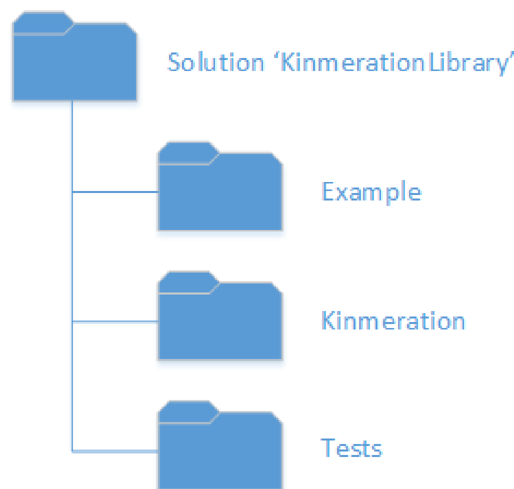


Abbildung 4.1: Solution Strukturierung

Example
In diesem Ordner werden alle Projekte abgelegt, die keine Funktionalität von ”Kinmeration” darstellt. Das Spiel ”aMAZEingMouse” gehört beispielsweise diesem Ordner an, da diese auf ”Kinmeration” aufbaut.

- aMAZEingMouse: Das Hauptprojekt des Spiels

- DebugPics: Ein Projekt welches zur Überprüfung des Kalibrationsprozesses genutzt wurde.

Kinmeration
Komponenten als Projekte, welche ”Kinmeration” direkt betreffen, befinden sich unter diesem Ordner.

Tests
Aus Übersichtsgründen werden die Testprojekte in einem eigenen Container gehalten. Dabei spielt es keine Rolle, ob es die Bibliothek oder das Spiel testet.

4.2 Kinmeration

4.2.1 Komponentenübersicht

Kinmeration besteht aus fünf Hauptkomponenten. Jede Komponente erfüllt einen bestimmten Zweck und ist in einem separatem Projekt abgespeichert.

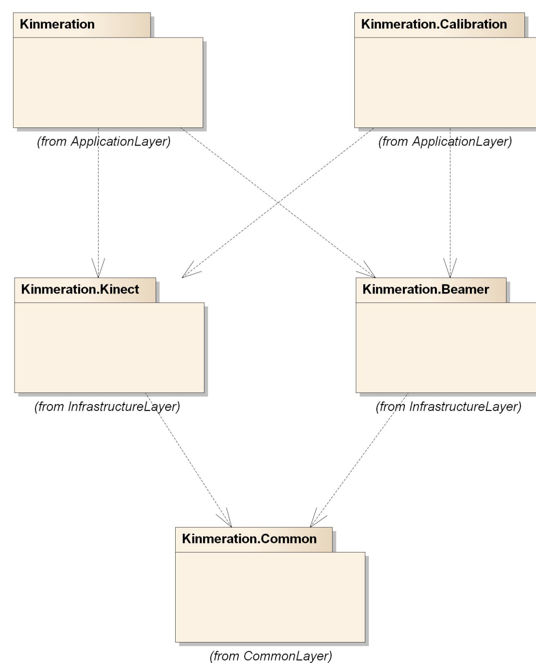


Abbildung 4.2: Komponentenübersicht mit Abhängigkeiten

In der obigen Grafik sind die Abhängikeiten der Komponenten untereinander ersichtlich. Dabei wurde aus Gründen der Übersichtlichkeit darauf verzichtet die transitiven Abhängigkeiten mit einzuzeichnen. Jede Komponente wurde auch einer logischen Schicht zugeordnet.

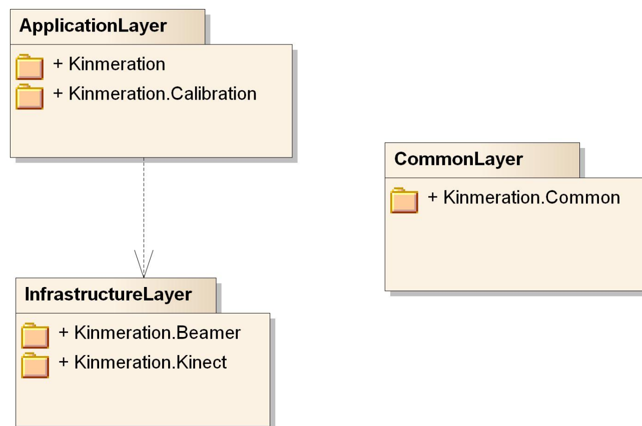


Abbildung 4.3: Komponentenübersicht innerhalb der Layers

4.2.2 Komponente Kinmeration

Die Kinmeration Komponente bietet high level Services, wie beispielsweise eine Objektdetektierung, an. Sie beinhaltet auch die Abstraktion der Spielfläche.

Schnittstellen

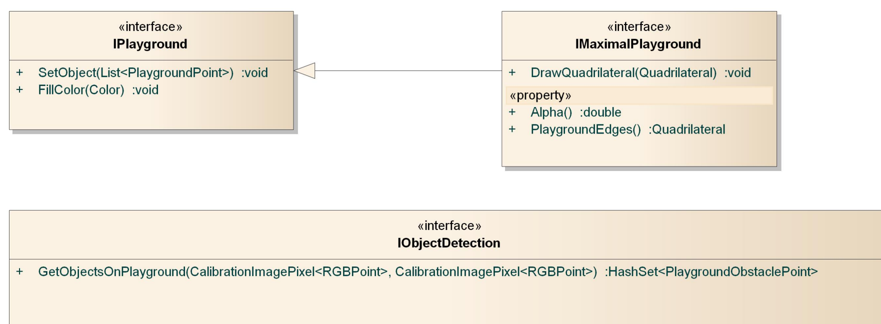


Abbildung 4.4: Schnittstellen der Komponente Kinmeration

`IPlayground` ist die Basisschnittstelle einer Playground. Sie enthält die grundlegendsten Operationen, welche alle Playground Typen beherrschen müssen.

`IMaximalPlayground` ist die Schnittstelle der Standardimplementation. Sie beinhaltet insbesondere zusätzliche Operationen um zu Zeichnen.

`IObjectDetection` beschreibt Operationen für die Objektdetektion.

ObjectDetection

Abbildung 4.5: Klassen der Objektdetektierung

Die Klasse `ObjectDetectionBase` ist die Basisimplementation der Objektdetektion. Sie beinhaltet das 1:1 Mapping von Area zu Playground. Die Methode `GetobjectOnPlayground` führt die eigentliche Objektdetektion basierend auf zwei ColorImages aus. Im Hintergrund wird ein Differenzbild generiert. Die Pixel welche als Differenz erkannt wurden, werden mit Hilfe des Mappings identifiziert und in ein HashSet gespeichert. Eine HashSet Datenstruktur wurde verwendet um die Redundanz zu veringern.

Die Klasse `ObjectDetectionFilled` baut auf der Klasse `ObjectDetectionBase` auf. Sie verdichtet aber das Resultat der Objektdetektion zusätzlich. Dabei wird anhand von zwei Korrekturparametern weitere Punkte in der Playground hinzugenommen.

Das Mapping der Area Koordinatensytems und des Playground Koordinatensystems wird in eine Factory ausgelagert.

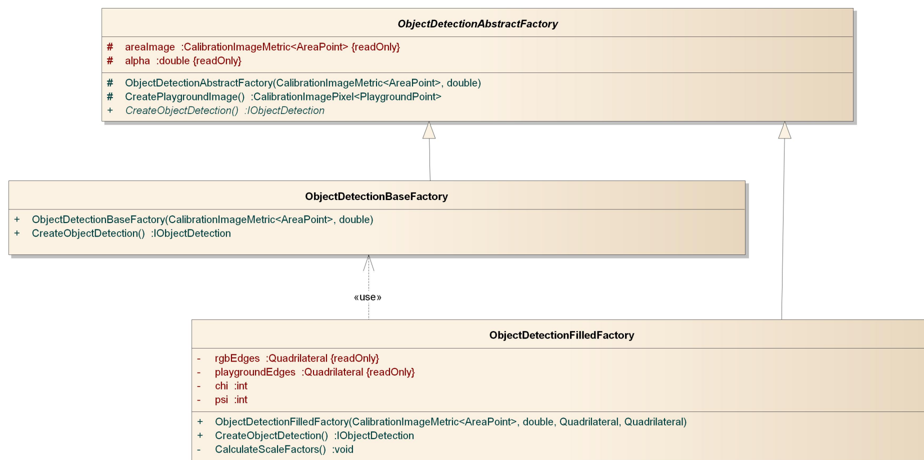


Abbildung 4.6: Klassen zur Erstellung der Objektdetektierung

Die Abstrakte Klasse `ObjectDetectionAbstractFactory` realisiert ein 1:1 Mapping der Koordinatensyteme von Area zu Playground.

Die Klasse `ObjectDetectionAbstractFactory` nutzt dieses Mapping und erzeugt eine `ObjectDetectionBase` Instanz.

`ObjectDetectionFilledFactory` baut auf der Klasse `ObjectDetectionAbstractFactory` auf, aber berechnet die Verdichtungsvariablen dynamisch um das Resultat optimal zu verdichten.

Nachfolgend wird das Zusammenspiel der Factory Klassen bei der Erstellung einer `ObjectDetectionBase` Instanz illustriert.

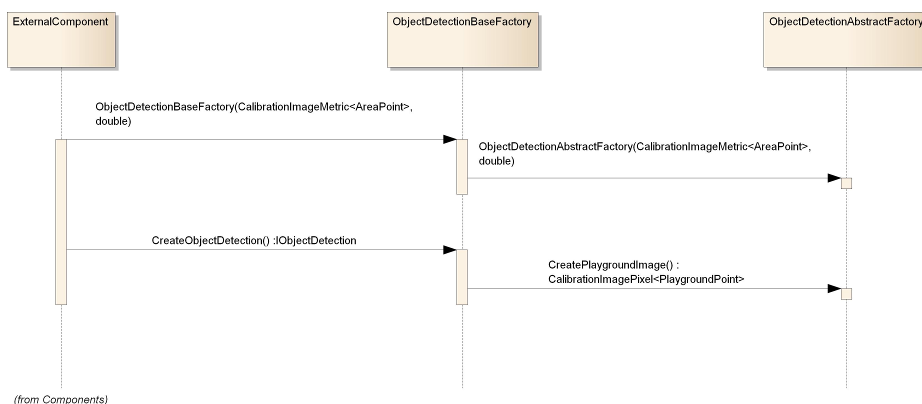


Abbildung 4.7: Erstellung einer Objektdetektionsinstanz

Parameterliste der Objektdetektierung

Eine wichtige Rolle spielen die Parameter bei der Erstellung einer Objektdetektierungsinstanz. Da diese Klassen auf der Kalibration und der Playground aufbauen, sollten die Parameter die gleichen sein wie die, welche in der Playground genutzt werden.

Parameter Typ	Beschreibung	Informationsursprung
`Calibration ImageMetric <AreaPoint>`	Ein Area Image, welches bei der Kalibration entsteht	`Calibration Result`
`double`	Skalierungsfaktor der Playground	`Playground`
`Quadrilateral`	Polygon der Eckpunkte in RGB Koordinaten	`Calibration Result`
`Quadrilateral`	Polygon der Eckpunkte in Beamer Koordinaten	`Calibration Result`

Tabelle 4.1: Parameter Objektdetektierung

Playground

Aufgrund der Komplexität wird Erstellung und Repräsentation eines Playground Objektes ebenfalls mit dem Factory Pattern, welches im Buch [EG94] beschrieben ist, getrennt.

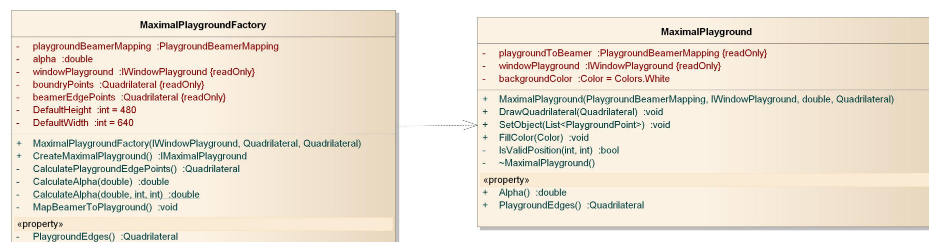


Abbildung 4.8: Playground Klassen

Die Klasse `MaximalPlayground` bildet die Grundlage für alle weiteren Playground Typen. Sie unterscheidet sich dadurch, dass sie den Beamer steuern kann und somit für das Zeichnen zuständig ist. Um das zu realisieren kann sie auf ein vorab berechnetes Mapping zurückgreifen.

Die Factory Klasse `MaximalPlaygroundFactory` erstellt die Klasse `MaximalPlayground`. Sie realisiert dabei das Mapping von Beamer Koordinaten in die Playground Koordinaten. Im Hintergrund wird ein `PlaygroundPoint` mittels Baryzentrischem Algorithmus einem `BeamerPoint` zugeordnet.

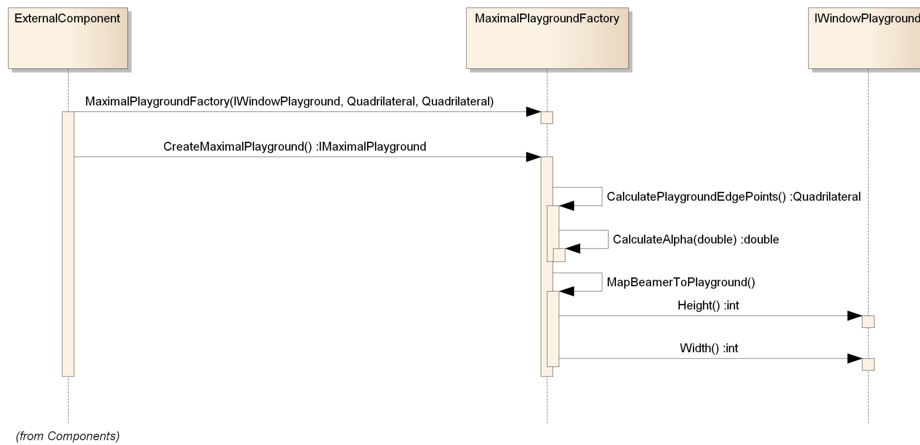


Abbildung 4.9: Erstellung einer Spielfläche

In einem ersten Schritt wird die Factory Klasse parametrisiert. Das Mapping wird erst beim Erzeugen der konkreten Klasse `MaximalPlayground` erstellt und der Klasse als Parameter im Konstruktor überreicht.

Parameter Typ	Beschreibung	Klasse
`IWindow Playground`	Eine Window Instanz der Beamer Komponente	`Window Factory`
`Quadrilateral`	Eckpunkte der Area	`Calibration Result`
`Quadrilateral`	Eckpunkte in Beamer Koordinaten	`Calibration Result`

Tabelle 4.2: Parameter Playground

Wird eine Objektdetektion für diese Playground benötigt muss darauf geachtet werden, dass als Area Repräsentation die gleiche genommen wird wie für die Objektdetektierung.

4.2.3 Komponente Kinmeration.Calibration

Der ganze Kalibrationsprozess wird von dieser Komponente gesteuert und verwaltet.

Schnittstellen
Die Schnittstellen für externe Komponenten beschreiben Daten-Objekte, welche aus einer Kalibration hervorgehen.

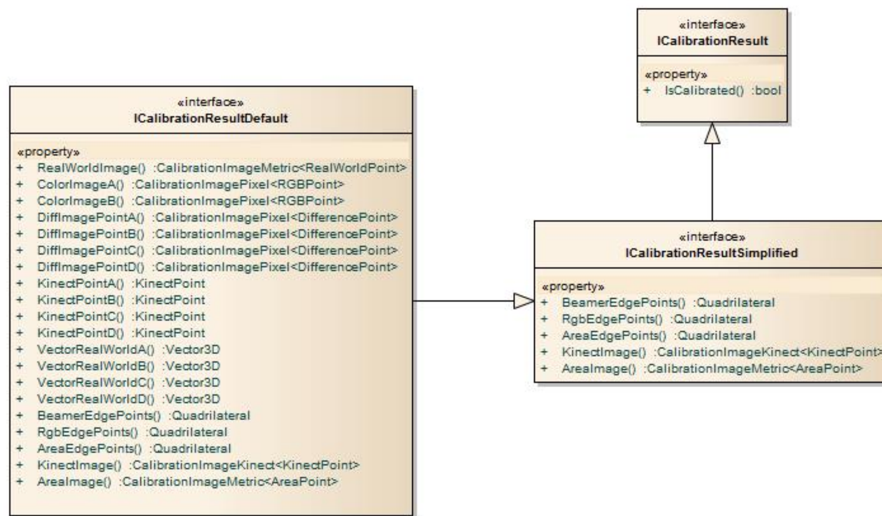


Abbildung 4.10: Schnittstellen der Komponente Kinmeration

Kalibrierung

Der ganze Kalibrierungsprozess wird innerhalb einer Factory ausgeführt. Die einzelnen Kalibrierungsschritte werden innerhalb der Klasse über private Methoden aufgerufen. Ergebnis eines einzelnen Schrittes wird in einer Field Variable zwischengespeichert. Diese Fields bilden am Schluss die Grundlage um eine `CalibrationResult` Daten-Objekt Instanz zu erstellen.

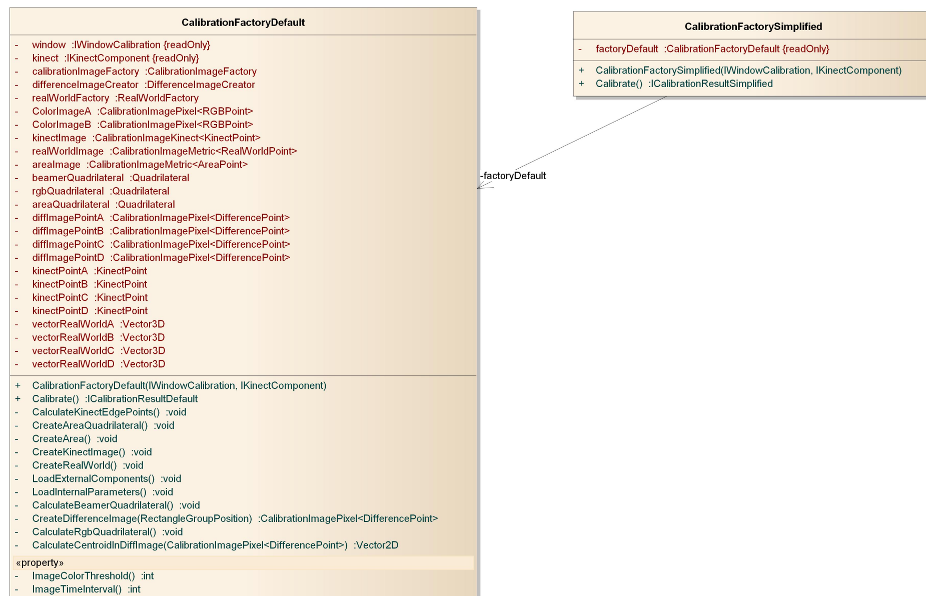


Abbildung 4.11: Kalibrationsprozess innerhalb einer Factory

Das Resultat einer Kalibrierung ist ein Daten-Objekt. Die Klasse `CalibrationResultSimplified` enthält lediglich die wichtigsten Kalibrationsresulate, wie beispielsweise das `AreaImage`. Die Klasse `CalibrationResultDefault` erweitert die vereinfachte Klasse um das Resultat jedes einzelnen Kalibrationsschritt. Mit einer Instanz dieser Klasse lässt sich beispielsweise überprüfen, was nach jedem Kalibrationsschritt gemacht wurde.

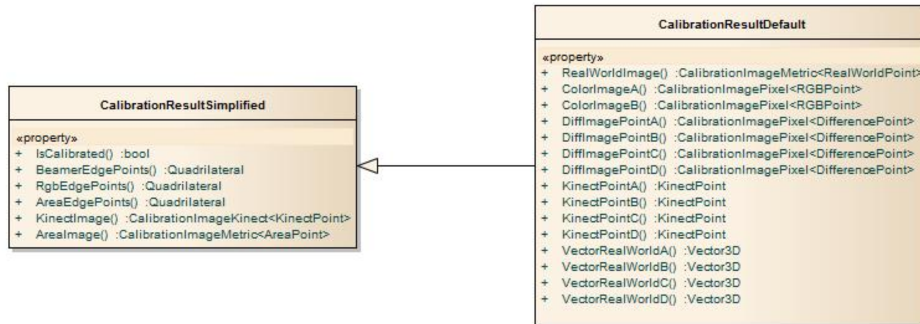


Abbildung 4.12: Resultat einer Kalibration

Nachfolgend wird ein Kalibrierungsprozess dargestellt.

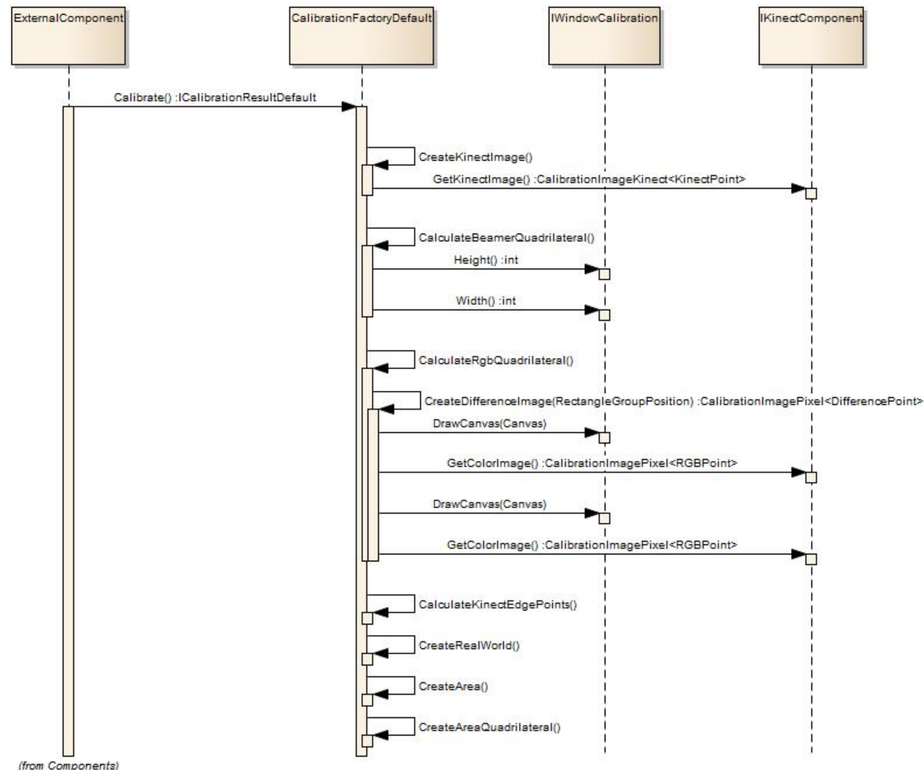


Abbildung 4.13: Kalibrationsprozess

Der Factory Klasse werden die Komponenten aus dem Infrastructure Layer übergeben. Über diese Komponeten kann auf Daten der Infrastructure Schicht zugegriffen oder das Beamerfenster angesteuert werden. Danach kann jeder einzelne Kalibrationsschritt sequenziell abgearbeitet werden. Als Beispiel muss für den Differenzbild-Algorithmus der zeitlichen Ablauf der Aktionen ”Bild anzeigen” und ”Bild aufnehmen” gesteuert werden.

4.2.4 Komponente Kinmeration.Beamer

Um die Beameransteuerung von den restlichen Programmteilen zu entkoppeln, wurde diese in eine separate Komponente ausgelagert. Grundsätzlich wird der Beamer als erweiterter Desktop dem Hauptdesktop zugeschaltet. Das heisst die Ansteuerung des Beamers beschränkt sich darauf den erweiterten Desktop anzusteuern. Da eine Treiberimplementation den Rahmen der Bachelorarbeit sprengen würde, wurde auf einem höheren Schicht angesetzt. Es reicht, wenn ein Fenster randlos und maximiert den erweiterten Desktop ausfüllt.

Schnittstellen
Die Schnittstellen definieren die Funktionalitäten der randlosen Fenster des erweiterten Desktops.

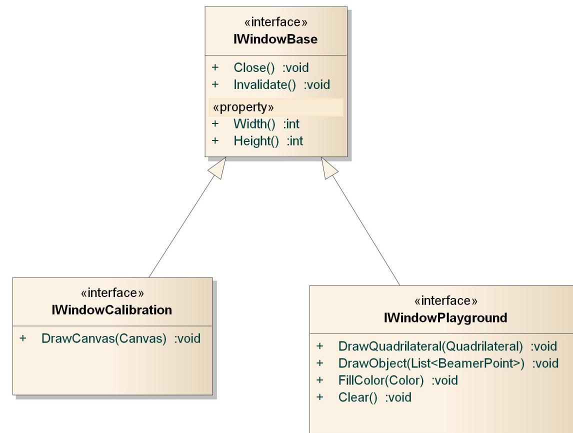


Abbildung 4.14: Interfaces der Beamer Komponente

Die Basisschnittstelle `IWindowBase` enthält alle grundlegenden Operation die ein Windows Objekt beschreiben. Aus Kompatibilitätsgründen zur Implementation in der Studienarbeit, wurde das `IWindowCalibration` erstellt. Mit der Methode `DrawCanvas(Canvas)` lässt sich ein WPF Canvas zeichnen. [The13] [Gei13]
Da die Playground mit einem Bitmap arbeitet, sind andere Operationen notwendig. Diese sind in der Schnittstelle `IWindowPlayground` enthalten.

Das Systemsequenzdiagramm in Abbildung 4.15 zeigt den Ablauf bei der Erstellung einer WindowCalibration Instanz.

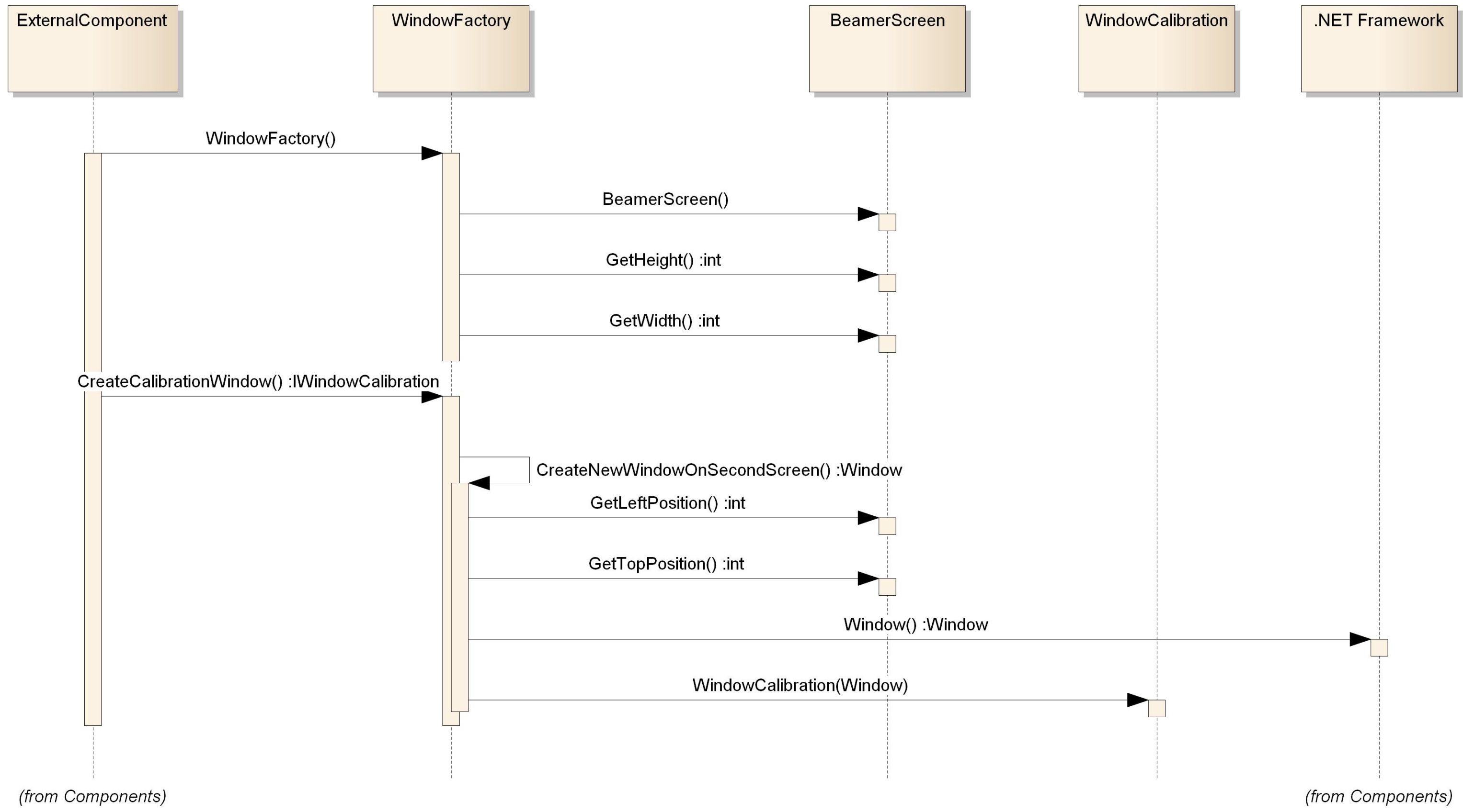


Abbildung 4.15: Erstellen einer WindowCalibration Instanz

Die Factory Klasse muss zuerst die Dimensionen des Haupt- und erweiterten Desktops kennen. Diese kann er von der Klasse `BeamerScreen` abrufen. Da WPF lediglich einen virtuellen Bildschirm kennt, musste diese Klasse die Informationen aus den WinForms Klassen des .NET Frameworks holen. Danach muss lediglich ein neues WPF Fenster erzeugt und mit den richtigen Dimensionen so platziert werden, dass es den erweiterten Desktop aufüllt.

4.2.5 Komponente Kinmeration.Kinect

Um die Kinect anzusteuern, wird die Kinect SDK von Microsoft benötigt. Die gelieferten Rohdaten der Kinect erwiesen sich als unhandlich um damit weiterzuarbeiten. Die Kinect Komponente kapselt die direkten Zugriffe auf die Microsoft Kinect SDK und bereitet die Daten auf. Als Ergebnis werden Image Objekte generiert, welche weiterverarbeitet werden können.

Schnittstellen

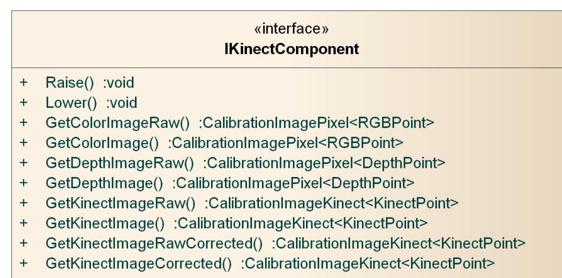


Abbildung 4.16: Kinect Interface

Die Schnittstelle beschreibt, welche Services diese Komponente bietet. Ein Grossteil der Methoden rufen Daten ab. Aber es muss auch möglich sein, die Kinect nach oben oder nach unten zu bewegen.

Basisimplementation

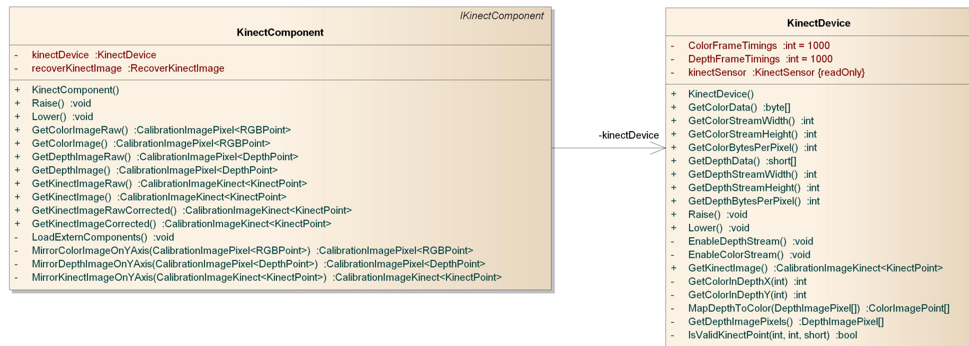


Abbildung 4.17: Kinect Klassen

Die Klasse `KinectDevice` ist das Bindeglied zwischen Kinmeration und der Kinect SDK von Microsoft. Sie stellt die Konnektivität zur Kinect Hardware her und bietet low level Services an um die Bitströme abzugreifen.
Aufbauend auf der Klasse `KinectDevice` gibt es die Klasse `KinectComponent`, welche die oben beschriebene Schnittstelle implementiert. Sie leitet Hardware Aufrufe an die `KinectDevice` weiter und wandelt die Bitströme in Image Objekt um.

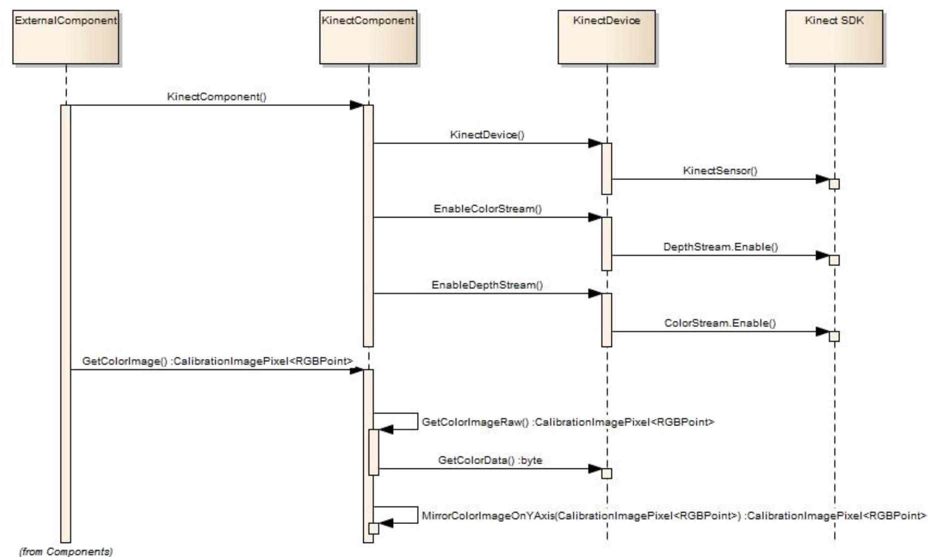


Abbildung 4.18: Erstellen einer KinectComponent Instanz und Abfragen von Color Daten

Der Client erstellt die Komponente über den Standardkonstruktor. Innerhalb dieses Konstruktors werden die restlichen Klassen erstellt und die Kinect SDK aufgerufen.
Ein Methodenaufruf auf die erstellte Instanz, behandelt im Hintergrund den Aufruf auf die Komponente der Kinect SDK. Des Weiteren werden die Daten der SDK in interne Datenstrukturen verpackt.

4.2.6 Komponente Kinmeration.Common

Sämtliche Datenstrukturen, welche im ganzen Projekt genutzt werden, werden in dieser Komponente gehalten. Auch Algorithmen, welche für andere Projekte nützlich sind, befinden sich in dieser Komponente.

Datenstrukturen

Für die Umsetzung des Image-Konzeptes, wird eine eigene Datenstruktur erstellt. Die Basis der Struktur ist ein zwei dimensionales Array.

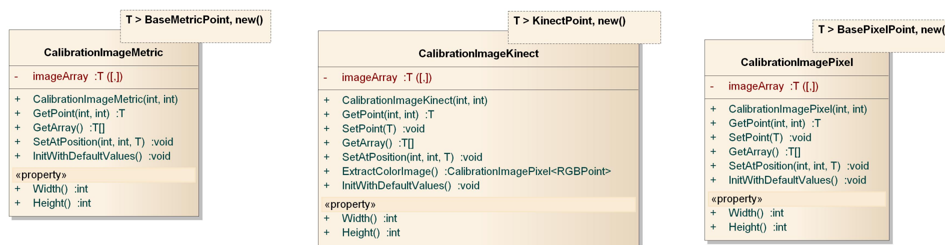


Abbildung 4.19: Image Datenstrukturen

Aus Gründen der Übersicht, wurden die Datenstrukturen je nach Einsatzzweck typisiert. Für ein metrisches Koordinatensystem wurde eine andere Datenstrukur verwendet als für Pixel basierte Koordinatensysteme. Mittels Generics konnte trotzdem auf Codeduplizierung verzichtet werden.

Um Punkte in verschiedenen Koordinatensystemen zu repräsentieren, sind wiederum weitere Datenstrukturen entstanden.

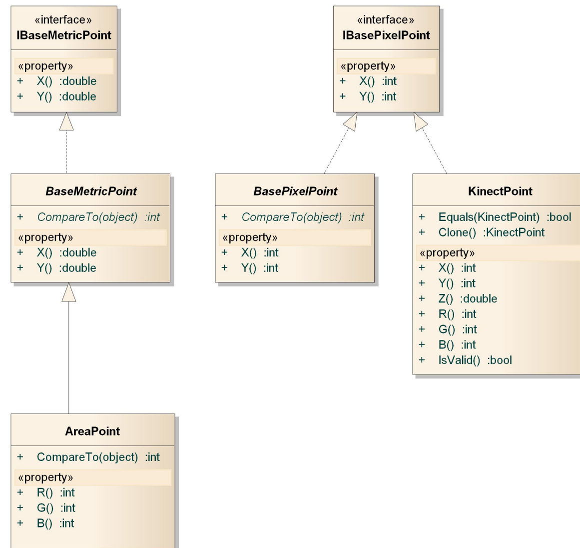


Abbildung 4.20: Punkte Datenstrukturen

Um Redundanzen zu vermeiden, wurde auf eine Vererbungshirarchie zurückgegeriffen. Da es sich lediglich um Datenklassen handelt, ist mit keinen Problemen zu rechnen.

Die Pixel basierten Punkte werden noch weiter verfeinert:

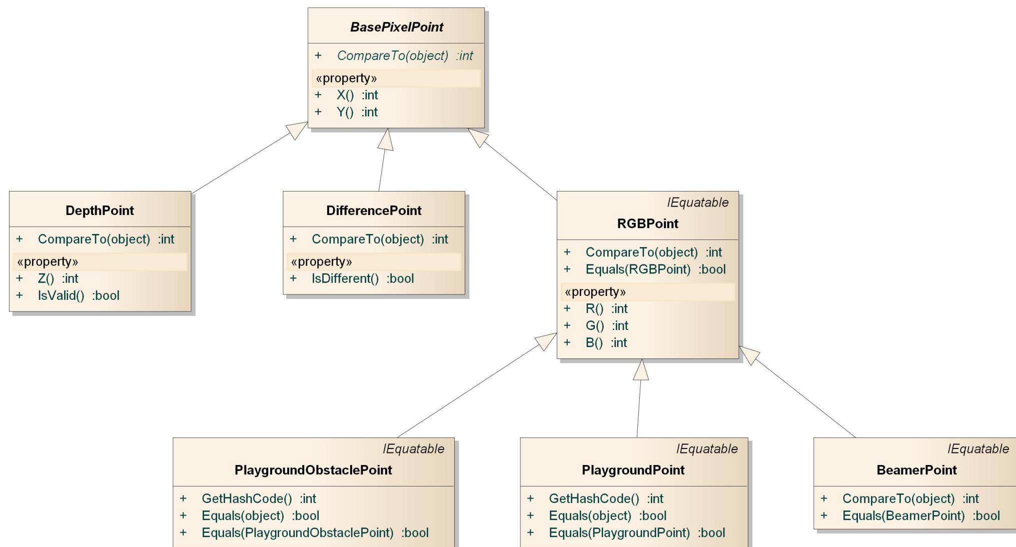


Abbildung 4.21: Pixel Punkte Datenstrukturen

Die Anzahl an Punktklassen ist unübersichtlich. Aber es kann somit sichergestellt werden, dass in einem Algorithmus immer mit kompatiblen Punkten aus dem selben Koordinatensystemraum gerechnet wird.

Um mathematische Konzepte abzubilden, gibt es die beiden Klassen `Vector2D` und `Vector3D`. Sie unterstützen die üblichen Vektoroperationen. Zudem bilden sie die Basis für allgemeine Algorithmen, welche somit portabel sind.

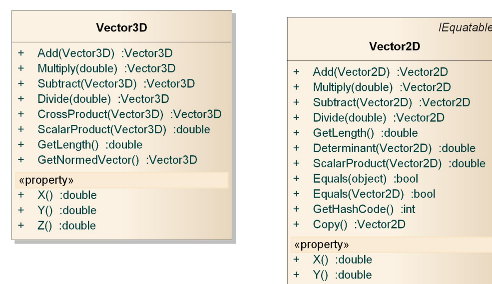


Abbildung 4.22: Vektorklassen

Die Repräsentation eines beliebigen Vierecks wird in einer Klasse realisiert.

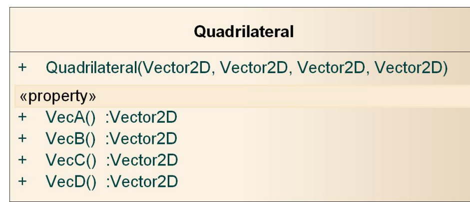


Abbildung 4.23: Vierecksklasse

Da diese Klasse auf Vektoren basiert, ist sie vielseitig einsetzbar.

4.2.7 Exception Handling

Innerhalb der Bibliothek können zwei Arten von Laufzeitfehlern entstehen.
Wird innerhalb einer Komponente auf eine andere Komponente zugegriffen, kann diese einen Laufzeitfehler auslösen. Diese Laufzeitfehler können innerhalb eines try/catch-Block behandelt werden. Tritt dieser Fall auf, wird die Exception in eine eigene Exceptionklasse `KinmerationException` verpackt. Als Debug Information wird die Call Methode und die entsprechende Komponente mitgeführt.
Diverse Validierungschecks können ebenfalls Laufzeitfehler auslösen. Diese werden ebenfalls in die Klasse `KinmerationException` gepackt.
In beiden Fällen, wird die Exception an die nächst höhere Schicht weitergeleitet, bis zum Client. Der Client kann dann entscheiden, was in einem konkreten Fall gemacht werden sollte.

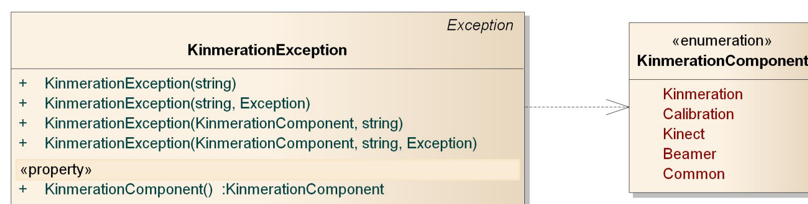


Abbildung 4.24: Exceptionklasse von Kinmeration

4.2.8 Tests

Für die Qualitätssicherung wurden automatisierte Unit Tests erstellt. Dabei wurde für jede Komponente eine separates Test Projekt erstellt.

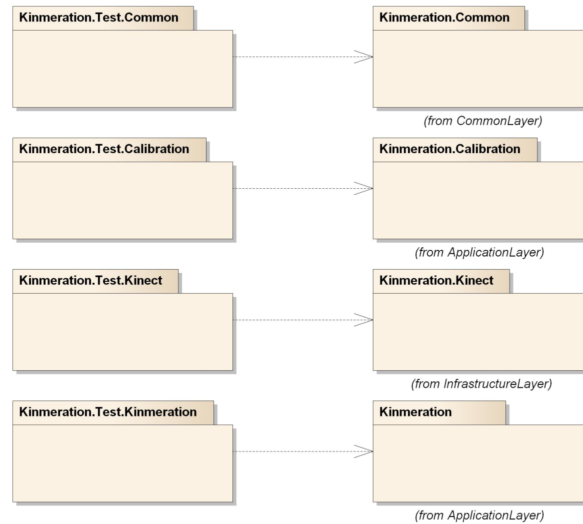


Abbildung 4.25: Testprojekte und Abängigkeiten zu den Komponenten

Da die Unit Tests nich alle Facetten abdecken konnten, wurden zusätlich Integrationstest ausgeführt. Diese Tests umfassten einen kompletten Durchlauf aller Kalibrationsschritte. Um die Resultate der einzelnen Schritte zu visualisieren und diese auch zu verifizieren, wurde ein separates Debug Projekt erstellt.

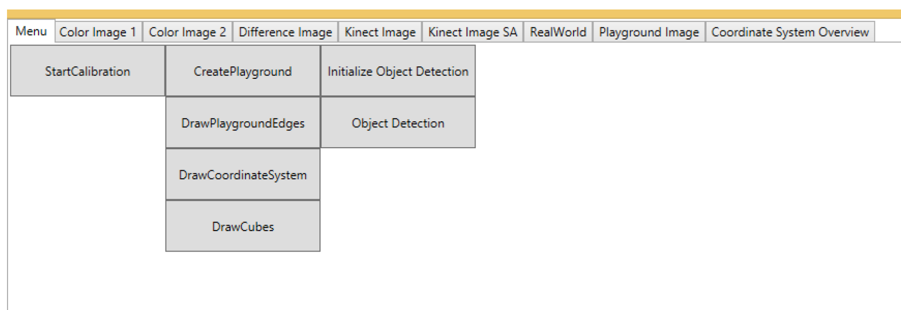


Abbildung 4.26: Oberfläche der Debug Applikation

Das Debug Projekt diente lediglich dazu das Finden von Fehlern zu vereinfachen. Die Integrationstests wurden immer dann ausgeführt, wenn eine Funktionalität von "Kinmeration" als abgeschlossen galt. Da mit Git ein Workflow mit mehreren Branches eingesetzt wurde, erfolgte bei einem Mergevorgang in den master Branch ein manueller vollumfänglicher Integrationstest.

4.3 aMAZEingMouse

4.3.1 Klassenübersicht

Die Klassen von ”aMAZEingMouse” können in drei Kategorien eingeteilt werden. Nachfolgend sind die Verantwortlichkeiten der einzelnen Klassen beschrieben.

4.3.1.1 Spiellogik

Die Spiellogik stellt im Wesentlichen die Stabilität und Funktionsweise des Spiels sicher. Die Klassen dieser Kategorie sind generell für die Kontrolle und Überwachung des Spiels zuständig.

Klasse	**Beschreibung**
`GameEngine`	Sie bildet die Hauptklasse von ”aMAZEingMouse” und ist somit für den ganzen Ablauf des Spiels verantwortlich. Jegliche Aufrufe laufen über die GameEngine. Sie weiss zu jeder Zeit in welchem Zustand sich das Spiel befindet und was zu tun ist.
`Mouse`	Auch sie ist eine wesentliche Klasse des Spiels. Die virtuelle Maus wird durch sie repräsentiert. Funktionen, um die Maus steuern zu können, werden von ihr angeboten.
`Exception Handling`	Sie stellt eine grundlegende Behandlung aller Fehler, sowohl von Benutzer als auch von externen Calls auf die ”Kinmeration”-Bibliothek, sicher. Sie teilt dem Benutzer mit, was er falsch gemacht hat und was er tun muss. Somit sorgt sie für einen stabilen Programmablauf.

Tabelle 4.3: aMAZEingMouse: Klassen - Spiellogik

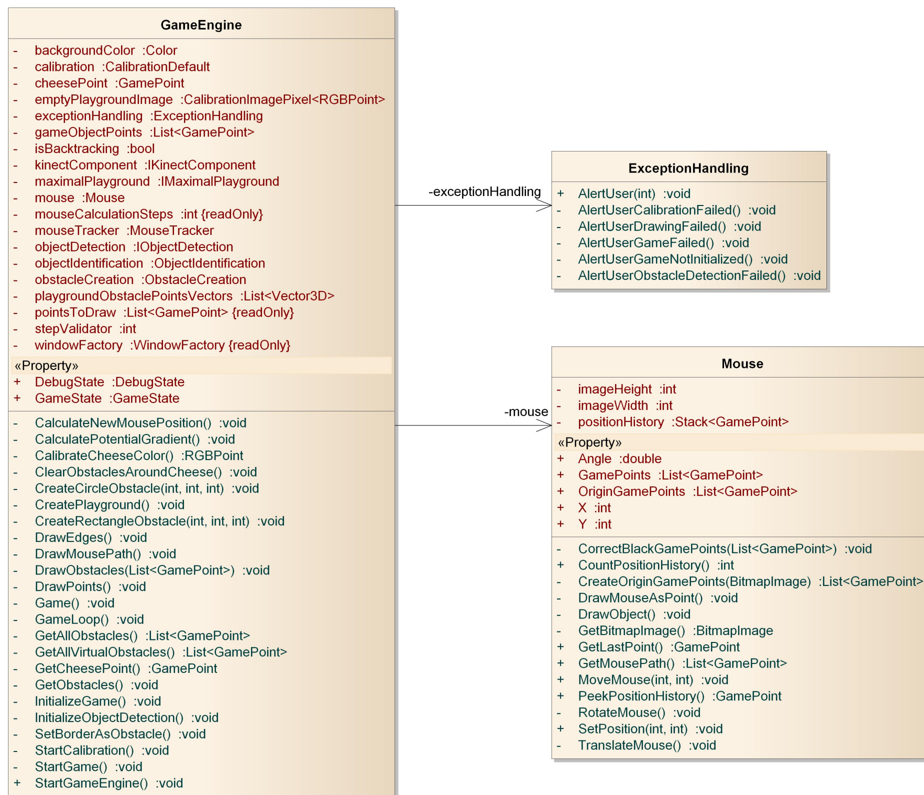


Abbildung 4.27: aMAZEingMouse - Spiellogik

4.3.1.2 Datenstrukturen

Das Spiel benötigt eigene Datenstrukturen, um unter anderem Spielobjekte oder Spielzustände abstrahieren zu können.

Klasse	Beschreibung
`GamePoint`	Ein GamePoint repräsentiert ein einziger Punkt innerhalb des Spielfeldes. Er kann verschiedene Spielobjekte, wie z.B. Hindernisse, Käse etc., repräsentieren. Alle Algorithmen von "aMAZEingMouse" benötigen GamePoints für ihre Berechnungen.
`GameState` (enum)	Wie der Name schon vermuten lässt, steht der GameState für den aktuellen Zustand des Spiels. Es gibt unterschiedliche GameStates, welche das Verhalten der GameEngine steuern können. Dadurch werden die Aktionen ausgeführt, welche für den aktuellen Status notwendig sind. Der GameState bezieht sich immer auf den Status des Spiel-Modus.
`DebugState` (enum)	Der DebugState hat praktisch dieselben Aufgaben wie der GameState mit dem Unterschied, dass er sich immer auf den Status des Demo-Modus bezieht. Somit kann er von der GameEngine verlangen, dass diese die von ihm gewünschten Demo-Funktionalitäten ausführt.
`MouseTracker`	Der MouseTracker ist ein Abbild der gesamten Spielfeld-Karte. Er hat eine sehr wichtige Aufgabe, denn er ist beim Backtracking der Maus in Klemmsituationen gefragt. Jeder Schritt der Maus wird in ihm abgelegt. So kann er jederzeit Auskunft geben, ob die Maus bereits einmal an einer bestimmten Position war und dadurch ein allfälliges Backtracking einleiten.

Tabelle 4.4: aMAZEingMouse: Klassen - Datenstrukturen

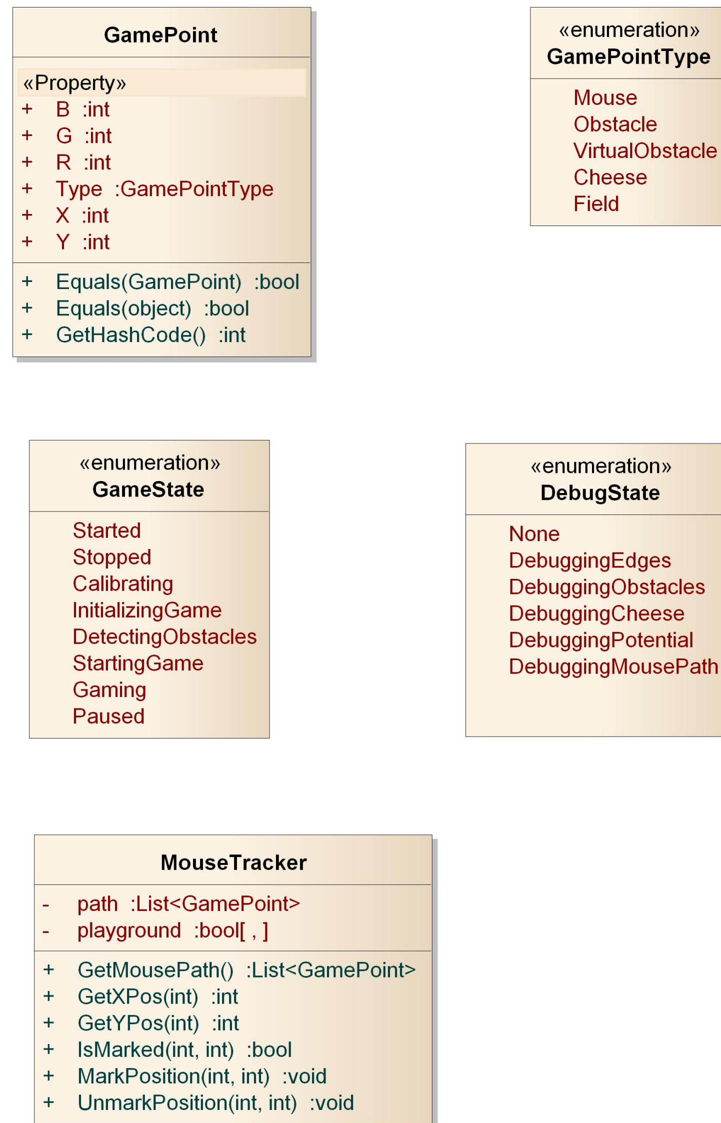


Abbildung 4.28: aMAZEingMouse - Datenstrukturen

4.3.1.3 Algorithmen

Die Klassen dieser Kategorie sind für das Spiel äusserst wichtig, da diese einerseits Objekte erkennen und andererseits der Maus sagen, wohin sie sich bewegen muss.

Klasse	Beschreibung
`Potential Calculation`	Diese Klasse implementiert die mathematischen Funktionen, welche für die Berechnung des Weges notwendig sind. Alle Formeln, welche mit der Käsesuche zu tun haben, befinden sich in dieser Klasse.
`MoveMouse DirectionVector` (obsolet)	Sie ist im aktuellen Release obsolet. Sie war zu Beginn der Spiel-Implementation nötig um Schritte der Maus zum Käse zu berechnen. Jedoch kann sie keine Hindernisse in ihre Wegsuche miteinbeziehen.
`MoveMouse PotentialBasic`	Sie ist die Klasse, welche aktuell zur Berechnung des nächsten Schrittes der Maus notwendig ist. Sie verwendet die mathematischen Funktionen der Klasse `PotentialCalculation`.
`Object Identification`	Die Klasse ObjectIdentification ist insofern wichtig, dass sie erkannte Objekte zusätzlich als Spielobjekte identifizieren kann. Sie entscheidet, ob es sich bei erkannten Objekten um Hindernisse oder Käse handelt.

Tabelle 4.5: aMAZEingMouse: Klassen - Algorithmen

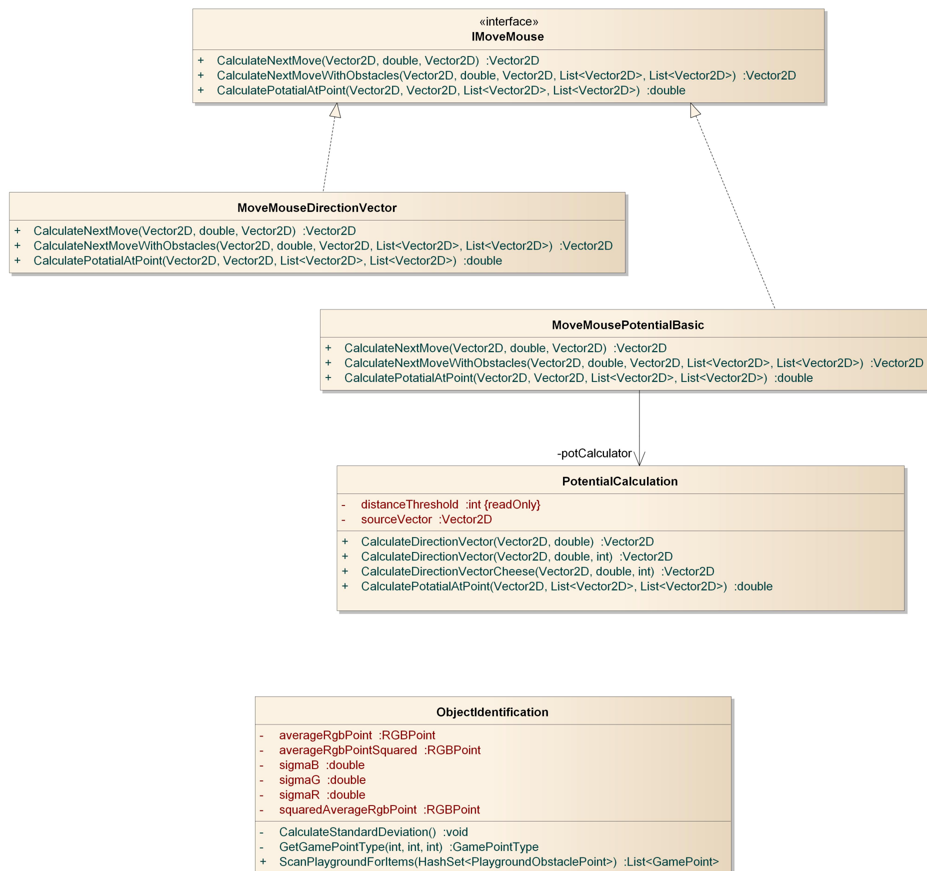


Abbildung 4.29: aMAZEingMouse - Algorithmen

Kapitel 5

Schlussfolgerung und Ausblick

5.1 Kinmeration

Die ”Kinmeration”-Bibliothek bietet für einen Entwickler die Möglichkeit Applikationen und Spiele in einer Kinect-Beamer-Umgebung zu realisieren. Dabei muss er sich nicht mit der Kalibrationsproblematik der unterschiedlichen Koordinatensysteme, welche Kinect, Beamer und die reale Welt umfassen, beschäftigen. Die Spielfläche wird ihm als einheitliches kartesisches Koordinatensystem zur Verfügung gestellt. Ausserdem bietet Kinmeration mehrere high-level Services, wie beispielsweise Objektdetektierung, an.

Die Architektur der ”Kinmeration”-Bibliothek ermöglicht Erweiterungen auf unterschiedlichen Ebenen. Sowohl ein Spielentwickler als auch ein Entwickler, welcher bestehende Komponenten abändern möchte, können mit Hilfe der Dokumentation Anpassungen vornehmen.

5.1.1 Beamerprojektion

Eine grundlegende Thematik ist die Projektion des Beamerbildes. Dabei gibt es zwei Aspekte, die für uns von grosser Bedeutung sind. Einerseits ist es wichtig den Beamer stabil und optimal ausrichten zu können. Andererseits sollten wir wissen, mit welchen Möglichkeiten von geometrischen Formen des projizierten Beamerbildes wir umgehen können müssen (siehe Anhang B) . Letzterer ist mitunter einer der wichtigsten Punkte, welcher uns Aufschlüsse für die benötigten mathematische Berechnungen liefert.

Für Test- und Demozwecke war es hilfreich, wenn wir den Beamer mit Hilfe einer provisorischen Halterung in eine stabile Lage bringen können. Die Stabilität der Halterung ist insofern wichtig, da das Beamerbild während der Kalibration nicht bewegt werden darf. Bereits während der Studienarbeit haben wir mit Alltagsge-

genständen (Bücher, Ordner etc.) Halterungen zusammengestellt. Im Rahmen der Bachelorarbeit haben wir diese optimiert und flexibler gestaltet. Das Bauen einer professionellen Halterung in Zusammenarbeit mit anderen Studenten der HSR (z.B. Maschinenbau) wäre denkbar und für einige Einsatzzwecke, wie z.B. "Informatik zum Anfassen", erwünscht. Dies hätte jedoch den Rahmen unserer Bachelorarbeit sprengen.

Die schwierigste Problematik bringt das Projizieren eines Beamers, welcher sich in Schräglage befindet, mit sich. Nicht nur die unterschiedlichen konvexen Formen, welche die Projektion aufweisen können, sondern auch die unterschiedliche Grösse der Pixel müssen beachtet werden. Dabei fällt auf, dass weiter entfernte Beamerpixel bei schrägen Projektionen im Vergleich zu den vorderen um Einiges grösser sind. Dies führt auch dazu, dass das Beamerbild gegen hinten eher unscharf wird. Diese Problematik macht sich vor allem beim Zeichnen von erkannten Objekten bemerkbar, da weit entfernte Pixel durch ihre Grösse mehrere erkannte Kinect-Pixel decken können. Dadurch wird ein erkanntes Objekt beim Zeichnen viel länger als es eigentlich ist.

Mit den unterschiedlichen Formen der Projektion kann die "Kinmeration"-Bibliothek umgehen, da wir jeweils die vier Eckpunkte in allen Koordinatenstemen kennen. Die Kalibration kümmert sich basierend auf diesen Eckpunkten mit Hilfe der implementierten Algorithmen um die Berechnung aller weiteren Punkte.

Die zweite Problematik, die variablen Pixelgrössen des Beamers, konnte aufgrund des Zeitbudgets nicht behandelt werden. Sinnvollerweise sollten nachfolgende Arbeiten dieses Thema aufgreifen, damit die Bibliothek für jedes Kinect-Beamer-Setup funktioniert, auch mit Projektionen auf den Boden. Momentan sind wir auf Beamerbilder, welche an die Wand projiziert werden, eingeschränkt. Leichte Verzerrungen sind möglich, aber nicht immer optimal.

5.1.2 Varianten von Playgrounds

In der Entwurfsphase haben wir von unterschiedlichen Formen und Abmessungen, welche eine Playground aufweisen kann, gesprochen. Dabei war die `MaximalPlayground` der Ursprung aller weiteren Playgrounds. Unser aktueller Release liefert lediglich diese umfassende Playground. Die Implementation der anderen Playgrounds bringen zusätzliche mathematische Probleme, für welche leider keine Zeit übrig blieb.

Die Implementation der verbliebenen Playgrounds sind in nachfolgenden Arbeiten gut denkbar. Die "Kinmeration"-Bibliothek ist flexibel genug, sich mit weiteren Playgrounds erweitern zu lassen. Die mathematischen Probleme, welche sich hinter den Erweiterungen verbergen, handeln unter anderem vom Finden eines mög-

lichst grossen Rechtecks innerhalb eines beliebigen Vierecks. Auch das Suchen von Kreisflächen müsste gelöst werden.

5.2 aMAZEingMouse

5.2.1 Übersicht

Die Umsetzung von ”aMAZEingMouse” stützt sich auf dem Konzept, welches wir in der Anforderungsanalyse erarbeitet haben. Dabei stand der minimale Modus mit beliebigen Hindernissen sowie der Demo-Modus im Fokus, da die Verbesserung der ”Kinmeration”-Bibliothek etwas länger dauerte als geplant. Für uns war es aber wichtig, dass wir der Bibliothek vertrauen können, sobald wir mit der Spielimplementierung beginnen. Deshalb haben wir mehr Zeit in die Bibliothek investiert, um bestehende Algorithmen zu verbessern. Da für das Spiel weniger Zeit als geplant zur Verfügung stand, mussten wir die Entwicklung sehr einfach halten. Verbesserungen des Codes, Architektur und der Algorithmen mussten teilweise aufgrund nicht vorhandener Zeit zurückgestellt werden. Auch das Entwerfen einer ansprechenden Benutzeroberfläche musste ein wenig vernachlässigt werden.

Für uns lieferte das Spiel nebst einem Ausstellungsobjekt für ”Informatik zum Anfassen” gerade auch den Proof of Concept für unsere Bibliothek. Mit Hilfe von ”aMAZEingMouse” konnten wir zeigen, dass unsere Bibliothek tatsächlich von Spielentwicklern genutzt werden kann.

Das nachfolgende Bild zeigt ”aMAZEingMouse” in Aktion.

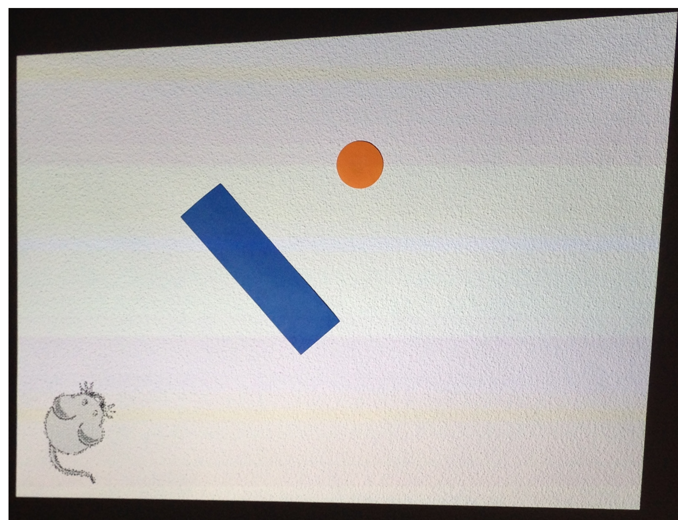


Abbildung 5.1: aMAZEingMouse - Spielverlauf

5.2.2 Bewertung und Fazit

Uns ist durchaus bewusst, dass es für die Problematik der optimalen Wegfindung bessere Algorithmen gibt als unser physikalisches Modell. Für diese Aufgabe würde sich auch der A* oder der D* Algorithmus sehr gut eignen. Da wir aber eine möglichst natürliche und realistische Wegsuche der Maus simulieren wollten, bot sich der physikalische Ansatz besser an. Die Maus muss nicht auf Anhieb den besten Weg finden, sondern darf für kurze Zeit die Orientierung verlieren. Dadurch wirkt die Suche wesentlich authentischer.

Die Qualität unserer aktuellen Implementation des Suchalgorithmuses ist insofern akzeptabel, dass die Maus in den meisten Fällen den Käse finden kann, wenn auch teilweise sehr langsam und mit einem suboptimalen Weg. Vor allem Optimierungen betreffend der Suchzeit sind nötig. Es wurden bereits einige Ansätze angedacht. Beispielsweise wäre es denkbar zu Beginn der Suche eine ”Karte” der im Spielfeld wirkenden Potentiale zu erstellen. Die Maus würde dann bei jeder Entscheidung, wo sie hinlaufen soll, das tiefste benachbarte Potential auswählen.

Ein weiterer Ansatz zur Optimierung wäre, dass die Maus an ihrer aktuellen Position so weit in alle Richtungen schaut bis sie ein reales oder virtuelles Hindernis wahrnimmt. Anschliessend bestimmt sie in welche Richtung sie gehen muss, um an das tiefste Potential zu gelangen. Zu jenem bestimmten Potential läuft sie dann direkten Weges. Ist sie am Punkt angekommen, so wiederholt sie nun die Schritte ”Umschauen” und ”Bewegen”. Dieser Ansatz sollte die unnötigen, suboptimalen Wege eindämpfen und dadurch die Maus schneller zum Käse befördern.

Aktuell wird der Mittelpunkt der Maus als ein Pixel gesehen. Alle Berechnungen und die darausfolgenden Translationen bzw. Rotationen der Maus gehen von diesem einzelnen Punkt aus. Dies kann unter Umständen dazu führen, dass es so aussieht als würde die Maus Hindernisse leicht betreten. Der Algorithmus funktioniert korrekt, denn der Mittelpunkt kann nicht in ein Hindernis gelangen. Da das Bild der Maus aber deutlich grösser als ein Pixel ist, gibt es teilweise Situationen, in denen es scheint als würden Teile der Maus sich in Hindernissen bewegen. Um dies zu verhindern, wäre es denkbar, dass die Maus in Zukunkt durch mehrere Pixel dargestellt wird.

Die Implementation unseres aktuellen Releases reicht für einfache und einige komplexere Szenarien aus. Dennoch benötigt er einen gewissen Feinschliff, um die Suchzeit minimieren und den Erfolg bei komplexen Labyrinthen immer gewährleisten zu können.