Einsatz von Game Engines zur Erstellung virtueller Testumgebungen für Computer Vision Anwendungen

Fabian Schmidt

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juni 2013

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Hagenberg, am 28. Juni 2013

Fabian Schmidt

Inhaltsverzeichnis

Vorwort

Diese Arbeit sowie das zugehörige Projekt sind im Zuge des Masterstudiums „Interactive Media“ an der Fachhochschule Hagenberg entstanden. Mein Dank gilt an dieser Stelle dem gesamten Kollegium, die dieses Studium ermöglichen, besonders meinem Betreuer Dipl.-Ing. Dr. Wilhelm Burger für seine Ratschläge. Außerdem möchte ich mich bei meinen Eltern und meiner Familie bedanken, die mir die Durchführung dieses Studiums durch ihre Unterstützung ermöglichten.

Kurzfassung

Im Bereich Computer Vision entwickelt sich derzeit sehr viel in den Bereichen automatischer Objekterkennung, Objekttracking oder Bildsegmentierung aus gegebenen 2D Bildern oder Videos. Wenn man solche Algorithmen entwickelt, ist dabei ein wichtiger Schritt, diese zu testen und zu evaluieren. Dazu benötigt man Umgebungen, in denen man den Algorithmus ausführen kann. Ein grundliegendes Problem ist jedoch die Erstellung solcher Umgebungen. Dies kann sehr aufwendig sein, da möglichst viele verschiedene Bedingungen und Situationen abgedeckt werden müssen, also z. B. verschiedene Lichtverhältnisse oder verschiedene Strukturen. Außerdem müssen diese Umgebungen auf den Algorithmus reagieren können, um so das Ergebnis widerzuspiegeln.

Des Weiteren kann es sein, dass man zur Evaluierung eines Algorithmus zusätzliche Metainformationen benötigt, die die Inhalte der Umgebung beschreiben, z. B. an welcher Position welches Objekt liegt. Diese Daten müssen als korrekt, also als Ground Truth, angenommen werden. Wenn man diese Daten manuell erstellen will, ist das meistens sehr zeitaufwendig und umständlich. Daher ist ein weiteres Problem die effiziente Generierung solcher nutzbaren Daten.

Diese Arbeit verfolgt den Ansatz, eine Game Engine zum Testen und Evaluieren von Computer Vision Algorithmen zu verwenden. Diese soll sowohl als virtuelle Testumgebung verwendet werden, als auch sehr schnell vielfältige Ground Truth Daten generieren. Außerdem wird *Virtual Environment Controlling Application* (VECA) vorgestellt – ein Framework, das den Benutzer dabei unterstützt, eine Game Engine als virtuelle Testumgebung aufzusetzen.

Abstract

The computer vision area is currently developing in many fields such as automatic object recognition, object tracking or image segmentation from 2D images or videos. When developing such an algorithm, at some stage in the progress, extensive evaluation and testing is important. To do so, environments are required which can be used to test the algorithm. A common problem is the creation of such environments, which can be very expensive because they need to cover as many conditions and situations as possible, such as different light conditions or different structures. They also have to be able to respond to the algorithm in order to reflect its result.

Furthermore, for the evaluation of the algorithm, there may be additional meta information required, which describes the content of the environments, for example the position of the objects in the scene. This data has to be presumed correct, thus called Ground Truth. A manual preperation of these data can be very circumstantial and time-consuming. Therefore, the effecicient generation of such data is another problem.

This work proposes an approach of using game engines for testing and evaluation of computer vision algorithms. They are used as virtual test environments as well as for the generation of manifold Ground Truth data. Furthermore, *Virtual Environment Controlling Application* (VECA) is presented, a framework which assists the user in setting up a game engine as a virtual test environment.

Kapitel 1

Einleitung

1.1 Motivation

Die Verwendung vielseitiger Computer Vision Technologien hat sich mittlerweile immer häufiger im allgemeinen Gebrauch verbreitet. Durch den rasanten Fortschritt im Bereich der Computertechnologie können immer komplexere und aufwendigere Algorithmen entwickelt werden. Dazu zählen Berechnungen in den Bereichen automatischer Objekterkennung, Objekttracking, Bildsegmentierung oder 3D-Rekonstuktion. Es wird laufend nach neuen Wegen gesucht, solche Algorithmen noch genauer und allgemeiner zu gestalten, um sie in vielen verschiedenen Bereichen einsetzen zu können. Gebiete wie Robotik, Human-Computer Interaction, Überwachung, Maschineninspektion oder visuelle Analyse im Verkehr oder bei sportlichen Veranstaltungen stützen sich auf Computer Vision. Trotz der umfangreichen Forschung in solche Technologien sind viele Lösungen für bestimmte Probleme im Bereich Computer Vision noch nicht erreicht bzw. nicht zufrieden stellend. Beispielsweise gibt es im Bereich 3D-Rekonstruktion noch keine Methode, die jedes beliebige Foto in eine exakt nachgebildete dreidimensionale Szene umwandeln kann, da es enorm schwierig ist, sämtliche Ausnahmesituationen und verschiedenen Erscheinungsbilder (Lichtverhältnisse, Objektbeschaffenheiten etc.) abzudecken. Das große Ziel, dass der Computer die Umgebung mit der gleichen Genauigkeit wie der Mensch wahrnehmen kann, liegt noch in weiter Ferne. Daher ist die Entwicklung neuer Technologien in diesem Gebiet heute, ebenso wie seit einigen Jahren, ein großes Thema.

1.2 Problemstellung

Wenn man nun solche Algorithmen entwickelt, ist es dabei ein wichtiger Schritt, diese ausgiebig zu testen und zu evaluieren. Das kann mitunter sehr schwierig und aufwendig sein, weil man den Algorithmus unter möglichst vielen verschiedenen Bedingungen und Situationen testen muss und die Er-

stellung einer Umgebung für diese Zwecke meist sehr mühsam ist. Außerdem benötigt man zum Evaluieren häufig umfangreiche Metainformationen über die abgebildete Szene. Wenn man nämlich mit Hilfe von Computer Vision Informationen aus einem Bild extrahiert, benötigt man diese Informationen, um feststellen zu können, ob der Algorithmus das richtige Ergebnis liefert. Diese Informationen müssen also von vornherein als korrekt gelten. Ein grundliegendes Problem ist also das aufwendige Erstellen von Umgebungen, auf die der Algorithmus getestet werden kann, sowie die effiziente Generierung von Metainformationen, um eine Evaluierung durchführen zu können. Im Zuge dieser Arbeit wird der Ansatz verfolgt, Game Engines für diese Aufgaben zu verwenden und diese sowohl als virtuelle Testumgebung, als auch zur Generierung von Daten nutzen zu können.

1.3 Zielsetzung

Diese Arbeit soll zeigen, dass es sinnvoll ist, Game Engines für die Erstellung von virtuellen Umgebungen zu verwenden. Es werden die Vorteile und nützliche Funktionalitäten von Game Engines analysiert, sowie die Anforderungen erforscht, die eine Game Engine erfüllen muss, um für diese Zwecke genutzt werden zu können. Prinzipiell wird in der Arbeit auf zwei Fragen eingegangen:

- Wie kann man eine Game Engine als virtuelle Testumgebung nutzen, in der ein Computer Vision Algorithmus läuft und die auf das Ergebnis des Algorithmus reagiert und sich anhand dessen verändert?
- Wie kann man eine Game Engine dazu nutzen, möglichst effizient sehr große Mengen an Ground Truth Daten (erklärt in Abschnitt 2.4) zu generieren?

Es wird also der Frage nachgegangen, wie die Game Engine mit der Applikation, die den Computer Vision Algorithmus enthält, kommunizieren kann, damit die Game Engine dann auf das Ergebnis des Algorithmus reagieren kann, um so als Testumgebung fungieren zu können. Des Weiteren wird untersucht wie man allgemein in die Game Engine eingreifen und die gewünschten Daten herausholen kann. Außerdem wurde ein Framework – gennant *Virtual Environment Controlling Application* (VECA) – entwickelt, das den Benutzer dabei unterstützt, eine Game Engine als virtuelle Testumgebung aufzusetzen und das die Hauptteile der Kommunikation und der Datengenerierung automatisiert.

1.4 Gliederung

Die Arbeit ist in sieben Kapitel eingeteilt. Nach der Einleitung werden in Kapitel 2 zunächst die für die Arbeit relevanten Grundlagen erläutert. Es

werden Einblicke in den Bereich Computer Vision gegeben, eine Abgrenzung zu ähnlichen Begriffen vollzogen und Beispiele sowie aktuelle Anwendungsbereiche angeführt. Auch der in diesem Zusammenhang für die Arbeit wichtige Begriff *Vision Engine* wird erklärt. Des Weiteren wird auf das Gebiet der Game Engines und auf die Bedeutung der angesprochenen Metainformationen, auch bekannt als *Ground Truth*-Daten, eingegangen. In Kapitel 3 werden dann bestehende Ansätze erläutert, die sich bereits damit beschäftigt haben, eine effiziente Datengenerierung für Computer Vision durchzuführen bzw. virtuelle Umgebungen zu erstellen. Kapitel 4 beschreibt detailliert den eigenen Ansatz mit der Verwendung von Game Engines. Hier wird das allgemeine Konzept beschrieben, wie sich die einzelnen Komponenten verhalten und miteinander kommunizieren. Des Weiteren wird analysiert, warum es sinnvoll ist, Game Engines zu verwenden, welche Anforderungen diese erfüllen müssen, um für solche Zwecke genutzt werden zu können und welche bekannten Game Engines anhand dessen in Frage kommen. Außerdem wird eine Einteilung der möglichen Szenarien vorgenommen, die man mit so einem Framework abdecken könnte. Anschließend wird in Kapitel 5 die Implementierung von VECA erläutert und in Kapitel 6 wird anhand eines Anwendungsbeispiels beschrieben, wie man VECA letztendlich in sein eigenes Projekt einbindet und die benötigten Teile implementiert, um einen Algorithmus testen zu können. Zum Schluss wird in Kapitel 7 noch ein kurzes Fazit gezogen und ein Ausblick auf mögliche Weiterführungen gegeben.

Kapitel 2

Grundlagen

In dieser Arbeit wird versucht mit der Verknüpfung von Computer Vision und Computerspielen verschiedene Bereiche der Computertechnik miteinander zu verbinden, die auf den ersten Blick sehr gegensätzlich wirken, aber sehr stark voneinander profitieren können. So kann das Gebiet rund um Computer Vision beispielsweise dazu verwendet werden, um dem Spieler durch Gestenerkennung natürliche Bewegungen als eine weitere Inputmethode anzubieten [11], was mittlerweile in Spielekonsolen eine weit verbreitete Anwendung findet. Auf der anderen Seite können Spiele dazu verwendet werden, um benötigte Daten für Computer Vision Systeme zu generieren, wie in Abschnitt 3.2 beschrieben wird. Der eigene Ansatz in Kapitel 4 beschreibt ebenfalls eine Möglichkeit, um im Bereich Computer Vision von Spielen profitieren zu können.

In folgendem Kapitel wird auf die verschiedenen in der Arbeit vorkommenden Gebiete allgemein eingegangen und deren Komponenten und Funktionsweisen erläutert, um einen Überblick über die verwendeten Begriffe und Bereiche zu geben. Zunächst wird der Begriff *Computer Vision* definiert, sowie Beispiele und Anwendungsbereiche genannt. Im Zuge dessen erfolgt auch eine kurze Beschreibung zur Abgrenzung zu anderen Fachgebieten wie *Image Processing* und *Computergraphik*. Des Weiteren wird der zugehörige, vielleicht nicht allzu bekannte Begriff *Vision Engine* erläutert. Außerdem wird der in der Arbeit sehr wichtige Begriff *Game Engine* definiert und ein Überblick über bestehende Arten von Game Engines gegeben. Zum Abschluss des Kapitels erfolgt eine Beschreibung von sogenannten *Ground Truth* Daten, die eines der Kernelemente der Arbeit bezeichnen und im Bereich Computer Vision eine wichtige Rolle spielen.

2.1 Computer Vision

2.1.1 Definition

Das Gebiet Computer Vision beschäftigt sich hauptsächlich mit der automatischen Analyse und Weiterverarbeitung von Bildern. Der Computer soll visuelle Informationen so interpetrieren können, dass er auf Beschreibungen des Dargestellten schließen kann. Trotz großer Fortschritte in diesem Bereich ist das Ziel, dass ein Computer ein Bild auf demselben Level wie ein Mensch interpretieren kann, noch lange nicht erreicht. So ist es für einen Menschen unglaublich einfach, die Umgebung über das visuelle Wahrnehmungssystem einschätzen zu können. Dadurch passiert es oft, dass von bestimmten Problemen im Bereich der Bildanalyse angenommen wird, dass eine schnelle Lösung existiert, obwohl sich in der Praxis dann oft herausstellt, dass eine effiziente Lösung sehr schwierig oder sogar unmöglich zu bewerkstelligen ist [5]. Zum Beispiel reicht für einen Menschen ein kurzer Blick auf ein Objekt, beispielsweise einen Tisch, aus, um eine genaue Abschätzung der Entfernung des Objekts zu tätigen, sowie Informationen über dessen Form und Strukturen anhand der Lichteinflüsse und Oberflächeneigenschaften zu erkennen. Bei der Betrachtung eines Bildes kann sofort beurteilt werden, ob es sich um eine Berglandschaft handelt oder eine Aufnahme einer großen Stadt. Durch unsere Erfahrung und unser Vorwissen können wir ohne Probleme jede Person auf einem Familienfoto benennen und sogar den emotionalen Zustand der Personen aufgrund der Interpretation der Gesichtsausdrücke erkennen. Ein Computer hat jedoch keinerlei Vorkenntnisse über Farbverhalten, Lichtstrahlung oder Oberflächenstrukturen, die solche Schlussfolgerungen zulassen. Daraus ergibt sich die große Schwierigkeit im Gebiet Computer Vision, weil man im Allgemeinen unbekannte Fakten aus unzureichenden Informationen extrahieren möchte. Daher muss man auf Physik-basierende und wahrscheinlichkeitstheoretische Modelle zurückgreifen, um potentielle Lösungen zu erforschen [23]. Computer Vision beschäftigt sich also mit dem Extrahieren von Informationen über eine in einem oder mehreren Bildern vorhandene Umgebung mit Hilfe von Algorithmen und Modellen [10].

2.1.2 Abgrenzung

Der Begriff Computer Vision wird häufig gemeinsam mit ähnlichen Begriffen wie *Computergraphik* oder *Image Processing* genannt. Obwohl sich diese Gebiete meist in ähnlichen Umfeldern befinden oder sogar ineinandergreifen, kann man doch eindeutige Unterschiede feststellen.

Computergraphik

In [20] wird beschrieben, dass das Gebiet der Computergraphik sich hauptsächlich mit der Erzeugung von virtuellen Ansichten beschäftigt. Durch ge-

gebene Informationen über eine Szene, wie Beleuchtungspositionen oder 3D-Modelle, wird mittels Rückprojektion auf die zweidimensionale Bildebene einer virtuellen Kamera das entsprechende Bild erzeugt. Solche computergraphische Visualisierungen werden in verschiedenen Anwendungsgebieten wie Produktentwicklung, Architektur oder Computerspielen eingesetzt. Ebenso wie die perspektivische Projektion spielen für die graphische Qualität des erzeugten Bilds auch weitere Komponenten wie Schattenwurf, Beleuchtung, Reflexionen oder Oberflächenstrukturen eine wichtige Rolle. Diese werden durch die in der Szene gegebenen Informationen berechnet. Game Engines bewegen sich sehr stark in diesem Gebiet. Mehr dazu in Abschnitt 2.3.

Die Verfahren aus dem Bereich Computer Vision verfolgen jedoch den umgekehrten Weg. Man versucht aus einem oder mehreren natürlichen Bildern mit keiner oder nur geringer Kenntnis über die Szeneneigenschaften, Informationen über die Welt zu extrahieren. Die Schwierigkeit dieser Problemstellung wird oft unterschätzt, da man zu früherer Zeit im Bereich der künstlichen Intelligenz angenommen hat, dass die kognitiven (Logik, Planen) Teile der Intelligenz um einiges komplizierter wären als die Wahrnehmungskomponenten [23]. Wenn man die beiden Disziplinen verbindet, also ein Bild statt einem 3D-Modell herzunehmen, um eine virtuelle Nachbildung zu erzeugen, nennt man diesen Vorgang *Image Based Rendering* [15].

Man kann jedoch feststellen, dass die beiden Disziplinen Computergraphik und Computer Vision sich immer mehr annähern und voneinander profitieren. Auch in dieser Arbeit wird versucht, diese beiden Bereiche erfolgreich miteinander zu verknüpfen.

Image Processing

Ein weiteres Fachgebiet in diesem Bereich ist Image Processing. Dieses umfasst die Manipulation von Bilddaten für die Betrachtung. Ein Beispiel hierfür ist die Restaurierung von Bildern. Nachdem viele Originalbilder durch ungewollte Verzerrungen oder unerwünschtes Rauschen nicht die gesuchten Informationen darstellen kann, ist die Entwicklung von Technologien, die die Bildqualität aufwerten, sehr wichtig geworden. Die Veränderung von Bilddaten kann aber auch als Stilmittel wie beispielsweise bei Special Effects für Filme eingesetzt werden. Image Processing verfolgt demnach zwei verschiedene Ziele [19]:

- Das Verbessern der visuellen Erscheinung von Bildern für die Betrachtung durch den Menschen, und
- das Vorbereiten von Bildern für das Messen von vorhandenen Merkmalen und Strukturen.

Image Processing verwendet also die direkte Manipulation von Bilddaten, um daraus ein neues Bild zu generieren.

Im Gegensatz dazu beschäftigt sich Computer Vision nur mit der Analyse von Bilddaten, um Erfassungen durchzuführen, Klassifikationen vorzunehmen, Beschreibungen zu extrahieren und nützliche Informationen zu bergen und diese zu interpretieren. Teilweise deckt sich das Gebiet jedoch mit jenem von Image Processing, wie beispielsweise in den Bereichen Segmentierung oder Kantenerkennung [5]. Beide Begriffe findet man in der Literatur häufig unter beiden Disziplinen.

2.1.3 Entwicklung

Das Gebiet Computer Vision existiert zwar schon etwa seit den 60er Jahren, aber erst kürzlich war es möglich nützliche Computersysteme im Bereich Computer Vision zu entwickeln [10]. Nachdem Bildverarbeitungsalgorithmen meist sehr rechenintensiv sind, lag am Anfang der Entwicklung in diesem Gebiet der Schwerpunkt auf dem Erreichen des Höchstmaßes an Effizienz, um auch auf den damaligen Prozessoren mit geringer Leistung Lösungen anbieten zu können. Durch den drastischen Anstieg der Leistungsfähigkeit der Rechner konnte der Fokus in letzter Zeit auf andere Gebiete gesetzt werden. So besteht mittlerweile reger Bedarf für allgemeine Lösungen, die auf weniger eingeschränkte Szenarien anwendbar sind und über Programmschnittstellen gekapselt werden. Außerdem entwickelt sich viel im Bereich der Echtzeit-Videoverarbeitung, wo typischerweise 25 Bilder pro Sekunde abgearbeitet werden müssen, und daher die Bildverarbeitungsmethoden dementsprechend auf eine Verarbeitungszeit von Millisekunden optimiert werden sollten [20]. Dieser Umstand ist auch heute noch eine große Herausforderung.

2.1.4 Beispiele

Um ein Gefühl dafür zu bekommen, welche Problemstellungen im Bereich Computer Vision vorkommen, werden hier ein paar typische Beispiele angeführt [10, 17].

Face Detection

Eines der wohl populärsten Beispiele in Computer Vision ist Face Detection. Man möchte dabei nicht nur Gesichter auf Bildern lokalisieren, sondern auch individuelle Merkmale (Augengröße, Abstände, Winkel, etc) extrahieren, um die abgebildeten Menschen identifizieren zu können. Es gibt verschiedene Faktoren, die diese Aufgabe so schwierig machen. So ist das auf dem Bild festgehaltene Erscheinungsbild meist extrem unterschiedlich, vor allem was Beleuchtung, Gesichtsausdruck, Kamerawinkel und das individuelle Aussehen eines jeden Menschen angeht. Dadurch kommen zusätzlich zu den Unterschieden der verschiedenen Gesichter die großen Unterschiede desselben Gesichts in verschiedenen Bildaufnahmen. Dennoch kommt Face

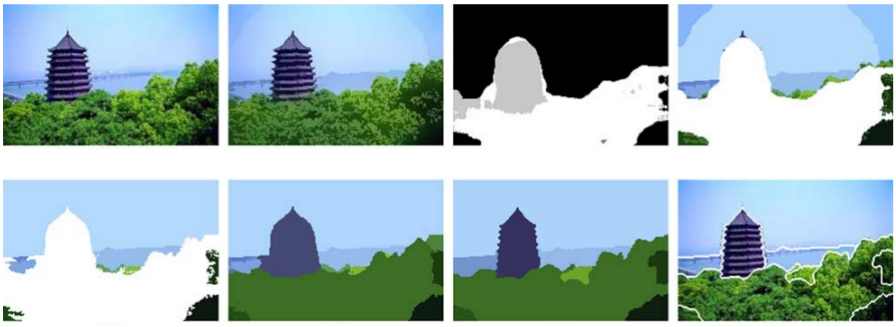


Abbildung 2.1: Ein typischer Vorgang in mehreren Schritten bei der Segmentation eines Bildes [6].

Detection mittlerweile schon erfolgreich in einigen alltäglichen Lebenssituationen, wie beispielsweise Kameras, zur Anwendung.

Depth Estimation

Der Mensch hat die Fähigkeit, die beiden durch die Augen wahrgenommenen Bilder automatisch zu verbinden und durch das Ausschöpfen ihrer Unterschiede eine ziemlich exakte Tiefenwahrnehmung zu erzeugen. Viele Bereiche in Computer Vision verwenden diesen Vorgang als Basis um Tiefeninformationen aus einem Bild zu extrahieren. Dazu werden meist zwei oder mehrere Bilder einer Szene hergenommen, um die Tiefe algorithmisch zu berechnen. Diese benötigt man beispielsweise, wenn man eine Rekonstruktion einer Szene in ein dreidimensionales Modell vornehmen möchte.

Segmentation

Nachdem eine große Aufgabe im Bereich Computer Vision daraus besteht, herauszufinden, welche Pixel eines Bildes wichtig sind, und welche zu ignorieren sind, ist die Segmentierung eine der grundlegendsten Methoden. Durch Segmentierung erfolgt eine Einteilung eines Bildes, indem zusammenhängende Regionen zusammengefasst werden, um eine kompakte Repräsentation der interessanten Bilddaten zu erzeugen, die die wichtigen Eigenschaften des Bildes hervorheben. Abbildung 2.1 zeigt ein Beispiel für Segmentation.

2.1.5 Anwendungsbereiche

Einige aus Computer Vision entstandene Methoden finden sich mittlerweile in vielen verschiedenen praktischen Anwendungen wieder [10, 26, 17, 23]:

- *Zugangsbeschränkungs- und Überwachungssysteme:* hier werden sehr häufig Authentifikationsmethoden wie Fingerabdruckanalyse oder Gesichtserkennung eingesetzt.

- *Human-Computer Interaction:* In den letzten Jahren hat sich ein neues Gebiet der *perceptual interfaces* entwickelt, bei dem es im Bereich Computer Vision darum geht, Sprach- und Soundverarbeitung, sowie haptische Interaktion als neue Inputmöglichkeiten zu erforschen. Die Motivation dahinter ergibt sich aus dem Bedürfnis, sich der wechselnden Natur des Computers anzupassen, sowie eine höhere und kraftvollere User Experience zu erzeugen. Populäre Anwendungen wie Gesture Recognition bei modernen Konsolen zählen zu diesem Gebiet.
- *Visual Effects:* Vor allem bei großen Filmproduktionen ist das Manipulieren und Erweitern von Bilddaten ein Hauptfaktor dafür geworden, dem Film seine persönliche Identität zu verleihen. Im allgemeinen Vordergrund steht dabei zwar die Disziplin der Computergraphik (siehe Abschnitt 2.1.2), aber auch Computer Vision trägt einen sehr großen Teil dazu bei, aufwendige Effekte in Filmen zu erzeugen.

  Beispielsweise ist die Technik des *Motion Capture*, welche physikalische Marker verwendet, die von mehreren Kameras gesehen werden, um die Bewegungen eines Schauspielers einzufangen, mittlerweile weit verbreitet. Diese Bewegungen können dann für die Computeranimation von virtuell erzeugten Modellen verwendet werden.

  Mittels *Match Move* können Compter-Generated Imagery (CGI) und Realbilder durch Tracking von Feature Points zum schätzen der 3D Kamerabewegung und Form der Umgebung zusammengeführt werden.

  Motion Tracking kann dazu verwendet werden, um bestimmte Bildpunkte über mehrere Videoframes hinweg zu tracken und Kamerabewegungen auszugleichen.
- *Traffic Management:* Kameras werden für die automatische Straßenüberwachung eingesetzt, beispielsweise mit eingebauter Optical Character Recognition (OCR) zum Erkennen von Nummernschildern.
- *Search Engines and Image Databases:* Während Suchmaschinen im Bereich der textbasierten Suche schon große Fortschritte gemacht haben, ist die reine Bildsuche ein weitgehend noch zu erforschendes Feld. Beim sogenannten *content-based image retrieval* geht es um die Verarbeitung von Bilddatenbanken mittels der Analyse der Bildinhalte. Sei es zur Suche eines Bildes, oder zur automatische Strukturierung und Organisation der Bilder.
- *Machine Inspection:* In der Industrie werden mittlerweile häufig automatische Verfahren für die Qualitätskontrolle, wie Toleranzmessungen bei Flugzeugflügeln oder Autoteilen, eingesetzt.
- *Sports Analysis:* Verschiedene Computer Vision Systeme werden erfolgreich dazu verwendet, um Statistiken zu sportlichen Ereignissen zu erstellen, die per Hand nicht zu erreichen wären. Abbildung 2.2 zeigt ein System bei einem Fußballspiel, das über drei Kameras das gesam-

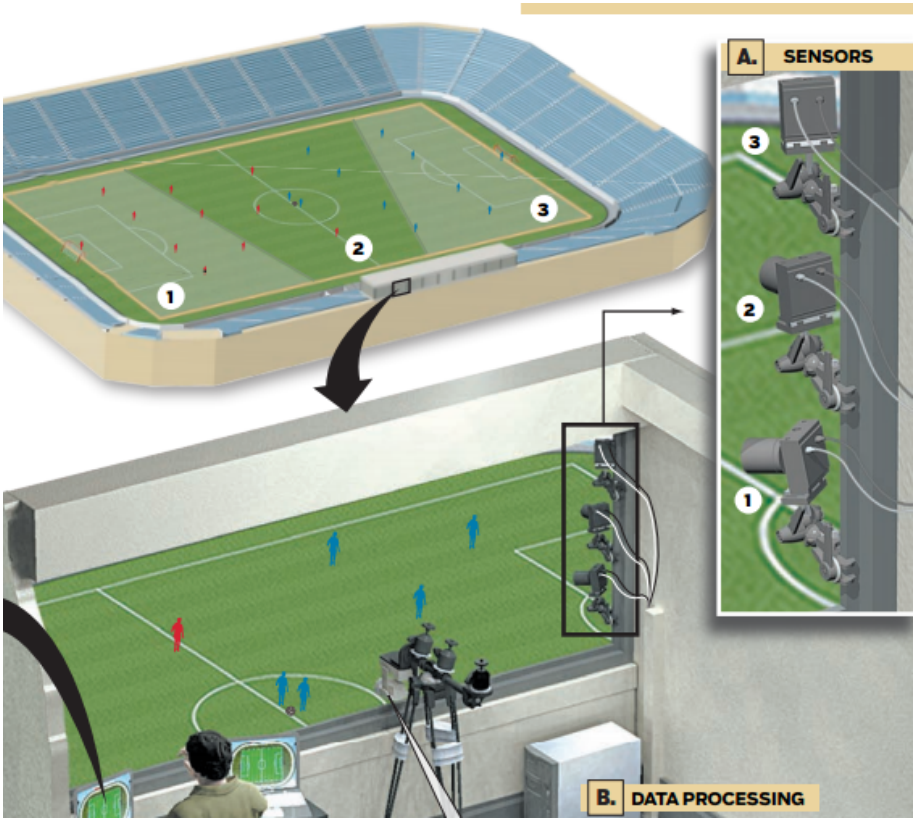


Abbildung 2.2: Ein modernes Computer Vision System zur Erhebung von Statistiken bei einem Fußballspiel [25].

te Feld abdeckt und Informationen über jeden Spieler, wie Spielzeit, Laufwege, Geschwindigkeit, Passgenauigkeit oder Torschüsse, protokolliert.

2.2 Vision Engines

2.2.1 Definition

Der Begriff *Vision Engine* ist im Gegensatz zu den bekannteren Begriffen *Computer Vision* oder *Game Engine* allgemein nicht gebräuchlich. Dementsprechend findet sich in der Literatur keine genaue Definition. Dennoch wird in dieser Arbeit der Begriff verwendet, um bewusst auf eine spezifische Komponente des Systems referenzieren zu können.

Eine Vision Engine kann als Teil eines Vision Systems gesehen werden. Dieses lösst sich nach [14] ähnlich eines biologischen Systems in folgende Komponenten aufteilen:

- *Radiationsquelle:* Eine von der Szene oder dem gewünschten Objekt ausgehende Lichtstrahlung. Diese ist notwendig, um die Umgebung wahrnehmen zu können.
- *Kamera:* Die Kamera ist dazu da, um die Radiation aufzunehmen. Dies geschieht meistens durch eine optische Linse.
- *Sensor:* Der Sensor wandelt die Radiation in ein geeignetes Signal zur Weiterverarbeitung um. Für ein Bildsystem wird üblicherweise ein 2D-Array von Sensoren benötigt, um die flächendeckende Verteilung aufnehmen zu können.
- *Processing Unit:* Diese dient zur Weiterverarbeitung der höher dimensionalen Daten und zur Analyse auf geeignete Merkmale, die dazu verwendet werden können, um Objekteigenschaften zu erkennen und zu kategorisieren. Diesen Part kann eine Vision Engine übernehmen.
- *Akteure:* Diese Komponente reagiert auf die Ergebnisse der visuellen Observation der Processing Unit. Sie wird dann ein Teil des visuellen Systems, wenn das System aktiv auf die Observation reagiert, zum Beispiel ein bestimmtes entdecktes Objekt verfolgt.

Eine Vision Engine ist also ein System, das entsprechende Computer Vision Algorithmen enthält und es ermöglicht, diese auf ein oder mehrere Bilder anzuwenden, die über eine Form des Inputs übergeben werden, um diese weiterzuverarbeiten und ein bestimmtes Ergebnis zu erzielen. Um eine Vision Engine zu beschreiben, kann eine Analogie zum Verhältnis zwischen *Computergraphik* und *Computer Vision* hergestellt werden. Wie in Abschnitt 2.1.2 beschrieben, werden bei Computergraphik vorhandene Informationen dazu verwendet, um ein neues Bild zu erzeugen. Wogegen im Bereich Computer Vision ein vorhandenes Bild verwendet wird, um Informationen des Abgebildeten zu extrahieren. Ebenso verhält es sich zwischen einer Vision Engine und einer Rendering Engine oder Game Engine. Rendering Engines und Game Engines sind Systeme, die sich die Computergraphik zunutze machen, um neue Bilder aus vorhandenen Szenenbeschreibungen zu generieren. Eine Vision Engine ist im Gegensatz dazu ein System, das mit Hilfe von Computer Vision eingehende Bilder auf Beschreibungen untersucht.

2.2.2 Beispiele

Es hat sich mittlerweile eine Vielzahl an Software entwickelt, auf die der Begriff Vision Engine zutreffen würde. Verschiedene Applikationen oder Software Libraries versuchen, Vision Algorithmen zu kapseln und zur Verfügung zu stellen. Sei es für spezifische Aufgabenstellungen oder für die allgemeine Verwendung.

Ein paar Libraries und Programme, die ein möglichst breites Spektrum an Computer Vision Algorithmen anbieten, um möglichst vielen Problemstellungen entgegenzukommen, sind hier angeführt:

- *OpenCV:* Die *Open Source Computer Vision Library*[1] ist eine Sammlung von C und C++ Quellcode und Executables, die ein großes Gebiet von Computer Vision Algorithmen abdecken. Der Code ist für Echtzeitapplikationen optimiert und bietet eine allgemeine Infrastruktur für Anwendungen im Bereich Computer Vision. Die Library enthält mittlerweile mehr als 2500 optimierte Algorithmen, z. B. für Human-Computer Interaction, Gesichtserkennung, Objekterkennung, Klassifizieren von menschlichen Bewegungen, Tracking, Extrahieren von 3D Modellen oder Szenenerkennung. In Abbildung 2.3 wird eine durchgeführte Kantenerkennung mit dieser Library gezeigt. OpenCV enthält C++, C, Python und Java Schnittstellen und unterstützt Windows, Linux, Mac OS sowie Android [17].
- *ImageJ:* Das Java-basierende Image Processing Programm *ImageJ*[2], legt auf die einfache Integrierung von benutzergeschriebenen Plugins in den Bildverarbeitungsprozess Wert. Es enthält außerdem einige eingebaute Methoden aus dem Bereich Image Processing und Computer Vision, wie Filterfunktionen, Kantenerkennung oder Flächenselektion.
- *IVT:* Das *Integrating Vision Toolkit*[3] ist eine plattformunabhängige open source C++ Computer Vision Library mit einer objektorientierten Architektur, die vorgefertigte Funktionen wie Corner Detection, Calibration, Pose Estimation oder Filtering enthält.
- *VXL:* Die sogenannte *Vision*-something-*Library*[4] ist eine Sammlung von C++ Computer Vision Algorithmen. Die Idee dahinter ist, das *X* mit dem Buchstaben auszutauschen, der die Funktionen der jeweiligen Kern-Library beschreibt. Beispielsweise *VNL* für numerische Operationen, oder *VGL* für Geometrie in verschiedenen Dimensionen.
- *VIGRA:* Diese Abkürzung steht für *Vision with Generic Algorithms*[5] und ist eine Computer Vision Library, die besonders auf anpassbare Algorithmen und Datenstrukturen Wert legt und es dem Benutzer ermöglicht, die Komponenten nach den Bedürfnissen der eigenen Applikation anzupassen.

Mittlerweile haben sich in vielen Anwendungsgebieten auch schon erfolgreich Programme entwickelt, die Computer Vision nutzen, um ihr jeweiliges Fachgebiet zu unterstützen. Ein Beispiel hierfür wäre die Software *CellProfiler*[6]. Diese wird von Biologen verwendet, um automatisch eine quantitative Messung von Phänotypen aus tausenden von Bildern durchführen zu können. Dazu sind vom Benutzer keine Kenntnisse über Programmierung oder

[1] http://opencv.org/
[2] http://rsb.info.nih.gov/ij/
[3] http://ivt.sourceforge.net/
[4] http://vxl.sourceforge.net/
[5] http://hci.iwr.uni-heidelberg.de/vigra/
[6] http://www.cellprofiler.org/

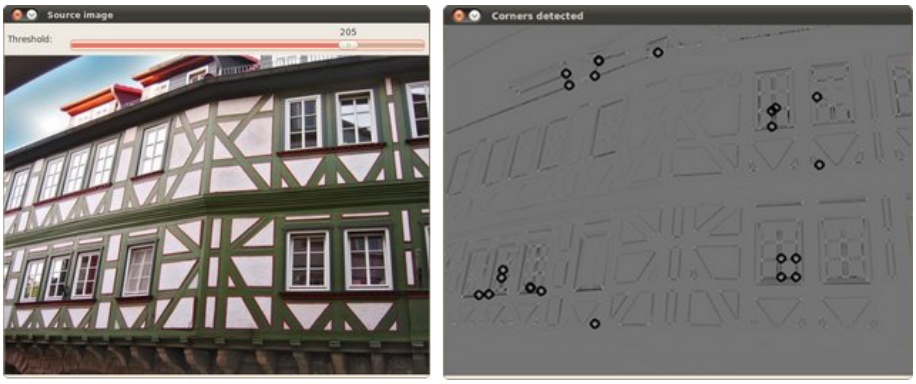


Abbildung 2.3: Beispiel des Harris Corner Detector von OpenCV. Links das Originalbild und rechts das Ergebnis [27].

Computer Vision gefordert, die Software arbeitet komplett für sich.

2.3 Game Engines

Neben der Vision Engine ist auch der Begriff Game Engine für den in dieser Arbeit vorgestellten Ansatz sehr wichtig, da eine solche Verwendet werden soll, um eine virtuelle Testumgebung aufzusetzen.

2.3.1 Definition

Eine Game Engine ist ein Programm oder ein Framework, das die Entwicklung neuer Computerspiele erleichtert. Allgemein kann man sie als eine Middleware-Applikation betrachten, die die allgemeinen Basisfunktionalitäten von Spielen kapselt und dem Spieleentwickler zur Verfügung stellt. Dieser muss dann nur noch die Spiele-spezifischen Komponenten hinzufügen. Durch die Nutzung einer Game Engine ersparen sich Spieleentwickler das Ausprogrammieren vieler Basiskomponenten und können stattdessen sofort auf vielseitig getestete und bewährte Programme zurückgreifen und diese für mehrere Projekte einsetzen. Außerdem profitieren mittlerweile einige Firmen davon, sich nur auf die Entwicklung von Game Engines statt auf fertige Spiele zu konzentrieren, um diese an Spieleentwickler verkaufen zu können.

2.3.2 Was bietet eine Game Engine?

Die verschiedenen eingebauten Komponenten einer Game Engine sind ein guter Grund, diese zu verwenden. Eine ausgereifte Game Engine hat nämlich typischerweise u. a. folgende nützliche Eigenschaften, die die Erstellung einer virtuellen Testumgebung erheblich erleichtern [29]:

- *Graphics:* Das Laden, Platzieren und Animieren von Modellen und Graphiken wird von Game Engines unterstützt.
- *Sounds:* Ebenso wird der Umgang mit Sounds durch eine Game Engine sehr erleichtert.
- *Input:* Das Abfragen von Maus, Tastatur, Gamepad oder anderen Inputmöglichkeiten wird von der Game Engine übernommen.
- *Graphical User Interface:* Ein GUI kann mit einer Engine meist sehr schnell erstellt werden.
- *Rendering:* Die Game Engine erstellt über projektivische Berechnungen automatisch aus der gegebenen 3D Szene ein zweidimensionales Bild, das dann weiterverwendet werden kann.
- *Texturing:* Die Erstellung von Objekten mit bestimmten Oberflächeneigenschaften wird erheblich erleichtert. Die Berechnung von Texturen und Oberflächen wird von der Game Engine übernommen.
- *Künstliche Intelligenz:* Oft haben Game Engines schon vorgefertigte Funktionen im Bereich der künstlichen Intelligenz eingebaut, beispielsweise Wegfindung oder Entscheidungsmodelle.
- *Level- und Statemanagement:* Game Engines sind darauf ausgelegt, dass es mehrere States oder Levels gibt und erleichtert die Handhabung solcher.
- *Lighting:* Typischerweise bieten Game Engines umfangreiche Beleuchtungsmodelle und führen deren Berechnungen automatisch durch. Dies inkludiert Lichtbrechungen, Reflexionen, Spiegelungen und Schattenwurf.
- *Simulation:* Game Engines sind darauf ausgelegt, Animationen und Bewegungen von Objekten darstellen zu können.
- *Physik:* Oft beinhalten Game Engines Möglichkeiten, um komplexe physikalische Berechnungen für realistische Verhaltensmuster durchführen zu können. Dazu zählen beispielsweise Kollisionserkennungen oder Kräftewirkungen.
- *Editor:* Viele Engines sind mit Editoren und graphischen Benutzeroberflächen ausgestattet, die die Erstellung von virtuellen Umgebungen noch weiter erleichtern.
- *Metainformationen:* Game Engines enthalten umfangreiche Informationen über alle in der Szene enthaltenen Objekte. Diese können als Ground Truth Daten genutzt werden.

Diese und weitere Gründe sprechen dafür, dass eine Game Engine im Allgemeinen genügend nützliche Eigenschaften bieten, um mit dieser eine virtuelle Testumgebung erstellen zu können, obwohl die Verwendung einer Game Engine für so eine Aufgabe vielleicht etwas zweckentfremdend sein mag.

2.3.3 Arten von Game Engines

Es ist nicht definiert, wie umfangreich die Middleware sein muss, oder welche Komponenten sie enthalten muss, um als Game Engine bezeichnet werden zu können. Eine Game Engine kann aus einer simplen Programmierschnittstelle oder einem voll ausgebauten Editor bestehen. Dementsprechend lassen sich Game Engines in verschiedene Arten einteilen, die sich je nach Verwendungsweise und Anpassungsmöglichkeiten unterscheiden.

Application Programming Interface

Sogenannte Application Programming Interfaces (APIs) werden meist nicht unter dem Begriff Game Engine geführt. Dennoch dienen sie als vorgefertigte Hilfsmittel, die häufig genutzt werden, um ein Spiel zu entwickeln. Es handelt sich hierbei um Schnittstellen, die eine Anbindung an das System zur Verfügung stellen. Zu den bekanntesten zählen hier *XNA*[7], *DirectX*[8] und *OpenGL*[9]. Zu beachten ist, dass diese Schnittstellen nur die untersten Basisfunktionalitäten eines Spiels, wie beispielsweise Rendering, abdecken. Erweiterte Funktionen wie Physik oder Objektmanagement müssen entweder selbst programmiert, oder über andere Libraries eingebunden werden.

Libraries

Viele Game Engines oder auch Teilkomponenten von Spielen präsentieren sich mittlerweile erfolgreich in Form von stark ausgebauten Programm-Libraries. Diese enthalten die allgemeinen Teile von Computerspielen in Klassen gekapselt und können im eigenen Code verwendet werden. Dazu zählen voll ausgebaute Engines wie beispielsweise die *Slick2D*[10] oder Engines, die sich auf eine bestimmte Komponente, wie etwa Physik, spezialisieren. Zu solchen zählen z. B. die *Havok Engine*[11] oder *Box2D*[12]. Solche Libraries werden häufig auch als *Physik Engine* bezeichnet.

Tools/Editors

Mittlerweile haben sich einige Game Engines entwickelt, die nicht nur die Funktionalitäten eines Spiels zur Verfügung stellen, sondern auch das grundlegende Erstellen und Konfigurieren eines Spiels erleichtern, indem sie graphische Benutzeroberflächen und Editoren beinhalten, mit dem sich ein Entwickler sein Spiel zusammenbauen kann. Oft sind hier kaum oder gar keine Programmierkenntnisse mehr notwendig, um erfolgreich ein neues Spiel zu

[7] http://msdn.microsoft.com/en-us/aa937791.aspx
[8] http://msdn.microsoft.com/library/windows/apps/hh452744.aspx
[9] http://www.opengl.org/
[10] http://www.slick2d.org/
[11] http://www.havok.com/
[12] http://box2d.org/

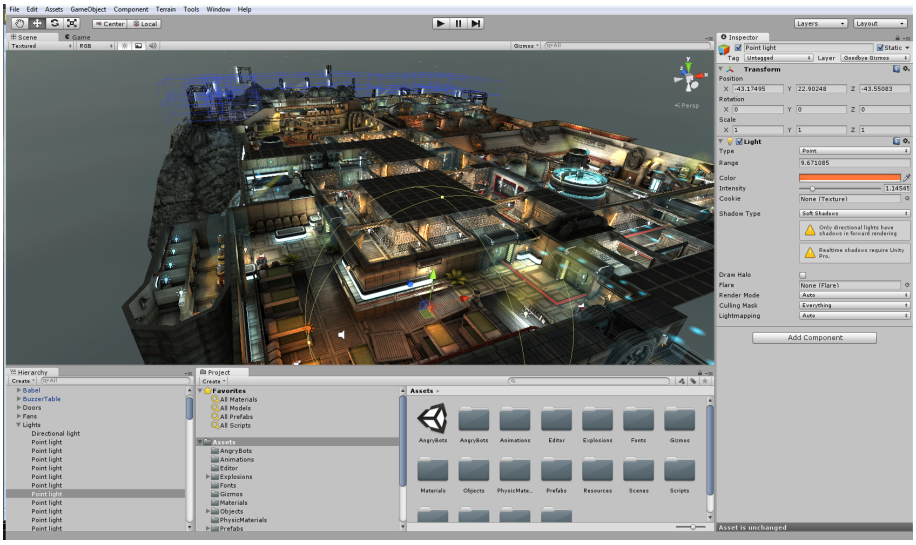


Abbildung 2.4: Ein Screenshot des Editors von *Unity3D* mit einem Standardprojekt.

erstellen und kann somit oft von den Spieledesignern selbst erledigt werden. Dennoch kann man meist über Scripts oder Codeteile in die Engine eingreifen, um seinem Spiel gewünschte Features hinzuzufügen. Zu den bekanntesten Beispielen gehören *CryEngine*[13], *UnrealEngine*[14], *Genesis3D*[15], *Torque*[16] oder *Unity3D*[17], dessen Editor in Abbildung 2.4 gezeigt wird.

2.4 Ground Truth Daten

Ein wichtiger Faktor im Bereich Computer Vision sind die sogenannten *Ground Truth Daten*. Vor allem wenn es um die Entwicklung und Evaluierung von Computer Vision Algorithmen geht, wird dieser Begriff oft genannt [9, 18]. Eine wichtige Voraussetzung für jedes System, das automatisierte Analysen durchführt, ist nämliche eine Methode, um dessen Leistung zu evaluieren. Bei Ground Truth Daten handelt es sich um Datensätze von Ergebnissen, die von vornherein als *korrekt* festgelegt wurden [8]. Die Ergebnisse und Entscheidungen der Computer Vision Applikationen werden dann mit jenen Datensätzen verglichen, um deren Genauigkeit abschätzen zu können. Wenn man beispielsweise einen Algorithmus zur Gesichtserkennung testen möchte, ist es bei einer geringen Anzahl an Bildern einfach manu-

[13] http://mycryengine.com/
[14] http://www.unrealengine.com/
[15] http://www.genesis3d.com/
[16] http://www.garagegames.com/products/torque-3d
[17] http://unity3d.com/

ell zu überprüfen, ob der Algorithmus die Gesichter erkannt hat. Falls man aber eine umfangreiche Evaluierung des Algorithmus mit mehreren Tausend Bildern durchführen möchte, wird die Überprüfung schon aufwendiger. Hier wäre es erstrebenswert, zu jedem Bild die korrekten Ground Truth Daten zu besitzen, die die Position der Gesichter auf dem jeweiligen Bild angeben, um eine automatische Evaluation des Algorithmus durchführen zu können.

2.4.1 Generierung

Nun stellt sich jedoch die Frage, wo man solche Ground Truth Daten herbekommt. Die manuelle Erstellung solcher Daten ist nämlich meist sehr mühsam und zeitaufwendig [18], vor allem wenn man es mit großen Datensätzen, wie ganze Bilddatenbanken oder Videos, zu tun hat. Bei einem Video von fünf Minuten, das mit 30 fps (frames per second) aufgenommen wurde, müssten um die zugehörigen Ground Truth Daten zu generieren insgesamt 9000 Bilder annotiert werden [22], was manuell sehr aufwendig ist. Ein weiteres Problem ist, dass diese Daten viele unterschiedliche Faktoren abdecken müssen, wie z. B. verschiedene Umgebungen, Lichtverhältnisse oder Szenerien, um einen umfangreichen und vielfältigen Testvorgang gewährleisten zu können. In Kapitel 3 werden verschiedene bestehende Ansätze beschrieben, wie man effizient Ground Truth Daten erzeugen kann. Diese Arbeit versucht Game Engines, neben der Verwendung als virtuelle Testumgebung auch für die reine Generierung großer Mengen an Ground Truth Daten herzunehmen.

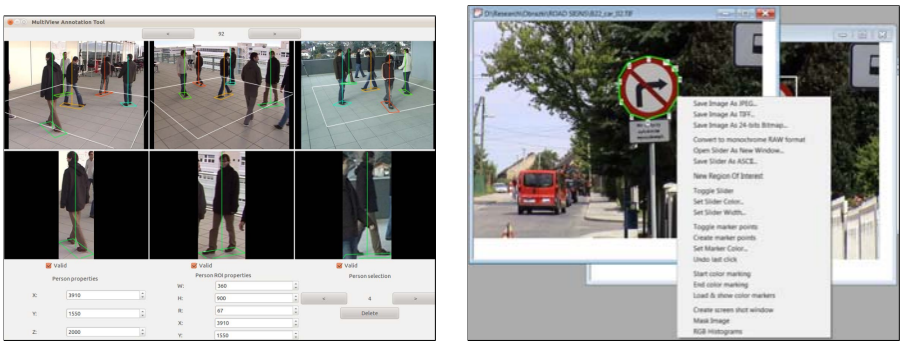
Kapitel 3

Bestehende Lösungsansätze

Die Verwendung aufwendiger Computer Vision Algorithmen für Bild- und Videoanalyse gewinnt immer mehr an Bedeutung und wird mittlerweile in vielen verschiedenen Bereichen eingesetzt. Wie bereits erwähnt, ist ein wichtiger Schritt in der Entwicklung solcher Algorithmen das ausgiebige Testen, sowie die Evaluierung anhand der in Abschnitt 2.4 beschrieben Ground Truth Daten. Dieses Kapitel beschäftigt sich mit bestehenden Lösungsansätzen, wie man solche Daten erzeugen kann und mit bereits erfolgreichen Erstellungen von virtuellen Umgebungen.

Nachdem es problematisch und aufwendig ist, manuell Ground Truth Daten zu erzeugen, könnte man im Allgemeinen annehmen, dass man, um dieses Problem zu lösen, das Bedürfnis nach einem Programm oder Algorithmus hätte, der automatisch Ground Truth Daten aus Bildern erzeugt. Dies würde aber wiederum keinen Sinn machen, denn dann würde dieses Programm bereits die Aufgabe erledigen, für die das eigentliche System das man evaluieren möchte gedacht ist und würde seinerseits wieder Ground Truth Daten zur Evaluierung benötigen [18].

Es gibt daher verschiedene Ansätze, wie man den Benutzer dabei unterstützen kann, Ground Truth Daten zu erzeugen. In [21] wird beispielsweise beschrieben, wie eine kleine, bereits vorhandene Menge an Ground Truth Daten für Segmentierungsalgorithmen dazu verwendet werden kann, durch kombinatorisches Zusammensetzen eine exponentiell größere Datenmenge zu generieren. Des Weiteren gibt es eine Vielzahl an Applikationen, die sich mit diesem Gebiet auseinander setzen. Diese reichen von semi-automatischen Tools, die ein Graphical User Interface (GUI) zur Verfügung stellen, über das Erstellen von virtuellen Umgebungen, bis hin zur Einbettung des Generierungsvorgangs in den Kontext eines Computerspiels. Im Zuge dieser Arbeit wird anschließend ein eigener Ansatz entwickelt, wie man eine virtuelle Testumgebung erstellen und das Problem der Generierung von Ground Truth Daten lösen kann. Mehr dazu in Kapitel 4.



(a) (b)

Abbildung 3.1: Benutzeroberflächen von zwei Programmen zur manuellen Annotation von Daten [7, 24]. Eines dient zum kennzeichnen von Personen in einem Video (a), das andere zur manuellen Erzeugung einer Kontur eines Objekts auf einem Bild mit Hilfe eines multifunktionalen Annotationstools (b).

3.1 Tools

Ein Ansatz, der sich zwar nicht mit virtuellen Umgebungen beschäftigt, aber das Problem der Generierung von Ground Truth Daten von Bildern anspricht, ist die Entwicklung darauf spezialisierter Tools, die den Benutzer unterstützen. Mit diesen Tools kann man Annotationen von Bildern durchführen. Meist können über graphische Benutzeroberflächen Eingaben getätigt werden, die zur Generierung von Ground Truth Daten dienen. Teilweise beinhalten solche Tools auch semi-automatische Algorithmen, die dem Benutzer grundlegende Aufgaben im Generierungsprozess abnehmen.

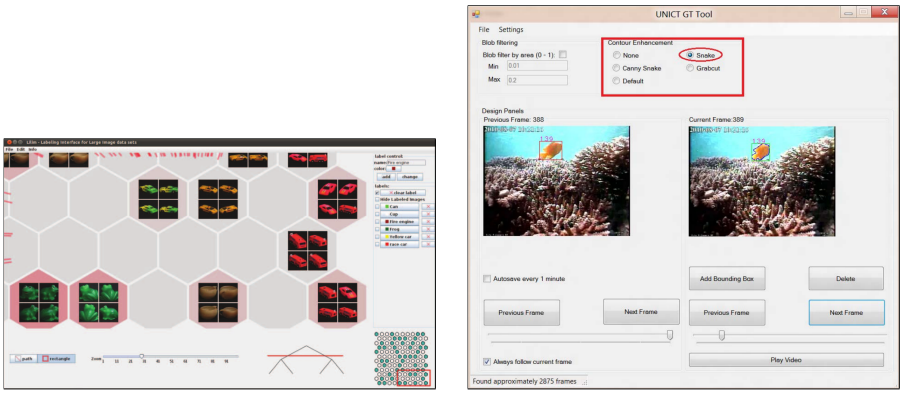
3.1.1 Annotation Tools

In [24] wird ein Programm vorgestellt, in dem man die Position und Ausrichtung von Personen in Videos kennzeichnen kann. Mit Hilfe einer Benutzeroberfläche können Rechtecke erzeugt werden, die die auf dem Video abgebildeten Personen repräsentieren.

Das Tool in [7] dient dazu, die Kontur von Objekten zu erzeugen, Punkte von Interesse hervorzuheben und Stereo-Korrespondenz zu kennzeichnen. Die Ground Truth Daten werden also durch Benutzereingabe manuell erzeugt. Die Verwendung der beiden Tools wird in Abbildung 3.1 dargestellt.

3.1.2 Semi-Automatische Tools

Solche Tools zur Erzeugung von Ground Truth Daten können aber durchaus auch teilweise automatisch sein, um den Benutzer noch weiter zu unterstützen. Das bedeutet, dass einfache Analysealgorithmen in einer Vorbearbei-



(a) (b)

Abbildung 3.2: Diese semiautomatischen Tools treffen Vorentscheidungen zur Annotation und können dann vom Benutzer angepasst werden [16, 18]. (a) zeigt die Benutzeroberfläche eines Tools, das die Bilder automatisch gruppiert, um die Annotation zu erleichtern. In (b) wird ein Tool gezeigt, dass Vorschläge für die Objektkontur automatisch generiert, die dann manuell angepasst werden können.

tung schon automatische Vorschläge generieren, die für das Endergebnis nur noch vom Benutzer abgeändert werden müssen und so eine höhere Usability gewährleisten. Ein vollautomatisches Tool dieser Art kann jedoch nicht existieren, da es die Aufgabe des eigentlichen Algorithmus schon erfüllen würde.

Beispielsweise wird in [18] ein Tool vorgestellt, wo sogenannte self-organizing maps verwendet werden, um die Bilder in einer Vorverarbeitungsphase schon mal automatisch zu kategorisieren und zu gruppieren. Dadurch wird der Prozess der Annotation erheblich erleichtert, weil der Benutzer dann nicht nur einzelne Bilder, sondern ganze Bildgruppen auf einmal kennzeichnen kann.

In [16] werden Vorschläge für Annotationen durch einfache Objekterkennung automatisch generiert. Der Benutzer kann diese dann manuell anpassen. Abbildung 3.2 zeigt die Benutzeroberflächen dieser Programme.

3.2 Spiele

Ein weiterer Ansatz zur Generierung von Ground Truth Daten ist die Verwendung eines Spielkontext, um solche Daten mit Hilfe von teilnehmenden Spielern zu erzeugen.

3.2.1 Games With a Purpose

Sogenannte *Games With a Purpose (GWAP)* [1] verwenden Spielumgebungen, um den Input der teilnehmenden Spieler in Ground Truth Daten umzuwandeln. Es wird hierbei die Idee verfolgt, die Energie, die von Menschen rund um die Welt in Computerspiele gesteckt wird, zu bündeln, um Probleme im großen Umfang lösen zu können. Solche Spiele nutzen die menschliche Wahrnehmungskraft als Prozessoren, um Daten zu generieren oder Probleme zu lösen, die von Computern noch nicht bewerkstelligt werden können. Diese Wahrnehmungskraft ist jener von Computer nämlich immer noch weit voraus. Der Mensch benötigt jedoch eine Form von Anreiz, um in diesen Prozess einzusteigen. Onlinespiele bieten eine Möglichkeit, um den Spieler dazu anzuregen, aktiv an dem Prozess teilzunehmen. Die Motivation der Spieler liegt hier nicht in der Intention, Probleme zu lösen, sondern im Spielspaß und im Spieldesign. *Games With A Purpose* sind mit Applikationen in verschiedenen Bereichen wie Internetsuche, Sicherheit, Internetzugänglichkeit oder auch Computer Vision vertreten. In diesem Abschnitt werden zwei Spiele betrachtet, die sich speziell mit der Generierung von Ground Truth Daten beschäftigen.

3.2.2 ESP Game

Vor allem für Suchmaschinen, aber auch für Computer Vision Algorithmen, ist es meist sehr wichtig, vorhandene Bilder mit angemessen Labels zu versehen [1]. Sei es, um die Bilder effizient kategorisieren zu können, oder um die Labels als Ground Truth Daten zu verwenden, die manuelle Kennzeichnung ist bei großen Bilddatenbanken jedenfalls sehr aufwendig. Das *ESP Game* [2] bettet den Prozess der Bildannotation in den Kontext eines Spiels und verwendet den Mehrspieleraspekt, um eine riesige Menge an Bildern mit Wörtern zu kennzeichnen. Dazu werden über das Internet jeweils zufällig zwei Spieler ausgewählt. Diese bekommen dann beide dasselbe Bild angezeigt und müssen versuchen, sich auf ein Wort zu einigen, das das Bild beschreibt, indem jeder von ihnen unabhängig Vorschläge in ein Textfeld eingibt. Die Spieler sehen aber nicht den Vorschlag des jeweils anderen. Sobald beide Spieler den gleichen Vorschlag gemacht haben, wird das Bild mit diesem Gekennzeichnet und das nächste Bild angezeigt. Über ein Punktesystem wird die Motivation der Spieler aufrechterhalten. Durch diese Methode könnte man mit nur 5000 permanent aktiven Spielern alle Bilder im Web, die von Google indiziert sind, innerhalb von ein paar Wochen kennzeichnen.

3.2.3 Peekaboom

Noch einen Schritt weiter geht das Spiel *Peekaboom* [3]. Für viele Bereiche in Computer Vision ist es nämlich nicht nur interessant, welche Objekte im Bild abgebildet sind, sondern auch, wo sie sich befinden. Dazu werden

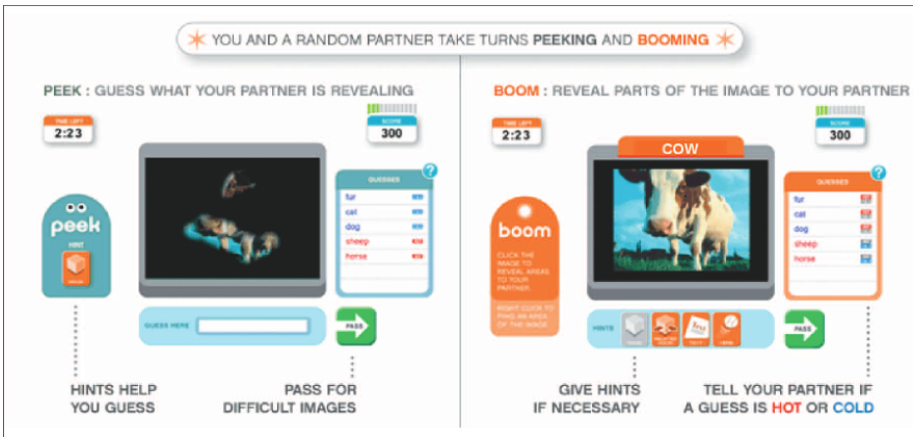


Abbildung 3.3: Das Spiel *Peekaboom* wird verwendet, um die Objektposition in einem Bild festzuhalten [3]

auch hier zwei Spieler ausgewählt, wobei einer der beiden das Bild, sowie ein zugehöriges Wort angezeigt bekommt. Dieser kann jetzt Schritt für Schritt Teile des Bilds für den anderen Spieler aufdecken, damit dieser das Wort erraten kann. Sobald er das Wort erraten hat, kann die ungefähre Position des Objekts festgehalten werden und das Spiel geht mit dem nächsten Bild weiter. Die verschiedenen Benutzeroberflächen für die beiden Spieler werden in Abbildung 3.3 dargestellt. Die angezeigten Wörter für den ersten Spieler werden direkt aus den Resultaten des *ESP Games* übernommen.

3.3 Virtuelle Umgebungen

Es gibt aber auch Ansätze, die bereits die Idee der virtuellen Umgebung verfolgen. Diese sind dem benötigten Kontext angepasst und dienen entweder als Testumgebung oder verwenden Renderings, um die benötigten Ground Truth Daten zu generieren. Solche Umgebungen können unter Umständen zwar aufwendig zu entwickeln sein, haben aber den Vorteil, einen sehr hohen Grad an Flexibilität zu bieten und können den verschiedensten gewünschten Situationen sehr schnell angepasst werden. Es gibt bereits bewährte Beispiele, wo virtuelle Umgebungen erfolgreich als Testumgebung genutzt werden und Ground Truth Daten generieren.

So wird beispielsweise in [4] eine Simulation von Entscheidungsprozessen und Navigationsmustern von Fußgängern bei der Suche nach vordefinierten Zielen innerhalb eines Gebäudes vorgestellt. Die Simulation soll dazu dienen, nachvollziehen zu können, wie Entscheidungen von Personen mit verschiedener Intention, sowie Personen mit eingeschränkter Mobilität, zustande kommen. Dazu wurde eine virtuelle Umgebung entwickelt, die ein 3D Modell des tatsächlichen Gebäudes beinhaltet. Ein Ausschnitt dieser virtuellen

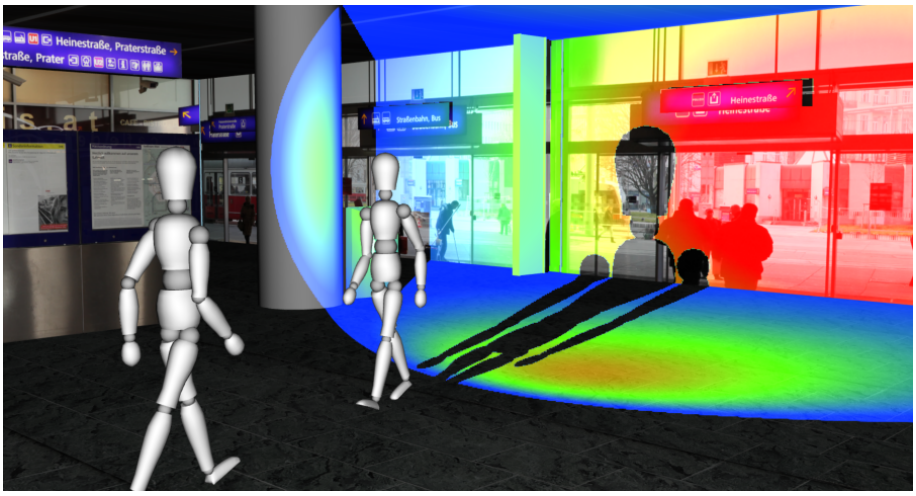


Abbildung 3.4: Diese virtuelle Umgebung dient der Simulation von Entscheidungsprozessen und Navigationsmustern von Fußgängern [4].

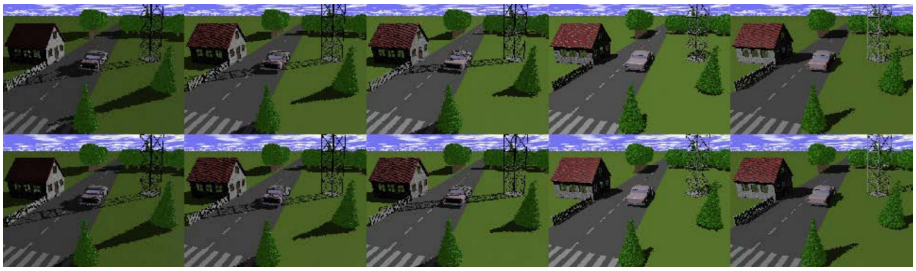


Abbildung 3.5: Eine Rendering Software dient zur Erzeugung von Vergleichsdaten für Schattenerkennungsalgorithmen [13].

Umgebung wird in Abbildung 3.4 gezeigt. Zusätzlich sind die einzelnen Objekte mit semantischen Notierungen versehen, die als Ground Truth Daten dienen. Innerhalb des Modells bewegen sich Agenten, die die Umgebung wahrnehmen können und aufgrund der annotierten Daten Entscheidungen treffen. Diese Entscheidungen werden von den Agenten dann direkt in der virtuellen Umgebung ausgeführt.

Abbildung 3.5 zeigt eine virtuelle Umgebung, die wiederum zur reinen Generierung von Ground Truth Daten dient. Diese werden für die Evaluierung eines Algorithmus benötigt, der Schatten von statischen Objekten im Freien erkennt. Dazu wird eine Rendering Software verwendet, die ein bestimmtes Szenario nachbildet. In dieser virtuellen Umgebung werden dann die Schatten exakt nach dem aktuellen Sonnenstand berechnet und können als Vergleichspunkt zu den Ergebnissen des Algorithmus dienen [13].

Durch die ständige Weiterentwicklung der Leistungsfähigkeit von Com-

putern und der immer realitätsnäheren Darstellungsmöglichkeiten komplexer Szenen ist es mittlerweile sinnvoll, virtuelle Umgebungen für solche Zwecke zu nutzen. Wie in den obigen Beispielen gezeigt, sind diese aber meist vollständig neu zu entwickeln und auf die jeweiligen Szenarien vollständig zugeschnitten. Daher ist der Entwicklungsaufwand für solche Umgebungen meist sehr hoch und es ist außerdem kaum eine Wiederverwendbarkeit vorhanden. In dieser Arbeit wird versucht, diesen beiden Nachteilen entgegenzuwirken, indem ein Framework erstellt wird, das auf vorgefertigte Game Engines angewendet werden kann, um virtuelle Umgebungen dynamisch erstellen und verwenden zu können.

Kapitel 4

Eigener Ansatz

Wie bereits mehrfach erwähnt wird als neuer Ansatz die Überlegung hergezogen, Game Engines als virtuelle Umgebungen aufzusetzen, um mit diesen sowohl Algorithmen direkt zu testen, als auch Ground Truth Daten zu generieren. Gleichzeitig wurde ein Framework entwickelt, das den Benutzer dabei unterstützt, in die Game Engine einzugreifen und gewünschten Daten zu erheben, sowie die Kommunikation zwischen der Game Engine und der Vision Engine zu übernehmen. Mehr zum Framework und dessen Implementierung befindet in Kapitel 5.

Ziel ist es, die schnelle Erstellung von virtuellen Umgebungen zum Testen von Computer Vision Algorithmen zu ermöglichen, ohne dass Umfangreiche Kenntnisse über Computergraphik vorhanden sein müssen. Außerdem soll über das Framework eine Austauschbarkeit von sowohl der Game Engine als auch der Vision Engine bereitgestellt werden.

Dieses Kapitel gibt einen Überblick über die wesentlichen Überlegungen. Zunächst wird ein allgemeiner Einblick in die Architektur gegeben und das Zusammenspiel zwischen Game Engine, Vision Engine und Framework erläutert. Des Weiteren wird noch auf die Komponente Game Engine genauer eingegangen. Es wird analysiert, warum es sinnvoll ist, eine Game Engine für diesen Zweck zu nutzen, welche Vorteile diese mit sich bringen und welche Anforderungen eine Game Engine erfüllen muss, um für solche Zwecke geeignet zu sein. Außerdem wird der Frage nachgegangen, welche bekannten Game Engines diese Anforderungen erfüllen und für diese Aufgabe in Frage kommen würden. Zum Abschluss wird beschrieben, welche Szenarien aus dem Bereich Computer Vision man mit diesem Framework abdecken könnte.

4.1 Architekturkonzept

In diesem Abschnitt wird erläutert, wie die Architektur aussieht und wie die einzelnen Teile zusammenspielen. Zunächst wird auf den grundlegenden

Aufbau und die einzelnen Komponenten, bestehend aus *Game Engine*, *Vision Engine* und einer *Schnittstelle*, eingegangen. Theoretischerweise sind die Game Engine und die Vision Engine austauschbar und können vom Benutzer des Frameworks frei gewählt werden, vorausgesetzt die Game Engine erfüllt die in Abschnitt 4.2.2 beschriebenen Anforderungen. Des Weiteren wird erläutert, für welche Zwecke das Framework schlussendlich genutzt werden kann. Eine genaue Beschreibung über die Implementierung und Verwendung des Frameworks befinden sich in Kapitel 5 und 6.

4.1.1 Grundlegender Aufbau

Wie bereits erwähnt, besteht das Framework aus drei eigenständigen Komponenten: die *Game Engine*, die *Vision Engine* und eine selbst entwickelte *Schnittstelle*. Jede dieser Komponenten läuft als eigenständiges Programm und übernimmt eine eigene Aufgabe.

Die Game Engine

Die Game Engine dient innerhalb des Frameworks als Datenlieferant. Sie stellt eine bestimmte Szene dar und soll die entsprechenden gerenderten Bilder und die zugehörigen Metainformationen liefern. Dazu soll die Engine fähig sein, zu einem beliebigen Zeitpunkt das von einer in der virtuellen Umgebung vorhandenen Kamera eingefangene Bild zu rendern und gemeinsam mit den entsprechenden Informationen über die Objekte in der Szene zu verschicken. Außerdem soll die Engine auf Kommandos reagieren und so von außen gesteuert werden können. Mehr zur Bedeutung von Game Engines innerhalb des Frameworks wird in Abschnitt 4.2 beschrieben.

Die Vision Engine

Die Vision Engine wird dann benötigt, wenn das Framework nicht nur zur reinen Generierung von Ground Truth Daten, sondern als Testumgebung für Computer Vision Algorithmen verwendet wird. Die Vision Engine beinhaltet hierbei den auszuführenden Algorithmus. Als Input wird ihr ein zu verarbeitendes Bild aus der Game Engine übertragen auf welchem der Algorithmus ausgeführt werden kann. Anhand des Ergebnisses kann dann ein Aktionskommando zurückgeschickt werden, wie sich die Game Engine verhalten soll.

Die Schnittstelle

Um die beiden Komponenten *Game Engine* und *Vision Engine* miteinander zu verbinden, wurde ein zusätzliches Programm entwickelt. Die Schnittstelle – der *Virtual Environment Supervisor* – steht zwischen den beiden anderen

Komponenten und übernimmt die Kommunikation und Steuerung. Er beinhaltet außerdem eine graphische Benutzeroberfläche, um dem Benutzer die Kontrolle über das Framework zu geben.

Kommunikation

Abbildung 4.1 zeigt den allgemeinen Ablauf innerhalb des Frameworks. Über die Schnittstelle hat man die Möglichkeit, ein Kommando an die Game Engine zu schicken, das ein gerendertes Bild bzw. die zugehörigen Metainformationen anfordert. Die Game Engine reagiert auf dieses Kommando und schickt die angeforderten Daten zurück. Die Schnittstelle hat dann zwei Möglichkeiten, mit diesen Daten fortzufahren:

- Wenn das Framework zu reinen Datengenerierung verwendet wird, werden das Bild und die Ground Truth Daten für eine spätere Verwendung einfach nur in einer Datenbank abgespeichert.
- Soll das Framework als virtuelle Testumgebung verwendet werden, wird das Bild zur sofortigen Verarbeitung an die Vision Engine weitergeschickt. Diese führt dann darauf den Algorithmus aus und definiert eine Aktion, wie sich die Game Engine anhand der Ergebnisse verhalten soll. Diese Aktion wird dann über die Schnittstelle wieder an die Game Engine weitergeleitet, wo eine bestimmte Reaktion ausgelöst wird.

4.1.2 Funktionsweisen

Durch den Aufbau des Frameworks ergeben sich also zwei primäre Funktionsweisen, für die das Framework genutzt werden kann.

Datengenerierung

Die Game Engine kann zur reinen Datengenerierung genutzt werden und eine Datenbank mit Bildern und zugehörigen Ground Truth Daten füllen, die dann später zur Evaluierung eines Algorithmus wiederverwendet werden können. Dazu werden von der Schnittstelle in regelmäßigen Abständen die Daten aus der Game Engine geholt und in einer Datenbank abgespeichert.

Virtuelle Testumgebung

Die Game Engine kann außerdem gemeinsam mit einer Vision Engine als Testumgebung für Computer Vision Algorithmen fungieren. Über eine definierte Aktion kann der Game Engine von außen übermittelt werden, wie sie sich anhand des in der Vision Engine errechneten Ergebnisses des Algorithmus verhalten soll.

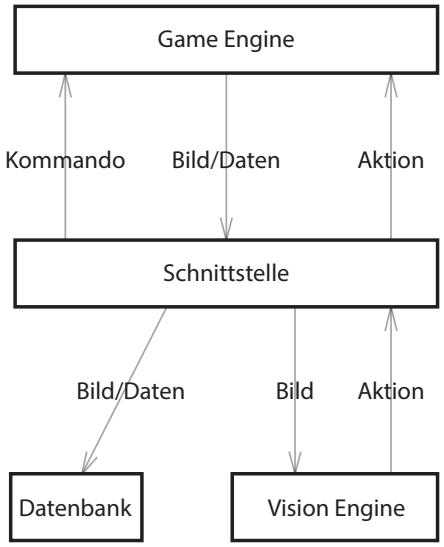


Abbildung 4.1: Allgemeiner Aufbau und Kommunikationsweise des Frameworks mit den drei Komponenten Game Engine, Vision Engine und Schnittstelle. Die zusätzlich eingezeichnete Komponente *Datenbank* ist nur ein Platzhalter für eine beliebige Form der Datenspeicherung und daher keine signifikante Komponente des Frameworks.

4.2 Game Engines

Nachdem in Abschnitt 4.1 erläutert wurde, wie das allgemeine Konzept und die einzelnen Komponenten aussehen, wird in diesem Abschnitt auf die Bedeutung von Game Engines für die Verwendung in diesem Ansatz eingegangen und gezeigt, warum es sinnvoll ist, Game Engines für diese Zwecke zu nutzen. Zunächst werden die Vorteile von Game Engines untersucht. Des Weiteren wird erforscht, welche Anforderungen an die Game Engine gestellt werden müssen, um in dieser Form genutzt werden zu können. Anhand dieser Anforderungen werden dann bekannte Game Engines analysiert.

4.2.1 Allgemeine Vorteile

In den letzten Jahren haben sich viele Firmen auf die umfangreiche Entwicklung von Game Engines spezialisiert. Dadurch sind mittlerweile viele sehr ausgereifte und relativ einfach zu bedienende Engines entstanden. Diese bringen einige Vorteile mit sich, um sie zur Erstellung von virtuellen Umgebungen verwenden zu können.

Game Engines sind darauf ausgelegt, schnell und dynamisch verschiedenste Szenerien zu erstellen und bieten sehr ausgereifte und weit entwickel-

te computergraphische Implementierungen. Sie sind meist sehr allgemein gehalten und nicht auf ein bestimmtes Szenario begrenzt. Das bedeutet, dass eine sehr hohe Anpassungsfähigkeit an die eigenen Bedürfnisse vorhanden ist. Des Weiteren sind für das Nutzen einer Game Engine wenige oder gar keine computergraphischen Kenntnisse notwendig. Es müssen keine Lichtberechnungen, Rendereinstellungen oder Shaderprogrammierungen selbst vorgenommen werden. Oft sind für das erfolgreiche Arbeiten mit einer Game Engine nicht einmal Programmierkenntnisse gefordert, dennoch können virtuelle Umgebungen verschiedenster Art meist sehr schnell erstellt werden. Des Weiteren sind oft die für diesen Ansatz so wichtigen Metainformationen über die in der Szene enthaltenen Objekte bereits in der Game Engine vorhanden.

4.2.2 Anforderungen an Game Engines

Nun kann nicht jede beliebige Game Engine hergenommen werden, um diesen Ansatz zu verfolgen. Je nach verfolgtem Ziel muss auf gewisse Parameter der Engine geachtet werden, damit mit ihr in diese Richtung gearbeitet werden kann. Folgende Funktionalitäten einer Engine könnten ausschlaggebend für dessen Verwendbarkeit sein:

Szenentyp: Manche Game Engines sind auf ein gewisses Genre oder einen bestimmten Szenentyp ausgelegt. Die Wahl der Game Engine sollte je nach Darstellungswunsch erfolgen. Typischerweise arbeitet Computer Vision mit dreidimensionalen Bildern, daher sind Game Engines, die auf 2D ausgelegt sind, ebenso wie Engines, die sich im Science-Fiction Bereich oder einem anderen ausgefallenen Genre bewegen, nicht optimal.

Realismus: Die Game Engine sollte einen möglichst realistischen Output haben, da zumeist Computer Vision Algorithmen getestet werden, die später auch in der realen Welt funktionieren sollen. Game Engines, die beispielsweise auf das Rendern eines Comic-Stils ausgelegt sind oder auf abstrakte Welten spezialisiert sind, machen für dieses Unterfangen keinen Sinn. Natürlich unterscheidet sich der Grad an Realismus auch je nach dem Algorithmus, den man testen möchte. Zu erkennende Stoppschilder sind in einer virtuellen Umgebung wesentlich einfacher zu erstellen als z. B. menschliche Gesichter für einen Gesichtserkennungsalgorithmus.

Ground Truth Daten: Es ist wichtig, dass bei der Game Engine die internen Daten verfügbar sind. Es sollte möglich sein, auf alle in der Szene enthaltenen Objekte und deren Eigenschaften zuzugreifen, um sie dann nach außen kommunizieren zu können. Dazu zählen Eigenschaften wie Position, Typ, Größe, Farbe, Oberfläche oder Struktur. Bei Engines, wo die Verwaltung der Objekte nicht von der Engine selbst übernommen wird, sollte man

bei der Erstellung der virtuellen Umgebung darauf achten, dies irgendwo im Programm selbst zu implementieren.

Erweiterungssprache: Gerade für die Nutzung des entwickelten Frameworks (in Kapitel 6 beschrieben) ist es sehr wichtig, dass eine Engine native Sprachen wie beispielsweise C++, C# oder Java unterstützt. Nur mit so einer Erweiterung kann das Framework in die Engine eingebunden werden. Außerdem kann über diese Sprachen die Engine erweitert werden und es können zusätzliche Funktionalitäten, die in der Engine eventuell nicht vorhanden sind, dazu implementiert werden.

Zusätzliche Daten: Je nach Algorithmus könnten auch weitere umfangreichere Daten benötigt werden wie beispielsweise eine *Depth-Map* – ein Bild, das Informationen über die Distanz von Szenenobjekten zu einen Blickpunkt enthält[1] – oder eine Stereoaufnahme der Szene, die von vielen Computer Vision Algorithmen benötigt wird. Dazu sollte die Game Engine mit mehreren Kameras umgehen können.

Raycasting: Oft wird versucht herauszufinden, ob ein Objekt von der Kamera gesehen wird, oder verdeckt ist. Dazu verwendet man oft Techniken wie beispielsweise *Raycasting*, bei der die Fortbewegung eines Strahls von der Kamera aus durch die Szene verfolgt wird und durch Treffer auf den Szenenobjekten bestimmt werden kann, welches Objekt am nächsten zur Kamera ist. Es ist ein großer Vorteil, wenn die Engine solche Funktionalitäten bereits eingebaut hat.

Geschwindigkeit: Meistens arbeiten Computer Vision Algorithmen nicht in Echtzeit, d. h. der Algorithmus braucht für die Berechnung länger als die Zeit, die es benötigt, das nächste Bild nachzuliefern. Daher wäre es von Vorteil, wenn man die Game Engine zu beliebigen Zeitpunkten anhalten könnte, z. B. während der Algorithmus rechnet. Außerdem kann es sein, dass man die Engine auch zurückspulen und verlangsamen können sollte, um eine Kontrolle des Programmflusses beibehalten zu können.

Physik: Die Simulation von physikalischen Vorgängen kann ausschlaggebend für die sinnvolle Verwendung sein und sollte möglichst exakte Berechnungen liefern. Vor allem die Wirkung von Kräften (Schwerkraft) sowie die Kollisionserkennung sollte möglichst realitätsnah sein, um eine glaubwürdige Simulation herstellen zu können.

[1] http://en.wikipedia.org/wiki/Depth_map

API: Eine Schnittstelle für den Zugriff auf interne Methoden und Funktionen der Game Engine kann ein großer Vorteil sein, um beispielsweise besser auf die internen Daten zugreifen zu können.

Netzwerkkommunikation: Für das Framework ist es wichtig, dass die Game Engine in irgendeiner Form nach außen kommunizieren kann, damit die Daten verschickt werden können. Dies könnte in Form von eingebauten Komponenten oder durch die Anbindung von verbreiteten Programmiersprachen wie C#, C++ oder Java erfolgen. Nicht gut möglich ist eine umfangreiche Netzwerkkommunikation, wenn die Game Engine nur mit eigenen Script-Sprachen arbeitet, wo solche Features nicht unterstützt werden, oder gar keine eigene Programmierung erlaubt.

Editor: Viele Game Engines bieten mittlerweile sehr umfangreiche und gut ausgebaute Editoren, um Szenen generieren zu können. Solche erleichtern natürlich die Erstellung von virtuellen Umgebungen erheblich und reduzieren die erforderlichen Programmierkenntnisse.

Levelgröße: Die Fähigkeit der Engine, sehr große Szenerien zu bauen, falls solche notwendig sind, kann ebenfalls ausschlaggebend sein.

Einarbeitungszeit: Nicht unwesentlich ist das Maß an Schwierigkeit und die Einarbeitungszeit, um mit der Game Engine befriedigende Ergebnisse erzielen zu können.

4.2.3 Analyse verschiedener Game Engines

In Abschnitt 2.3 wurden die verschiedene Arten von Game Engines vorgestellt und eine allgemeine Einteilung vorgenommen. Nun wird analysiert, welche dieser Game Engines sich anhand der in Abschnitt 4.2.2 angeführten Anforderungen überhaupt für diese Aufgabe eignen würden. Dazu wird auf jede Engine allgemein eingegangen. Tabelle 4.1 zeigt außerdem eine tabellarische Zusammenfassung der wichtigsten Funktionen der verschiedenen Engines.

Application Programming Interfaces

APIs wie beispielsweise *XNA*, *OpenGL* oder *DirectX* (werden in Abschnitt 2.3.3 genauer beschrieben) sind zwar keine Game Engines im herkömmlichen Sinne, eignen sich aber dennoch prinzipiell ebenso für die Erstellung virtueller Umgebungen. Sie sind auf allgemeine Programmiersprachen wie *C#* und *C++* aufgebaut und bieten daher einen hohen Grad an Anpassungsfähigkeit. Die eigenen spezifischen Bedürfnisse können sehr leicht abgedeckt werden und der Zugriff auf die internen Daten ist kein Problem. Der große

Tabelle 4.1: Analyse von beliebten Game Engines anhand einiger ausgewählter Anforderungen. Die Farben sind nach Nutzbarkeit für Computer Vision Testumgebungen gewählt. Am besten für die Verwendung eignen sich die Engines *Unity3D* und *CryEngine*, da diese die meisten Anforderungen erfüllen. Am ungünstigsten ist die Verwendung von *Slick2D* oder der *Unreal Engine*, da diese entweder vom Szenentyp nicht passen bzw. zu eingeschränkt sind. Schnittstellen, wie z. B. *XNA*, *DirectX* oder *OpenGL* eignen sich prinzipiell sehr gut, haben aber den großen Nachteil, dass kein Editor vorhanden ist und sehr viele Funktionalitäten selbst implementiert werden müssen.

	Schnitt-stellen	Slick2D	CryEngine	Unreal Engine	Unity3D
Szenentyp	3D	2D	3D	3D	3D
Potenzieller Realismus	Hoch	Gering	Hoch	Hoch	Hoch
Ground Truth	Ja	Ja	Ja	Ja	Ja
Erweiter-ungs-sprachen	C++, C#	Java	C++	Keine	JavaScript, C#
Depth-Map	Implemen-tierbar	Implemen-tierbar	Ja	Nein	Implemen-tierbar
Raycasting	Implemen-tierbar	Implemen-tierbar	Ja	Implemen-tierbar	Ja
Geschwin-digkeit	Implemen-tierbar	Implemen-tierbar	Ja	Ja	Ja
Physik	Nein	Nein	Ja	Ja	Ja
API	Ja	Ja	Ja	Ja	Ja
Netzwerk	Implemen-tierbar	Implemen-tierbar	Implemen-tierbar	Nein	Implemen-tierbar
Stereo	Ja	Ja	Ja	Ja	Ja
Editor	Nein	Nein	Ja	Ja	Ja
Einarbei-tungszeit	Mittel	Gering	Hoch	Hoch	Mittel

Nachteil bei der Verwendung dieser Interfaces ist jedoch dass diese Daten selbst erstellt und verwaltet werden müssen. Außerdem liegt hier ein hoher Aufwand an Selbstprogrammierung vor, da diese Interfaces keinen Editor zur Verfügung stellen und nur die Basisfunktionalitäten wie beispielsweise das Rendering übernehmen. Das Erstellen einer kompletten virtuellen Umgebung mit möglichst hohem Realismus ist hier im Gegensatz zu anderen Arten von Engines um einiges zeitaufwendiger und verlangt ein hohes Maß an Programmierkenntnissen. Funktionen zum Erstellen einer Depth-Map, zur Verwendung von Raytracing, zur Geschwindigkeitskontrolle, für eine Netzwerkkommunikation oder zum Simulieren von physikalischen Vorgängen müssen komplett selbst implementiert werden oder durch zusätzliche Libraries abgedeckt werden.

Slick2D

Die *Slick2D* ist eine Library, die schon einige eingebaute Komponenten, wie Rendering, Animationen oder ein Partikelsystem beinhaltet. Durch die Verwendung von *Java* besteht auch hier die gewünschte hohe Anpassungsfähigkeit und Erweiterbarkeit. Dennoch ist auch hier ein relativ hoher Programmieraufwand notwendig, um die internen Daten zu erstellen und zu verwalten, sowie um eine umfangreiche virtuelle Umgebung erstellen zu können. Außerdem ist dies eine 2D Engine und daher wahrscheinlich für viele Computer Vision Anwendungen nicht brauchbar und der potentiell erreichbare Realismus ist im Gegensatz zu anderen Engines um einiges geringer.

CryEngine

Diese hauseigene Game Engine der Firma *Crytek*[2] ist eine der umfangreichsten aktuellen Game Engines. Sie bietet hochkomplexe computergraphischen Berechnungen und kommt schon sehr nah an eine realistische dreidimensionale Darstellung heran. Mit einem ausgereiften Editor können über graphische Benutzeroberflächen eigene Szenerien erstellt werden. Abbildung 4.2 zeigt den Editor mit einer Demo-Szene. Über eine C++ Anbindung kann auf die internen Daten zugegriffen und das Programm erweitert werden. Physikalische Berechnungen, Depth-Map-Erstellung, Raytracing und Geschwindigkeitskontrolle sind in die Engine integriert. Die Verwendung von mehreren Kameras erlaubt die Erstellung von Stereo-Aufnahmen. Allerdings ist der Editor sehr komplex und das Arbeiten mit dieser Engine verlangt eine lange Einarbeitungszeit.

Unreal Engine

Die *Unreal Engine* hat mit ihrem Level Editor *Unreal Development Kit* ebenfalls ein sehr ausgeprägtes Angebot um virtuelle Umgebungen zu erstellen. Programmieren mit C++ ist hier allerdings nur mit einer Lizenz möglich, ansonsten ist die Engine-spezifische Skriptsprache *UnrealScript* zu verwenden, welches zwar auch Netzwerkkommunikationsmöglichkeiten bietet, jedoch für das Framework nicht optimal ist, da mit der Verwendung dieser Sprache keine Austauschbarkeit mehr möglich wäre.

Unity3D

Unity3D bietet ebenfalls einen Editor, der ein schnelles Erstellen virtueller Umgebungen ermöglicht. Die Engine ist auf dreidimensionale Szenen mit hohem Realismus ausgelegt, kann aber durchaus auch für die Erstellung von 2D-Spielen verwendet werden. Sie stellt eine gute Mischung aus vorgefertigten Komponenten und Integrierung eigener Programmstücke in den Spra-

[2] http://www.crytek.com/

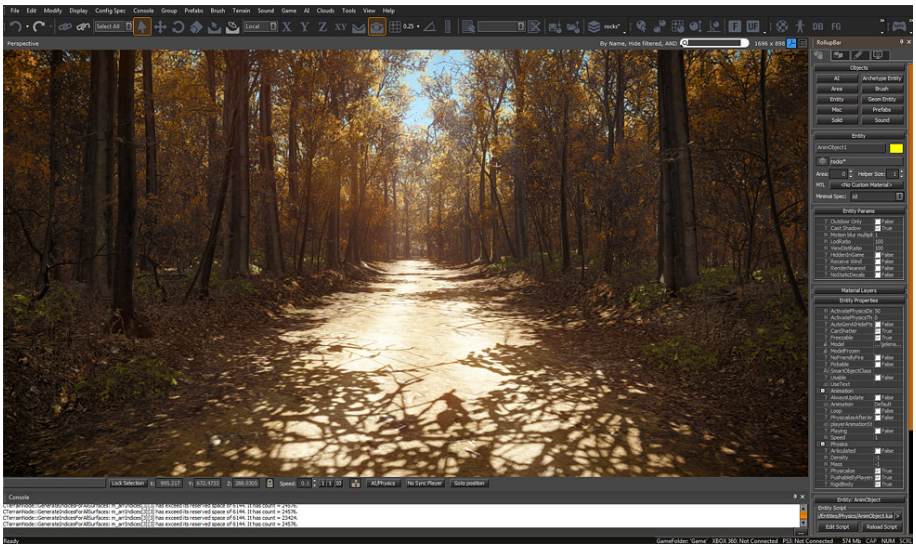


Abbildung 4.2: Ein Screenshot des Editors der *Cryengine* mit einer Demo-Szene [30]. Die Engine erzeugt ein hohes Maß an Realität, ist aber nicht einfach zu bedienen.

chen *JavaScript*, *C#* und der hauseigenen Scriptsprache *Boo* zur Verfügung, mit denen man die Applikation beliebig erweitern, sowie auf die API der Engine und deren internen Daten zugreifen kann. Für zusätzliche Funktionalitäten wie *Raytracing*, *Stereo* oder das Erstellen einer *Depth-Map* stehen vorgefertigte Funktionen zur Verfügung. Ebenfalls integriert sind physikalische Berechnungen, Texturen, Kollisionserkennungen und Partikelsysteme. Auch die Geschwindigkeit der Simulation kann über die API gesteuert werden. *Unity3D* bietet eine gute Balance zwischen Leveleditor und eigener Programmierung und ist außerdem relativ einfach zu erlernen. *Unity3D* wurde wegen dieser Gründe als Hauptengine für die Beispielimplementierung verwendet. Mehr dazu in Kapitel 5.

4.3 Beispielszenarien

Nun stellt sich die Frage, in welchen Fällen man Game Engines in Verbindung mit dem in Abschnitt 4.1 beschriebenen Konzept verwenden könnte. Es gibt verschiedene Szenarien, die durch dieses Konzept realisiert werden können. Dieser Abschnitt gibt einen Überblick über eine mögliche Einteilung der potenziellen Szenarien sowie ein paar dazu passende Beispiele. Um so eine Einteilung zu bewerkstelligen, wurden drei Basisszenarien erstellt: *Passive Agent*, *Invasive Agent* und *Interactive Agent*. Diese sind i. Allg. sehr ähnlich, unterscheiden sich aber durch ihr übergeordnetes Ziel, durch ihre abgedeckten Bereiche und durch ihr verschiedenes Verhalten.

- *Passive Agent:* Virtuelle Umgebungen können dazu genutzt werden, eine Szenerie nachzubilden und über Simulationen und Visualisierungen zu veranschaulichen. Dadurch können Schwierigkeiten und Besonderheiten von bestimmten Situationen aufgezeigt werden. Durch die Verfügbarkeit von einstellbaren Parametern können komplexe Situationen unter verschiedenen Bedingungen simuliert werden. Diese Simulationen dienen sozusagen zur Veranschaulichung von Ausnahmesituationen. In diesem Szenario werden weder Computer Vision noch eine Vision Engine eingesetzt und auch noch keine zusätzlichen Daten generiert. Zu solchen Situationen zählen beispielsweise die Simulation von Verkehrssituationen, das Nachbilden von Terrains oder das Darstellen von Flugrouten.
- *Invasive Agent:* Mit Game Engines können Daten für Computer Vision Algorithmen erzeugt werden, welche darauf ausgelegt sind, die Szene zu analysieren und Informationen daraus zu extrahieren. Die virtuelle Umgebung stellt eine gewisse Szene dar und liefert die Sensortaten, z. B. Bilder, auf die die Algorithmen angewandt werden können. Um diese Algorithmen dann zu evaluieren ist es ebenfalls notwendig, in die Game Engine einzugreifen und die notwendigen Ground Truth Daten zu erheben, also die Daten, die die Szene in der Engine beschreiben. Eingeteilt wird dieses Szenario in zwei Hauptaufgaben, die diese Algorithmen verfolgen:
  - *Object reconstruction:* Man möchte Informationen über die in der Szene enthaltenen Objekte extrahieren. Dazu zählen Algorithmen zur Objekterkennung, Einteilung oder Positionsbestimmung. Ein Beispiel hierfür ist die Videoanalyse beim Sport, wo man etwa die Anzahl der gespielten Pässe bei einem Fußballspiel automatisch ermitteln möchte, oder die Sicherheits-und Überwachungsindustrie, in der Personen identifiziert, sowie Fingerabdrücke erkannt werden sollen. Ein anderes Beispiel ist die automatische Erkennung von Schildern auf der Straße oder von Autokennzeichen mit Straßenkameras.
  - *Szene reconstruction:* Auch die Rekonstruktion von Informationen über die gesamte Szene kann interessant sein. Dazu gehören Gebiete wie 3D-Rekonstruktion, Tiefenerkennung, Selbst-Lokalisierung (SLAM[3]) oder Segmentierung. Wenn beispielsweise ein Roboter seine Umgebung wahrnehmen will, müssen Informationen über die Raumtiefe und -struktur extrahiert, sowie die momentane Position abgeschätzt werden. Wenn man auf Satellitenbildern automatisch Vegetation (z. B. Wald) erkennen möchte, müssen die Bilder zunächst möglichst genau segmentiert werden.

[3]Simultaneous Localization and Mapping

- *Interactive Agent:* Ein Szenario dieser Art ist eine Erweiterung des *Invasive Agent*. Zusätzlich zur Analyse der Szene möchte man die virtuelle Umgebung steuern und diese verändern, um die Simulation beeinflussen zu können. Es wird sozusagen eine geschlossene, sich wiederholende Schleife eingesetzt, in der auf die in einer Vision Engine durchgeführten Analyse des Outputs der Game Engine eine entsprechende Reaktion folgt, die dann die Parameter und Bedingungen innerhalb der Game Engine verändert und anschließend erneut eine Analyse durchführt. Einer der Bereiche, in dem man dieses Szenario oft einsetzen könnte ist das automatische Fahrverhalten von Fahrzeugen. Zum Beispiel soll abgebremst werden, wenn die Analyse ergibt, dass man zu nah an einem Hindernis ist, oder die Lenkung soll automatisch anhand des erkannten Straßenverlaufs erfolgen. Dann wird anhand des Ergebnisses der Analyse der Game Engine gesagt, wie sie sich verhalten soll. Außerdem kann man in diesem Szenario z. B. die Entscheidungsmuster von intelligenten Agenten simulieren, indem die auf Computer Vision basierenden Entscheidungen von den Agenten in der virtuellen Umgebung auch sofort ausgeführt werden. Beispielsweise könnten Agenten Schilder wahrnehmen und dann über eine Textanalyse Entscheidungen zur Navigation treffen.

Für jedes dieser Szenarios ist es durchaus sinnvoll, eine Game Engine zu verwenden. Zum ausschließlichem Nachbilden und Simulieren einer Szene sind nicht einmal zusätzliche Komponenten notwendig. Um die etwas komplexeren Szenarios der Analyse und der aktiven Kontrolle besser durchführen zu können, unterstützt das in Kapitel 5 beschriebene Framework den Benutzer dabei, diese Aufgaben möglichst effizient durchführen zu können.

Kapitel 5

Implementierung

Um den in Kapitel 4 beschriebenen Ansatz zu realisieren, wurde *Virtual Environment Controlling Application* (VECA) entwickelt – ein Framework, das den Benutzer dabei unterstützt, eine Game Engine als virtuelle Umgebung aufzusetzen, um einen Computer Vision Algorithmus testen zu können. Dadurch sollen vor allem die im vorherigen Kapitel (Abschnitt 4.3) beschriebenen Szenarien *Invasive Agent* und *Interactive Agent* unterstützt werden. Sprich, die Game Engine soll dazu genutzt werden, sowohl für Computer Vision notwendige Daten (Bilder und Ground Truth) zu generieren, damit diese später wiederverwendet werden können, als auch um als virtuelle Testumgebung zu fungieren, die von einer Vision Engine gesteuert werden kann.

Im Vordergrund bei der Implementierung standen die möglichst einfache und schnelle Nutzungsmöglichkeit, sowie die Gewährleistung der hohen Austauschbarkeit der einzelnen Komponenten. Das Framework kann durch den objektorientierten Aufbau sehr schnell in die eigene Applikation integriert und verwendet werden.

In diesem Kapitel wird zuerst der allgemeine Aufbau, sowie der schrittweise Ablauf des Zusammenspiels der Komponenten erläutert. Dann werden die Implementierungen der einzelnen Komponenten genauer beschrieben. Zunächst die Implementierung der Library für die Komponenten Game Engine und Vision Engine, danach der Aufbau und die Funktionsweise der Schnittstelle – des *Virtual Environment Supervisor*. Die konkrete Verwendung und Einbindung des Frameworks in ein eigenes Projekt wird anhand eines Anwendungsbeispiels in Kapitel 6 erklärt.

5.1 Aufbau

VECA wurde in C# entwickelt und besteht aus zwei Hauptteilen. Der erste Teil ist eine Library, die sowohl in die Game Engine als auch in die Vision Engine eingebunden werden kann. Dazu müssen diese Engines vom

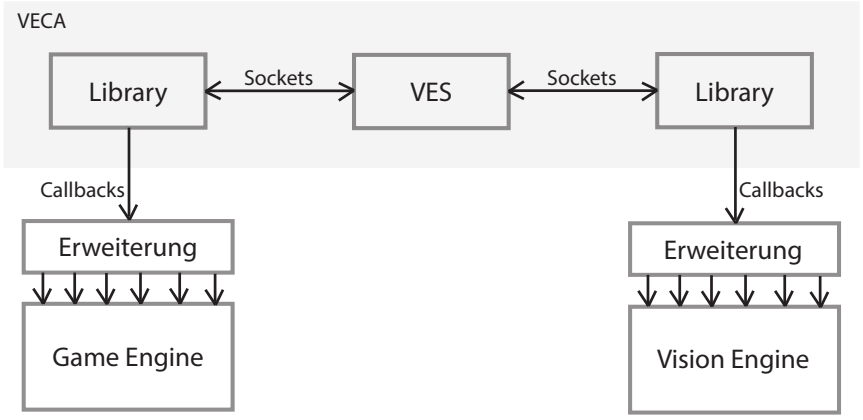


Abbildung 5.1: Allgemeiner Aufbau der Komponenten. Der grau unterlegte Bereich ist jener, den VECA zur Verfügung stellt. Die Erweiterungen zu den Engines sind Codestücke, die die Library einbinden. Diese müssen vom Benutzer erstellt werden. Diese werden mit den Engines mit kompiliert und vollziehen die konkreten Zugriffe auf die Engines. Die Schnittstelle kommuniziert dann mit den Libraries und kann so die Engines steuern.

Benutzer mit Hilfe von zusätzlichem Code erweitert werden, der mit den Engines mit kompiliert wird. Dieser Code muss dann die Library inkludieren (Abbildung 5.1). Die Library selbst kapselt allgemein benötigte Funktionalitäten, wie z. B. die Netzwerkkommunikation über Sockets, und stellt Callback-Funktionen zur Verfügung. Diese Callback-Funktionen sollen die Engine-spezifischen Implementierungen, wie z. B. den Zugriff auf die internen Daten der Game Engine, beinhalten und werden daher nicht von der Library implementiert sondern müssen vom Benutzer im Erweiterungscode der Engine implementiert werden. Die Callback-Funktionen werden dann zu gewissen Zeitpunkten vom Framework automatisch aufgerufen.

Der zweite Teil ist der *Virtual Environment Supervisor* (VES) – ein eigenständiges Programm, das als Schnittstelle zwischen den beiden anderen Komponenten dient und die beiden Engines miteinander verbindet, indem er mit der in den Engines eingebundenen Library über Sockets kommuniziert. Über diese Socketverbindungen werden Kommandos an die Library gesendet, die dann entsprechend darauf reagieren und gegebenenfalls die entsprechenden Callback-Funktionen aufrufen kann. Die erhobenen Daten werden dann wieder über den Socket an die Schnittstelle gesendet, die diese weiterverarbeitet. Der VES bietet außerdem eine graphische Benutzeroberfläche, um das Framework steuern zu können.

Schrittweiser Ablauf

Mit dem in Abbildung 5.1 gezeigten Aufbau des Frameworks kann der VES über Socketverbindungen Kommandos an die in die Engines eingebundene Library schicken, wo dann die entsprechenden, vom Benutzer in der Erweiterung implementierten Callback-Funktionen aufgerufen werden. Ein typischer Ablauf innerhalb des Frameworks, wenn man dieses als virtuelle Testumgebung verwenden möchte, sieht so aus:

1. Der VES schickt das Request-Kommando für ein Bild an die Library der Game Engine.
2. Die Library ruft daraufhin die Callback-Funktion der Erweiterung auf, in der die Implementierung des Benutzers das gewünschte Bild erstellt und zurückgibt. Die Implementierung für die Erstellung eines Bildes ist je nach Game Engine unterschiedlich und muss daher vom Benutzer geschrieben werden.
3. Das Bild wird dann von der Library über die Socketverbindung zurück an den VES gesendet.
4. Der VES leitet das Bild über die zweite Socketverbindung an die in die Vision Engine eingebundene Library weiter.
5. Dort wird abermals die vom Benutzer implementierte Callback-Funktion aufgerufen, an die das Bild übergeben wird und die den Computer Vision Algorithmus enthalten soll.
6. In dieser Callback-Funktion soll dann der Algorithmus ausgeführt werden und anhand des Ergebnisses eine Aktion definiert werden, wie sich die Game Engine verhalten soll. Wie diese Aktion genau aussieht, kann der Benutzer frei wählen.
7. Die Aktion wird dann wieder über die Socketverbindungen über den VES zurück an die Game Engine geleitet, wo die Library abermals eine Callback-Funktion aufruft und dieser die Aktion mitgibt, damit dort der Benutzer definieren kann, wie die Game Engine darauf reagiert.

Somit kann die Game Engine anhand des Ergebnisses des in der Vision Engine enthaltenen Computer Vision Algorithmus gesteuert werden.

Falls man das Framework zur reinen Datengenerierung verwenden möchte, werden die ersten drei Schritte sowohl mit dem Kommando für ein Bild, als auch mit jenem für die Ground Truth-Daten ausgeführt. Diese werden dann vom VES nur noch abgespeichert.

5.2 Die Library – Game Engine

Die Library-Teil von VECA soll es den beiden Engines ermöglichen, die notwendigen Daten zu erheben und zum VES zu kommunizieren. Dazu müssen diese im Erweiterungscode der Engines verwendet werden. In diesem

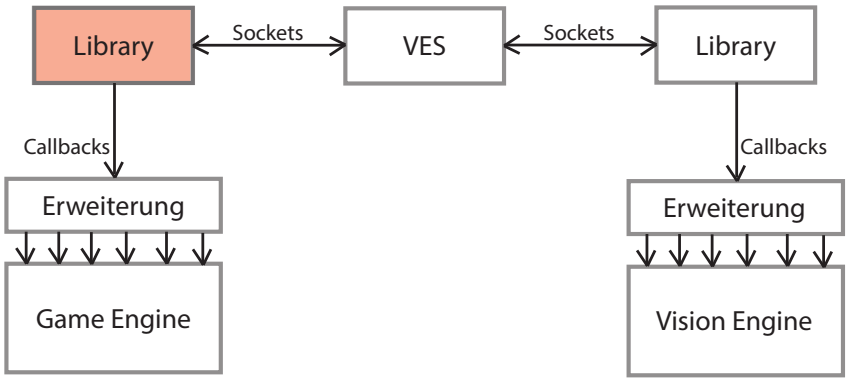


Abbildung 5.2: Dieser Abschnitt behandelt die Implementierung der Library für die Game Engine.

Abschnitt wird die Implementierung des Game Engine-Teils der Library (Abbildung 5.2) erläutert. Diese stellt hauptsächlich die Austauschbarkeit der Engine in den Vordergrund. Der Benutzer soll die verwendete Engine frei wählen können und trotzdem das Framework noch nutzen können. Die Game Engine selbst soll durch die Einbindung der Library fähig sein, auf die visuellen Daten sowie die internen Metainformationen zugreifen und diese verschicken zu können. Die Netzwerkkommunikation zum VES wird hierbei von der Library übernommen, sämtliche Game Engine-spezifischen Funktionalitäten, wie beispielsweise Datenzugriff, Pausieren, Zurücksetzen etc. müssen vom Benutzer über Callback-Funktionen im Erweiterungscode implementiert werden, da diese Implementierungen je nach Game Engine unterschiedlich sind.

5.2.1 Klassenstruktur

Die Hauptklasse der Library für die Verwendung in der Game Engine ist der `GameEngineController`. Dieser kümmert sich um den allgemeinen Ablauf des Frameworks innerhalb der Engine, enthält die Interpretation von empfangenen Daten und stellt die Callback-Funktionen bereit. Diese Klasse ist jene, von der der Benutzer innerhalb der Erweiterung ableiten muss. Sie verwendet noch einige weitere implementierte Hilfsklassen, wie beispielsweise die Klasse `SocketServer`, die den eigentlichen Umgang mit Sockets und Verbindungen enthält, aber sehr allgemein gehalten ist und sich daher nur um das Senden und Empfangen von Daten kümmert, aber nicht um deren Interpretation. Folgende Methoden und Funktionsweisen sind in der Klasse `SocketServer` enthalten:

- `SocketServer(int port)`

Konstruktor der Klasse `SocketServer`. Hier werden die für den Socket notwendigen Objekte initialisiert und der `Port` mitgegeben. Dieser beschreibt die Portnummer, an der auf eingehende Verbindungen gehorcht werden soll. Die Library für die Game Engine verwendet einen anderen Port als jene für die Vision Engine, damit sich der VES zu beiden gleichzeitig verbinden kann.

- `void start()`
  Diese Methode startet den eigentlichen Server. Das Programm beginnt hier über ein `TcpListener`-Objekt auf eingehende Verbindungen zu horchen. Damit die Applikation nicht stecken bleibt, während man auf einen Client wartet, wird diese Operation asynchron ausgeführt. D. h. es wird in einem eigenen Thread auf eingehende Verbindungen gewartet, während das Hauptprogramm weiterlaufen kann. Sobald eine Verbindung eingeht, wird die interne Funktion `acceptTcpClientCallback(IAsyncResult ar)` aufgerufen.
- `void acceptTcpClientCallback(IAsyncResult result)`
  Diese Funktion wird aufgerufen, sobald eine Clientverbindung akzeptiert wurde. Hier wird das entsprechende `TcpClient`-Objekt erstellt, um die Verbindung in weiterer Folge steuern zu können. Außerdem wird ein zugehöriger `NetworkStream` erstellt, um Daten über die Verbindung schicken und empfangen zu können.
- `void writeData(byte[] data)`
  Diese Methode wird nun für das eigentliche Senden von Daten verwendet. Ein byte-Array von beliebiger Größe wird auf den `NetworkStream` geschrieben.
- `byte[] readData(int buffersize)`
  Mit dieser Methode werden von der Clientverbindung Daten gelesen und als byte-Array zurückgeliefert. Mit dem Parameter `buffersize` kann angegeben werden, wie groß der Buffer für den jeweiligen Leseaufruf sein soll.
- `void close()`
  Schließt die Verbindung und stoppt den Listener.
- `bool isRunning()`
  Diese Hilfsmethode gibt an, ob gerade eine Verbindung besteht.

Der `GameEngineController` verwendet nun diese Funktionalitäten, um den Ablauf und den Datenaustausch durchführen zu können. Zu Beginn wird ein Objekt des Typs `SocketServer` erstellt. Über ein implementiertes Eventwird dann der `GameEngineController` informiert, wenn eine Verbindung hergestellt wurde. In weiterer Folge werden in einer Endlosschleife über die Methode `SocketServer.readData(32)` eingehende Daten abgefragt. Falls Daten ankommen, werden diese über ein Kommandosystem interpretiert.

5.2.2 Kommandosystem

Damit die Library innerhalb der Game Engine auf eingehende Anfragen reagieren kann, wurde ein Kommandosystem entwickelt. Über die Socketverbindung in der Klasse `SocketServer` kann der Game Engine ein bestimmtes Kommando geschickt werden, die dieses interpretiert und anschließend eine entsprechende Aktion ausführt. Auf folgende Kommandos kann das Framework derzeit reagieren:

- `GET IMAGE` – Wird dieses Kommando empfangen, soll der aktuelle Render-Output zurückgesendet werden.
- `GET OBJECTS` – Dieses Kommando bezeichnet die Anfrage nach den Ground Truth-Daten der Objekte. In Folge dessen sollen die aktuellen Metainformationen über die in der Szene enthaltenen Objekte zurückgeschickt werden.
- `PAUSE` – Die Game Engine soll angehalten werden.
- `UNPAUSE` – Fortfahren der Game Engine.
- `RESET` – Zurücksetzen der Szene.
- `CLOSE` – Schließen der Verbindungen.
- `PING SERVER` – Beim Empfang dieses Kommandos soll nur die Verbindung überprüft und der Request-Response-Ablauf getestet werden. Daher wird das empfangene Kommando einfach über den Socket wieder zurückgeschickt.
- Falls keines dieser Kommandos zutrifft, wird angenommen, dass der Game Engine eine auszuführende Aktion von der Vision Engine übermittelt wurde. Dieses kann verschiedenstes Aussehen haben. Beispielsweise eine Zahl oder ein vom Benutzer definiertes zusätzliches Kommando.

5.2.3 Callbacks

Das Framework reagiert nun auf zweifache Weise auf diese Kommandos. Einerseits werden interne Mechanismen der Library ausgelöst, z. B. das Schließen des Sockets, andererseits müssen Game Engine-spezifische Funktionalitäten ausgeführt werden, z. B. das Pausieren der Szene. Diese spezifischen Funktionalitäten müssen vom Benutzer in der Erweiterung implementiert werden. Dazu stellt die Klasse `GameEngineController` folgende abstrakte Methoden zur Verfügung, die beim Empfang von Kommandos entsprechend aufgerufen werden:

```
protected abstract byte[] getImageData();
protected abstract byte[] getObjectData();
protected abstract void pauseGameEngine();
protected abstract void unpauseGameEngine();
protected abstract void resetEngine();
protected abstract void performAction(string action);
```

Diese Callback-Funktionen müssen vom Benutzer in der Erweiterung in der abgeleiteten Klasse implementiert werden. Je nach eingehendem Kommando wird dann sowohl eine Library-interne Funktionalität durchgeführt, als auch die entsprechende Callback-Funktion aufgerufen, die den Code des Benutzers ausführt. Folgende Methode im `GameEngineController` implementiert die Interpretation von eingehenden Kommandos[1]:

```
1  private void handleCommand(string command)
2  {
3    switch (command)
4    {
5      case "GET OBJECTS":
6        byte[] msg = getObjectData();
7        socketServer.writeData(msg);
8        break;
9      case "GET IMAGE":
10       byte[] img = getImageData();
11       socketServer.writeData(img);
12       break;
13     case "PAUSE":
14       pauseGameEngine();
15       break;
16     case "UNPAUSE":
17       unpauseGameEngine();
18       break;
19     case "RESET":
20       resetEngine();
21       break;
22     case "CLOSE":
23       socketServer.close();
24       break;
25     case "PING SERVER":
26       socketServer.writeData(Encoding.ASCII.GetBytes(command));
27       break;
28     default:
29       performAction(command);
30       break;
31   }
32 }
```

In den Zeilen 6, 10, 14, 17, 20 und 29 werden die Callback-Funktionen aufgerufen, die in der Unterklasse implementiert sein sollten. Diese decken jenen Bereich der Implementierung ab, der je nach verwendeter Game Engine unterschiedlich ist. Ein Beispiel für die Verwendung des Frameworks sowie eine mögliche Implementierung der Callback-Funktionen befindet sich in Kapitel 6.

[1] Diese Methode dient zur Veranschaulichung des Ablaufs und befindet sich in dieser Form nicht mehr in der Library, da die Handhabung der Kommandos über die Verwendung von Design-Patterns anders implementiert wurde. Mehr dazu in Abschnitt 5.2.4

5.2.4 Design-Patterns

Um die Implementierung und die Handhabung des Kommandosystems einfacher und dynamischer zu gestalten, wurde dieses mit Hilfe von zwei Design-Patterns erstellt. Dadurch ergibt sich eine stark verbesserte Erweiterbarkeit und Übersichtlichkeit des Systems.

Command-Pattern

Das sogenannte *Command-Pattern* ist ein weit verbreitetes objektbasiertes Verhaltensmuster, in dem Kommandos als Objekte gekapselt werden und daher Anfragen an unbekannte Anwendungsobjekte gerichtet werden können [12]. Generell besteht das *Command-Pattern* aus mehreren Komponenten, die lose miteinander Gekoppelt sind:

- Der `Invoker` dient zur Verwaltung und Ausführung der Kommandos. Diese Klasse enthält eine Liste von Kommandos und kann diese entweder sofort ausführen, oder für einen späteren Zeitpunkt abspeichern. Die Liste enthält Objekte der abstrakten Klasse `Command`.
- Die `Command`-Klasse definiert die Schnittstelle für jedes Kommando. Sie enthält eine Referenz auf den `Receiver` und definiert die zu überschreibende Methode zum Ausführen des Kommandos.
- Für jedes gewünschte Kommando wird eine neue Klasse erstellt, die von der Schnittstelle `Command` ableitet und die Methode zum Ausführen implementiert. Diese ruft typischerweise eine Aktion des `Receiver` auf.
- Der `Reveiver` definiert die eigentlichen Aktionen, die durch die Kommandos ausgelöst werden sollen. Diese werden von den implementierten Kommandos aufgerufen.
- Der `Client` erstellt nun die eigentlichen Kommandos und gibt ihnen den gewünschten *Receiver* mit. Dadurch kennen sich `Command` und `Receiver` nicht und sind in hohem Grad austauschbar.

Im konkreten Fall dieses Frameworks werden die bisweilen nur als Zeichenkette definierten Kommandos (`GET IMAGE`, `PAUSE` etc.) nun als Objekte definiert, die sich alle nach der abstrakten Klasse `Command` richten. Zusätzlich gibt es die Klasse `Invoker` zur Verwaltung der Kommandos, die gleichzeitig zur Lösung des Threading-Problems des Frameworks dient, wie in Abschnitt 5.2.5 beschrieben wird. Die Klasse `GameEngineController` übernimmt im Framework die Doppelfunktion für `Client` und `Receiver`. Sie erstellt je nach erhaltenen Daten die benötigten Kommandos und übergibt diese dem `Invoker`, der sie dann ausführen kann. Abbildung 5.3 zeigt das UML[2]-Diagramm des im Framework eingebauten *Command-Patterns*, durch dessen Implementierung sich eine hohe Austauschbarkeit und Erweiterbarkeit des Programms ergibt. Außerdem ist durch die Verwendung dieses Pat-

[2] Unified Modeling Language

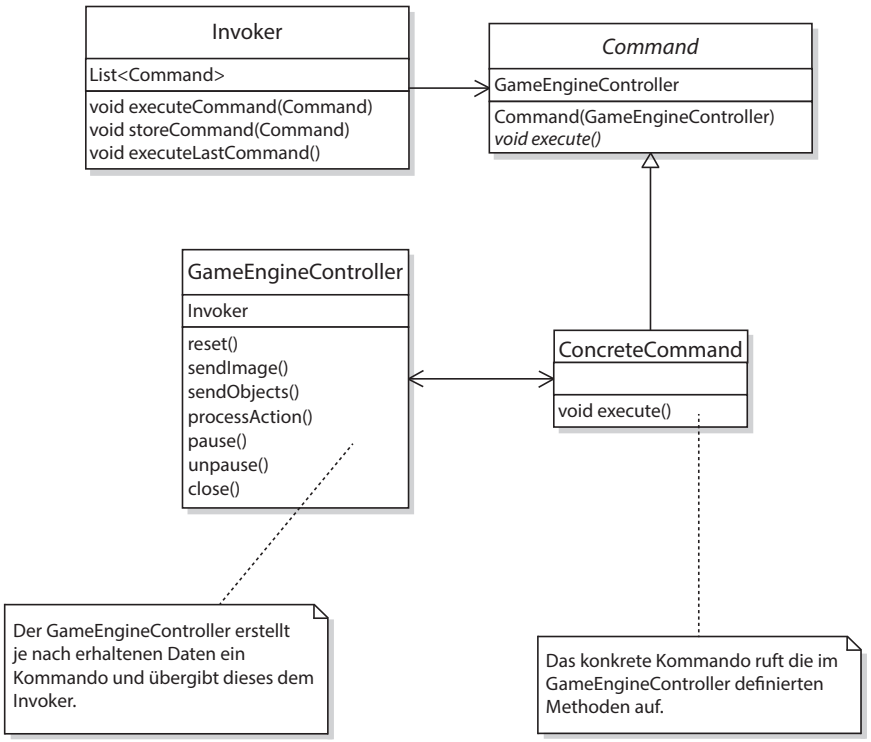


Abbildung 5.3: UML-Diagramm des *Command-Patterns*, wie es im Framework eingebaut ist. Das Erstellen der Kommandos übernimmt in der finalen Implementierung die hier nicht eingezeichnete *CommandFactory*.

terns gewährleistet, dass mehrere aufeinanderfolgende Anfragen gereiht und hintereinander abgearbeitet werden können.

Factory-Pattern

Um die Implementierung des Kommandosystems noch weiter zu vereinfachen, wurde ein weiteres Pattern verwendet. Mit dem *Factory-Pattern* wird die konkrete Erstellung von Objekten gekapselt und in eine eigene Klasse ausgelagert. Im Framework werden daher die Kommandos nicht mehr vom `GameEngineController`, sondern in einer zentralen `Factory`-Klasse erstellt. Dadurch wird das Programm noch übersichtlicher und besser erweiterbar. Diese `CommandFactory` besteht nur aus einer statischen Methode mit zwei Übergabeparametern. Der erste ist das über den Socket erhaltene Kommando als Zeichenkette. Je nach Interpretation dieses Parameters wird ein entsprechendes Kommando erzeugt und zurückgegeben. Der zweite Übergabeparameter ist eine Referenz auf den Controller selbst, der den erstellten Kommandos mitgegeben wird.

Durch die Verwendung dieses Patterns wird der Aufruf in der Klasse `GameEngineController` um ein neues Kommando zu erstellen nachdem Daten vom Server empfangen wurden auf ein Minimum reduziert:

```
byte[] receivedData = socketServer.readData(32);
string commandString = Encoding.ASCII.GetString(receivedData);
Command command = CommandFactory.CreateCommand(commandString, this);
invoker.executeCommand(command);
```

Dieser Codeabschnitt ist die finale Implementierung im Framework zum Empfangen und Interpretieren von Daten. Diese wird in einer Endlosschleife immer wieder aufgerufen, um auf sämtliche vom VES kommenden Kommandos reagieren zu können.

5.2.5 Threading

Eines der Hauptprobleme bei der Implementierung der Library war der richtige Umgang mit Threads. Nachdem der Socket beim Warten auf eingehende Daten das Programm solange anhält, bis Daten empfangen werden, ist es wichtig, dass dies in einem eigenen Thread passiert, damit die Game Engine parallel dazu weiterlaufen kann. Dies wird durch eine asynchrone Abfrage auf eingehende Clientverbindungen erreicht, durch die ein Nebenthread eröffnet wird. Die Endlosschleife zum Empfangen und Interpretieren von Daten über diese Verbindung befindet sich daher ebenfalls im Nebenthread und es findet keine Blockierung der Applikation statt. Die Aufrufe der abstrakten Callback-Funktionen, die direkt nach der Interpretation der empfangenen Daten erfolgen, sollten jedoch vom Hauptthread aus erfolgen, da sonst der Benutzer bei dessen Implementierungen eventuell Probleme mit Cross-Thread-Aufrufen bekommen könnte (die in Kapitel 6 beschriebene Implementierung mit *Unity3D* hat beispielsweise solche Probleme ausgelöst, da die Aufrufe der Unity-API nur im Hauptthread erfolgen dürfen). Um dieses Ziel zu erreichen, wurde mit der in Abschnitt 5.2.4 beschriebenen Klasse `Invoker` gearbeitet, die es ermöglicht, Kommandos zunächst nur abzuspeichern und zu einem späteren Zeitpunkt beziehungsweise von einem anderen Thread heraus auszuführen. Das im Nebenthread empfangene Kommando wird also dem `Invoker` nur zur Speicherung übergeben. Der Aufruf zum Ausführen erfolgt dann aus dem Hauptthread heraus. Abbildung 5.4 illustriert diesen Vorgang.

Mit der in diesem Abschnitt beschriebenen Architektur und den verwendeten Implementierungen und Design-Patterns hat der Benutzer die Möglichkeit durch die Verwendung des Frameworks eine virtuelle Umgebung sehr einfach in einen Datenlieferanten, mit der Möglichkeit zur Kommunikation und Steuerung von außen, umzuwandeln und muss selbst nur die Erweiterung der Game Engine vornehmen und dort die Game Engine-spezifischen Implementierungen in die bereitgestellten Callback-Funktionen einbauen.

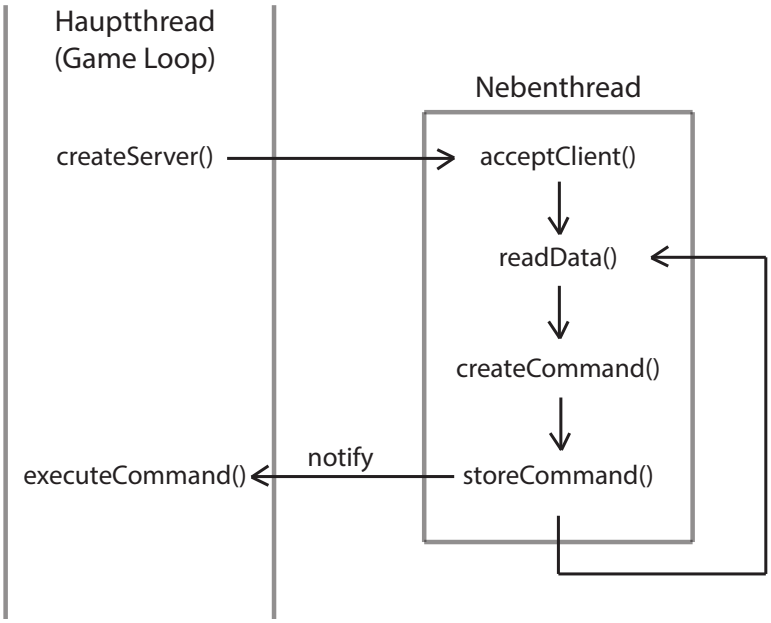


Abbildung 5.4: Verlauf und Kommunikation zwischen dem Hauptthread, in dem die Funktionalitäten der Game Engine laufen und sämtliche Kommandos ausgeführt werden sollen, und dem Nebenthread, der sich um den Datenaustausch und die Socketverbindung kümmert.

5.3 Die Library – Vision Engine

In diesem Abschnitt wird nun der Vision Engine-Teil der Library (Abbildung 5.5) beschrieben. Abermals muss die Library zur Verwendung in die Erweiterung der Engine eingebunden werden. Die Implementierung von VECA für die Vision Engine ist im Gegensatz zu jener für die Game Engine um einiges weniger Umfangreich, da nicht so viele Steuerungsmechanismen und keine Cross-Thread Aufrufe notwendig sind. Um aber auch hier die Austauschbarkeit zu gewährleisten, wurde ebenfalls mit den Konzepten der Callback-Funktionen und der Kommando-Interpretation gearbeitet, wenn auch in stark vereinfachter Form. Die Erstellung einer Vision Engine muss vom Benutzer aber nur dann vorgenommen werden, wenn dieser das Framework als virtuelle Testumgebung verwenden möchte. Für die reine Generierung von Daten reicht das Aufsetzen der Game Engine. Im Zuge einer virtuellen Testumgebung werden dann nämlich die Bilder aus der Game Engine direkt an die Vision Engine weitergeleitet, damit diese darauf reagieren kann.

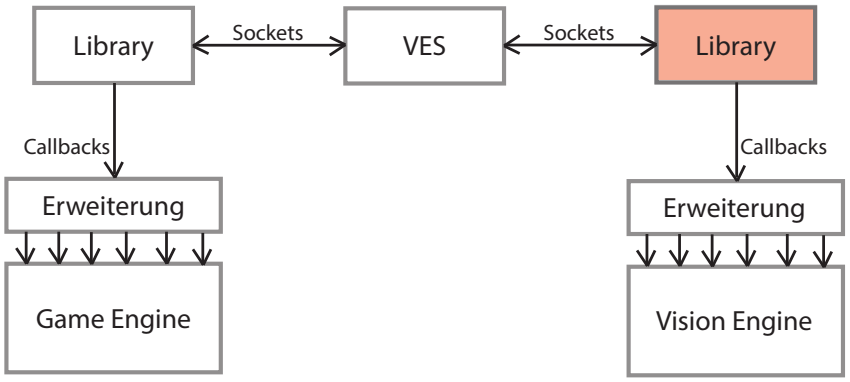


Abbildung 5.5: Die Library muss in den vom Benutzer erstellten Erweiterungscode der Vision Engine eingebunden werden.

5.3.1 Klassenstruktur

Für die Verwendung des Frameworks in der Vision Engine wird diesmal eine andere Hauptklasse zur Verfügung gestellt. Der `VisionEngineController` kümmert sich um die Netzwerkkommunikation und den allgemeinen Ablauf. Für den Umgang mit Sockets wird abermals die in Abschnitt 5.2.1 beschriebene Klasse `ServerSocket` verwendet, mit deren Hilfe auf eingehende Verbindungen gehorcht wird – natürlich auf einen anderen Port als der `GameEngineController` – und Daten empfangen werden können. Abermals wird auf die eingehenden Daten mit Hilfe von Kommandos reagiert, allerdings ohne die umfangreiche Implementierung eines Kommandosystems, da die Vision Engine nur auf drei verschiedene Dinge reagieren muss:

- Das Kommando `CLOSE` verlangt das Schließen des Sockets und das Beenden des Programms.
- Beim Kommando `PING SERVER` soll wieder nur die Socket-Verbindung überprüfen. In diesem Fall wird dasselbe Kommando wieder über den Socket zurückgesendet.
- Alle anderen eingehenden Daten werden versucht als Bilddaten zu interpretieren.

Sollten Bilddaten angekommen sein wird folgende abstrakte Callback-Methode aufgerufen:

```
protected abstract byte[] proceedImage(byte[] image);
```

Diese Methode muss vom Benutzer in der Erweiterung implementiert werden, wenn er von der Klasse `VisionEngineController` ableitet und hat als Übergabeparameter die Bilddaten und als Rückgabewert eine Aktion. Hier soll der eigentliche Computer Vision Algorithmus implementiert sein, der

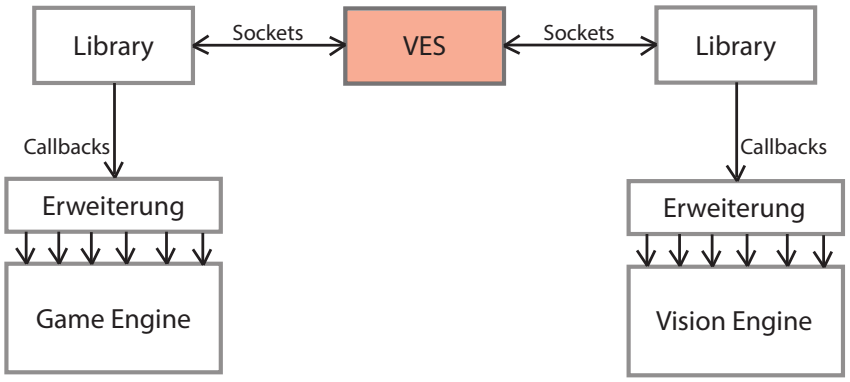


Abbildung 5.6: Die Schnittstelle kommuniziert über Sockets mit der Library.

dann auf den Bilddaten angewendet werden soll. Anhand des Ergebnisses soll dann eine Aktion definiert und zurückgegeben werden. Diese soll beschreiben, wie sich die Game Engine verhalten soll und wird dann vom Framework an die Game Engine weitergeleitet. Wie diese Aktion aussieht ist dem Benutzer überlassen. Die Interpretierung der Aktion erfolgt auf Game Engine-Seite dann in einer Callback-Funktion der Klasse `GameEngineController`.

5.4 Der Virtual Environment Supervisor

Die dritte Komponente im Framework ist die Schnittstelle, die die beiden Engines miteinander verbindet (Abbildung 5.6). Diese ist im Gegensatz zu den anderen Komponenten keine Library, sondern ein eigenständiges Programm, das vom Benutzer nur ausgeführt werden muss und sich automatisch zu den in den Engines eingebundenen Libraries verbindet. Sie stellt eine graphische Benutzeroberfläche zur Verfügung, mit der man die Verbindungen und den Datenaustausch steuern kann. Ziel war es hier, die Handhabung des Programms möglichst einfach und mit wenigen Formularkomponenten zu realisieren, und dem Benutzer dennoch eine möglichst umfangreiche Kontrolle über das Geschehen innerhalb des Frameworks zu geben.

5.4.1 Klassenstruktur

Der interne Aufbau des Programms ist relativ einfach gehalten und kommt mit wenigen Klassen aus. Neben der weiter unten beschriebenen Hauptklasse `Supervisor` wurden zusätzlich noch folgende Hilfsklassen implementiert:

UserInterface

Die Klasse `UserInterface` erstellt und verwaltet die graphische Benutzeroberfläche. Sie stellt unter anderem Methoden zum Anzeigen von Bildern und Ausgeben von Text zur Verfügung. Außerdem werden hier Benutzereingaben – hauptsächlich über Buttons – abgefragt und an den `Supervisor` weitergeleitet, wo sie dann interpretiert werden. Aussehen und Funktionsweise der graphischen Benutzeroberfläche werden in Abschnitt 5.4.2 beschrieben.

SocketClient

Ähnlich der in Abschnitt 5.2.1 beschriebenen Klasse `SocketServer` gibt es auch hier eine Klasse, die sich um die Netzwerkverbindungen und Datentransfers kümmert. Die Klasse `SocketClient` kann sich über einen spezifizierten Port zu einem Server verbinden und anschließend über diese Verbindung Daten senden und empfangen. Innerhalb des Frameworks verbinde sich zwei Objekte der Klasse `SocketClient` sowohl zur Library in der Game Engine als auch zu jener in der Vision Engine.

IDataSaver

Wenn die Game Engine zur Datengenerierung verwendet wird, dient diese Hilfsklasse zum Abspeichern der von der Engine erhaltenen Daten. Um sich hier die Möglichkeit offen zu halten, die Art der Speicherung möglichst dynamisch ändern zu können wurde diese als Interface implementiert:

```
1 interface IDataSaver
2 {
3   void saveImage(byte[] imgData, string id);
4   void saveData(byte[] data, string id);
5 }
```

Im Moment wird eine Implementierung verwendet, die die Daten auf die Festplatte schreibt. Dazu implementiert die Klasse `FileDataSaver` das genannte Interface und überschreibt die gegebenen Methoden, in denen sowohl die Bilddaten als auch die Metadaten in zwei verschiedene Unterordner des Projektordners gespeichert werden. Die Methoden werden zu bestimmten Zeitpunkten vom `ClientFormController` aufgerufen. Der Parameter `id` dient zur Zuordnung zwischen Bild und Daten und bezeichnet gleichzeitig den Dateinamen.

Settings

Zusätzlich gibt es noch die statische Klasse `Settings`, die lediglich gewisse Konstanten für die Applikation zur Verfügung stellt. So werden hier beispielsweise die Texte für die Tooltips der Buttons oder für die Warnmeldungen definiert.

Supervisor

Der `Supervisor` ist die Hauptklasse, die die oben genannten Klassen beinhaltet, um die Applikation zu verwalten und zu steuern. Sie beinhaltet zwei Objekte der Klasse `SocketClient`, um Verbindungen zur Game Engine und zur Vision Engine herstellen zu können. Außerdem sind hier verschiedene Methoden implementiert, die zur unterschiedlichen Kommunikation mit den beiden Engines dienen und je nach Interaktion des Benutzers mit dem Interface aufgerufen werden. Folgende Methoden sind hierfür in der Klasse `Supervisor` enthalten:

- `connect()`
  Diese Methode versucht sich zu den beiden Engines zu verbinden, falls noch keine Verbindungen vorhanden sind, und gibt bei Fehschlag eine entsprechende Meldung in der Konsole aus.
- `checkConnections()`
  Hier wird zunächst überprüft, ob Verbindungen vorhanden sind. Falls dies zutrifft wird über diese Verbindungen ein einfaches `PING SERVER`-Kommando geschickt, das von den Engines wieder zurückgesendet werden soll. Der Fortschritt der Überprüfung sowie eventuelle Fehlschläge werden in der Konsole ausgegeben.
- `receiveAndDisplayImage()`
  Sendet einen Request für das aktuelle Render-Bild an die Game Engine, interpretiert anschließend die erhaltene Antwort als Bild und versucht, dieses im Vorschaufenster anzuzeigen.
- `receiveAndDisplayObjects()`
  Sendet einen Request für die aktuellen Metadaten der in der Szene enthaltenen Objekte an die Game Engine und gibt anschließend die erhaltene Antwort in der Konsole aus.
- `receiveAndSaveImage(string id)`
  `receiveAndSaveObjects(string id)`
  Diese Methoden senden jeweils ein Kommando an die Game Engine, um das aktuelle Bild und die aktuellen Metadaten zu erhalten. Diese werden dann mit Hilfe des `FileDataSaver` gespeichert. Als `id` wird ein Zeitstempel in Millisekunden-Genauigkeit generiert.
- `recieveAndSendImage()`
  Abermals wird ein Request für das aktuelle Bild an die Game Engine gesendet. Die erhaltenen Daten werden dann im Vorschaufenster angezeigt und an die Vision Engine weitergesendet.
- `recieveAndSendAction()`
  Hier wird zunächst auf eingehende Daten aus der Vision Engine gewartet, die die auszuführende Aktion für die Game Engine beinhalten sollen, die dann an diese weitergeleitet werden.
- `resetGameEngine()`

`pauseGameEngine()`
`unpauseGameEngine()`
Diese Methoden schicken jeweils ein Kommando zum Zurücksetzen, Pausieren oder Fortsetzen an die Game Engine.

- `closeConnections()`
  Sendet jeweils ein Kommando zum Schließen an die Game Engine und die Vision Engine und schließt anschließend beide Sockets.
- `startStop()`
  Diese Methode dient zum Starten beziehungsweise Stoppen des automatischen Durchlaufs. Beim Starten wird in einem eigenen Thread eine Endlosschleife gestartet, die je nach eingegebenen Einstellungen vom Benutzer in gewissen Zeitabständen Daten von der Game Engine holt und diese entweder abspeichert oder an die Vision Engine weiterschickt und gegebenenfalls die Antwort der Vision Engine auch wieder an die Game Engine weiterleitet.
- `append(string output)`
  Über diese Methode können Ausgaben auf der Konsole gemacht werden.

Damit diese Methoden in der gesamten Applikation verfügbar sind, wurde der `Supervisor` mit dem *Singleton-Pattern* implementiert. Dabei wird sichergestellt, dass nur eine Instanz des `Supervisor` existiert. Über eine statische Methode kann auf diese Instanz von überall in der Applikation aus zugegriffen werden:

```
private static Supervisor instance;
public static Supervisor GetInstance()
{
  if (instance == null)
  {
    instance = new Supervisor();
  }
  return instance;
}
```

Dadurch können z. B. vom `UserInterface` aus direkt die gewünschten Methoden aufgerufen werden, nachdem ein Benutzerinput erkannt wurde. Außerdem können die in der Klasse `SocketClient` eventuell auftretenden Verbindungsfehler direkt eine Ausgabe in der Konsole erzeugen, indem auf die Instanz des `Supervisor` zugegriffen wird, ohne dass eine Objektreferenz vorhanden sein muss.

5.4.2 Graphische Benutzeroberfläche

Das Kernstück der Applikation ist die in Abbildung 5.7 gezeigte graphische Benutzeroberfläche, die es dem Benutzer ermöglicht, VECA zu steuern und zu konfigurieren. Sie interagiert sehr stark mit der Klasse `Supervisor` und

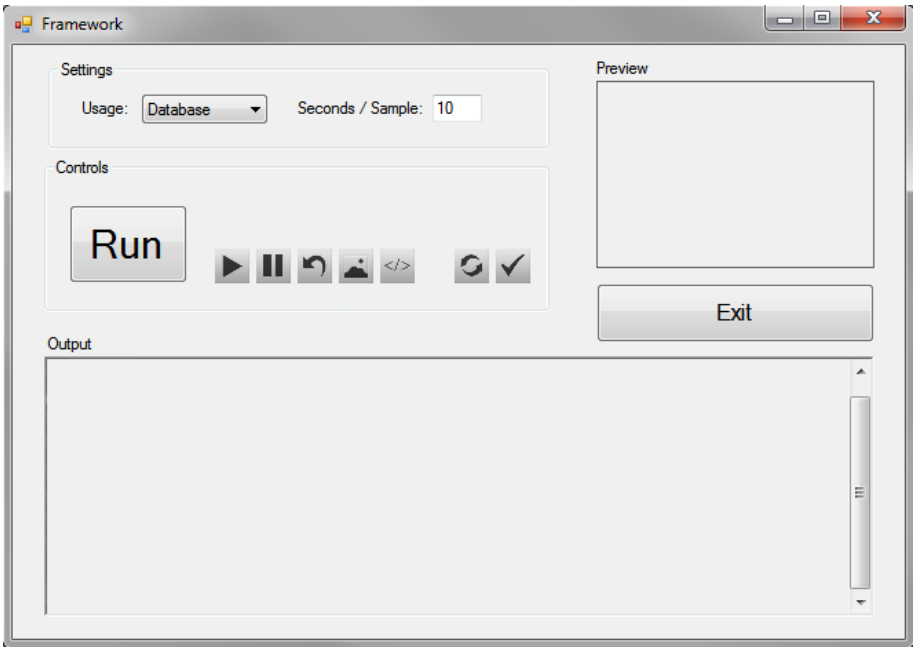


Abbildung 5.7: Graphische Benutzeroberfläche der Schnittstelle. Sie ist in die Bereiche *Settings*, *Controls*, *Output* und *Preview* eingeteilt.

ruft deren Methoden auf. Die Benutzeroberfläche ist in vier gekennzeichnete Bereiche eingeteilt:

Settings: In diesem Bereich am oberen Rand können vom Benutzer die wichtigsten Einstellungen gemacht werden:

- *Usage:* Über diese Auswahlbox kann festgelegt werden, wie das Framework zu verwenden ist:
  - *Database:* Man verwendet das Framework zur reinen Datengenerierung von Bildern und zugehörigen Ground Truth-Daten und möchte diese einfach nur in einer Datenbank abspeichern.
  - *Vision Engine:* Man verwendet das Framework als virtuelle Testumgebung unter Verwendung einer Vision Engine. In diesem Fall werden die Bilder der Game Engine an die Vision Engine weitergeleitet und dessen definierte Aktion wieder zurück an die Game Engine gesendet.
- *Seconds/Sample:* Hier kann das Intervall in Sekunden angegeben werden, in welchen Abständen neue Daten generiert werden sollen.

Controls: Direkt unter den *Settings* befindet sich der Bereich *Controls*. Hier werden folgende Buttons zur Steuerung des Frameworks zur Verfügung

gestellt:

- *Run* startet den Durchlauf des Frameworks. Je nach angegebenen Einstellungen werden in regelmäßigen Zeitabständen Daten aus der Game Engine geholt und abgespeichert, oder Daten von der Game Engine an die Vision Engine weitergeleitet, sowie dessen Antwort wieder zurückgesendet.
- Die nächsten drei Buttons dienen zur direkten Steuerung, indem sie der Game Engine jeweils ein Kommando zum Pausieren, Fortsetzen oder Zurücksetzen senden.
- Zwei weitere Button dienen zum Testen der `GET IMAGE`- und `GET OBJECTS`-Funktionen. Bei einem Klick auf einen dieser Buttons wird das entsprechende Kommando an die Game Engine gesendet. Das erhaltene Bild wird anschließend im Vorschaufenster angezeigt, die erhaltenen Daten in der Konsole ausgegeben.
- Die letzten zwei Buttons dienen zu Kontrolle der Verbindungen zu den Engines. Der eine dient zum Neuaufbau, der andere zum Überprüfen der Verbindungen.

Preview: Der Bereich rechts oben dient als Platzhalter für den Fall, dass von der Game Engine ein Bild geholt wurde, um eine Vorschau anzeigen zu können. Gleich darunter befindet sich der Button zum Schließen der Applikation, bei dem die Verbindungen geschlossen und anschließend das Programm beendet wird.

Output: In der Konsole am unteren Rand der Oberfläche werden nützliche Informationen zu den Verbindungen und zum Status der Applikationen ausgegeben.

Kapitel 6

Anwendungsbeispiel

Das letzte Kapitel dieser Arbeit beschäftigt sich damit, wie man VECA verwendet. Es wird anhand einer Beispielimplementierung gezeigt, wie man die Library in ein eigenes Projekt einbindet, die benötigten Callback-Funktionen erstellt und anschließend den *Virtual Environment Supervisor* verwendet, um seine Applikation zu steuern. Zunächst werden das allgemeine Setup und das angewendete Szenario beschrieben. Danach wird gezeigt, wie man VECA jeweils in die Game Engine und in die Vision Engine einbindet und die für das Szenario notwendigen Implementierungen vollzieht. Anschließend wird erläutert, wie man nun mit Hilfe der graphischen Benutzeroberfläche des VES das Szenario steuern und das Framework zur Datengenerierung bzw. als virtuelle Testumgebung verwenden kann.

6.1 Setup

Um die Verwendung von VECA zeigen zu können wurde ein ganz einfaches Szenario ausgearbeitet, das innerhalb des Frameworks laufen soll. Als Game Engine für dieses Beispiel wurde *Unity3D* ausgewählt. In dieser wird eine einfache Szene dargestellt in der sich zwei Autos auf einer Straße hintereinander bewegen. Als Basis für diese Szene wurde ein Online verfügbares Unity3D-Tutorial verwendet [28]. Die Kamera der Szene befindet sich im hinteren Auto. Abbildung 6.1 zeigt die Szene aus Sicht des Unity3D-Editors. Über Computer Vision soll nun automatisch erkannt werden, ob dieses Fahrzeug noch beschleunigen kann, oder ob es schon zu nah ans Vorderauto herangerückt ist und lieber abbremsen sollte. Der Algorithmus zum Erkennen der Entfernung der Fahrzeuge befindet sich in einem eigenen Programm, welches den Part der Vision Engine erfüllt. Das Beispiel selbst dient als Demonstration für die richtige Verwendung des Frameworks. Die in der Game Engine abgebildete Szene ist vermutlich noch nicht realitätsnah genug, um eine wirkliche Anwendung zu finden, reicht aber durchaus für eine Beispielimplementierung aus. Ebenso ist der implementierte Computer Vision Algorithmus

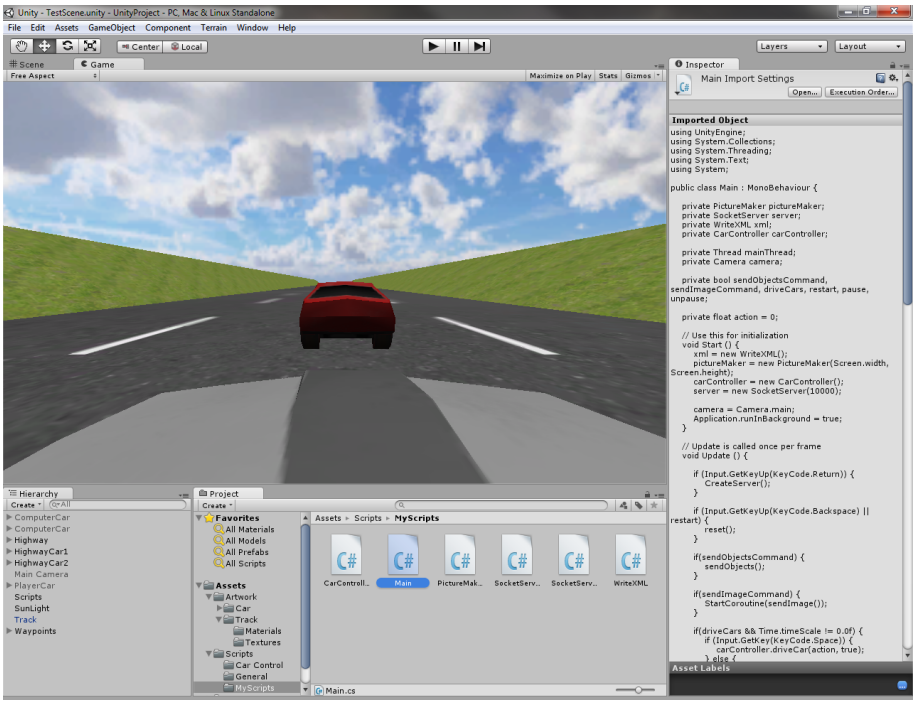


Abbildung 6.1: Die für die Beispielverwendung implementierte Szene innerhalb des Unity3D-Editors.

wohl nicht allgemein genug, um weitgehend eingesetzt werden zu können, ist aber darauf ausgelegt, die in der Game Engine abgebildeten Autos zu erkennen. Somit kann das Zusammenspiel zwischen den einzelnen Komponenten sowie eine mögliche Implementierung der Callback-Funktionen und die Verwendung der graphischen Benutzeroberfläche gezeigt werden.

6.2 Implementierung

6.2.1 Die Game Engine

Um VECA in der Game Engine verwenden zu können, muss die Engine zunächst um einen Code erweitert werden, der die kompilierte Datei des Frameworks – `veca.dll` – einbindet (Abbildung 6.2). In dieser Erweiterung wird anschließend über ein `using`-Statement die Library verwendet und es muss eine Klasse – hier `MyController` genannt – erstellt werden, die von der in VECA enthaltenen Klasse `GameEngineController` ableitet. Hier können nun die von der Klasse `GameEngineController` vorgegebenen abstrakten Callback-Methoden in der eigenen Klasse implementiert werden. Diese sollen die gewünschten Daten liefern bzw. die gewünschte Aktion ausführen und werden automatisch aufgerufen. Programm 6.1 zeigt den Quellcode für

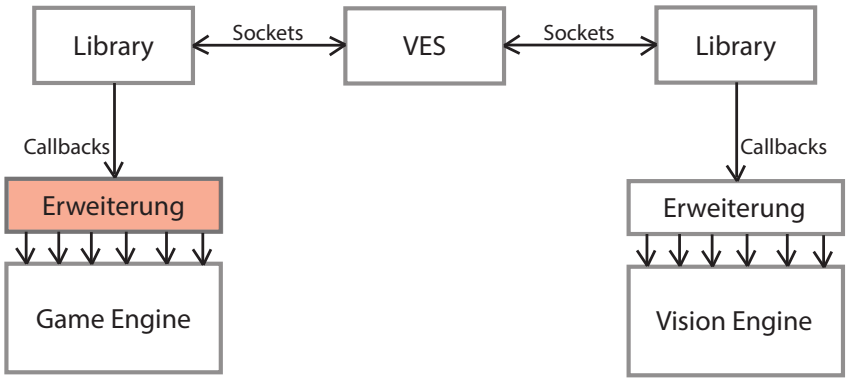


Abbildung 6.2: Die Erweiterung der Game Engine bindet die Library ein und implementiert die zur Verfügung gestellten Callback-Funktionen.

die Implementierung der Funktionen unter Verwendung der Unity-API und zwei selbst geschriebenen Hilfsklassen. Diese Hilfsklassen `PictureMaker` und `WriteXML` in Zeile 2 und 3 kapseln die Funktionen zum erzeugen der Bild- bzw. Objektdaten. Wie man diese Daten beispielsweise aufbereiten kann, wird weiter unten beschrieben. Um die Engine zurückzusetzen, wird der Unity3D-spezifische Aufruf `Application.LoadLevel(0)` verwendet. Zum Anhalten und Fortsetzen der Simulation wird die Variable `Time.timeScale` entsprechend gesetzt. Die Methode `performAction(string action)` wird aufgerufen, wenn von der Vision Engine eine Aktion erhalten wurde. In diesem Beispiel wurde spezifiziert, dass die Aktionsvariable entweder 0 oder 1 beinhaltet und angibt, ob das Auto beschleunigen darf oder nicht. Dieser Wert wird für die spätere Verwendung erstmal abgespeichert. Des Weiteren ist zu beachten, dass die Klasse `GameEngineController` eine Methode `update()` beinhaltet, die innerhalb der Game-Loop aufgerufen werden sollte. Dazu wurde in der eigenen Klasse `MyController` eine eigene `update()`-Methode eingebaut (Zeile 7), die einerseits die verlangte Methode der Basisklasse aufruft und sich andererseits um die Fortbewegung der Autos kümmert. Hier wird die zuvor gespeicherte Variable `speed` mitgegeben, um zu spezifizieren, ob das hintere Auto beschleunigen (Wert: 1) darf oder abbremsen soll (Wert: 0).

Wenn nun von der Library ein entsprechendes Kommando empfangen wird, wird automatisch die zugehörige Callback-Funktion aufgerufen. Somit sind alle Voraussetzungen erfüllt, dass der Controller innerhalb des Frameworks funktioniert.

Programm 6.1: Von der Klasse *GameEngineController* abgeleitete Implementierung mit sämtlichen ausgefüllten Callback-Funktionen.

```
1 public class MyController : GameEngineController {
2   private PictureMaker pictureMaker = new PictureMaker(Screen.width,
      Screen.height);
3   private WriteXML writeXml;
4   private CarController carController = new CarController();
5   private float speed = 0;
6
7   public void update() {
8     base.update();
9     carController.driveCar(speed, false);
10  }
11
12  protected override void resetEngine() {
13    Application.LoadLevel(0);
14  }
15
16  protected override byte[] getImageData() {
17    return pictureMaker.takePicture();
18  }
19
20  protected override byte[] getObjectData() {
21    return writeXml.WriteGameObjectsIntoXml();
22  }
23
24  protected override void performAction(string action)  {
25    try {
26      speed = float.Parse(action);
27    }catch(System.IO.IOException e){}
28  }
29
30  protected override void pauseGameEngine() {
31    Time.timeScale = 0.0f;
32  }
33
34  protected override void unpauseGameEngine() {
35    Time.timeScale = 1.0f;
36  }
37 }
```

Introspektion

Wie bereits angesprochen wird in diesem Abschnitt noch kurz erläutert, wie man beispielsweise auf die Daten innerhalb der Game Engine zugreifen und diese aufbereiten kann. Je nachdem wie die Weiterverarbeitung erfolgt, ist es dem Benutzer selbst überlassen, wie die Datenaufbereitung aussieht. In dieser Beispielimplementierung wurde das einfache Konzept von

XML[1] verwendet. In der selbst erstellten Klasse `WriteXML` wird in der Methode `WriteGameObjectsIntoXml` zunächst über folgenden Unity3D-Befehl auf sämtliche Objekte in der Szene zugegriffen:

```
GameObject[] objects = (GameObject[]) Object.FindObjectsOfType(typeof(
    GameObject));
```

Die Instanzvariablen der im Array enthaltenen Objekte mit dem Typ `GameObject` enthalten nun schon einige nützliche Informationen wie Name oder Position des Objekts in der Szene. Außerdem werden über Raytracing und Geometriefunktionen noch einige weitere nützliche Informationen berechnet. Diese beinhalten die Screen-Position des Objekts, Koordinaten des umschließenden Rechtecks sowie Angaben, ob sich besagtes Objekt im Sichtfeld der Kamera befindet und ob es von einem anderen Objekt verdeckt wird. Ein fertiger XML-Eintrag, der sämtliche für das Objekt relevanten Informationen beinhaltet, sieht so aus:

```
<gameObject>
  <name>RedCar</name>
  <worldPosition>
    <worldX>0</worldX>
    <worldY>0</worldY>
    <worldZ>-50</worldZ>
  </worldPosition>
  <screenPosition>
    <screenX>581</screenX>
    <screenY>306</screenY>
    <screenZ>12</screenZ>
  </screenPosition>
  <boundingRectangle>
    <center>
      <x>581</x>
      <y>315</y>
    </center>
    <width>135</width>
    <height>87</height>
  </boundingRectangle>
  <isInCameraView>True</isInCameraView>
  <isSeenByCamera>True</isSeenByCamera>
</gameObject>
```

Wenn man das Framework zur reinen Datengenerierung verwendet, werden diese Daten gemeinsam mit dem entsprechenden Bild an den VES gesendet, der diese dann abspeichert.

6.2.2 Die Vision Engine

Um VECA als virtuelle Testumgebung zu verwenden, muss auch eine Vision Engine erstellt werden. Dazu wurde in diesem Szenario eine einfache Konsolenapplikation erstellt, die einen simplen Computer Vision Algo-

[1] Extensible Markup Language

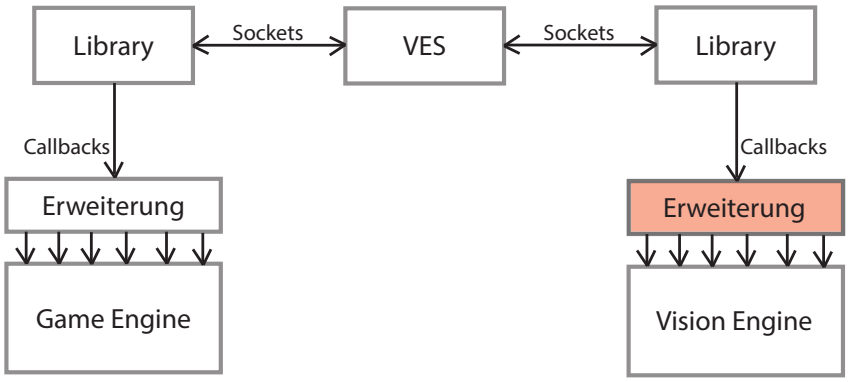


Abbildung 6.3: Die Erweiterung der Vision Engine bindet die Library ein und implementiert die zur Verfügung gestellte Callback-Funktion.

rithmus enthält. Über einen implementierten Erweiterungscode wird abermals die Library eingebunden (Abbildung 6.3). Erneut wird in diesem Code eine eigene Klasse implementiert, als Basisklasse wird jetzt aber der `VisionEngineController` verwendet. Dieses Mal muss nur eine einzige Callback-Funktion implementiert werden, die auch wieder automatisch vom Framework aufgerufen wird. Diese Methode `proceedImage(byte[] image)` erhält als Übergabeparameter die Bilddaten aus der Game Engine. Ausgefüllt werden soll die Methode mit dem gewünschten Computer Vision Algorithmus und als Rückgabewert soll eine Aktion für die Game Engine spezifiziert werden. In diesem Beispiel ist das ein einfacher selbst implementierter Algorithmus, der den Abstand von einem im Bild vorhandenen Auto zur Kamera abschätzt. Wenn das Auto noch weit genug entfernt ist, wird als Aktion die Zahl 1 definiert, ansonsten wird die Zahl 0 zurückgegeben. Eine vereinfachte Form der Implementierung der Klasse sieht so aus:

```
1 class MyController : VisionEngineController
2 {
3     private static float MIN_DISTANCE = 50.0f;
4
5     protected override byte[] proceedImage(byte[] image)
6     {
7       float distance = this.compute(image);
8
9       string action = (distance > MIN_DISTANCE) ? "1" : "0";
10
11      return Encoding.ASCII.GetBytes(action);
12    }
13 }
```

Die in Zeile 7 aufgerufene Methode `compute(byte[] image)` enthält einen Algorithmus, der das Auto auf dem Bild erkennt (Abbildung 6.4) und die Di-

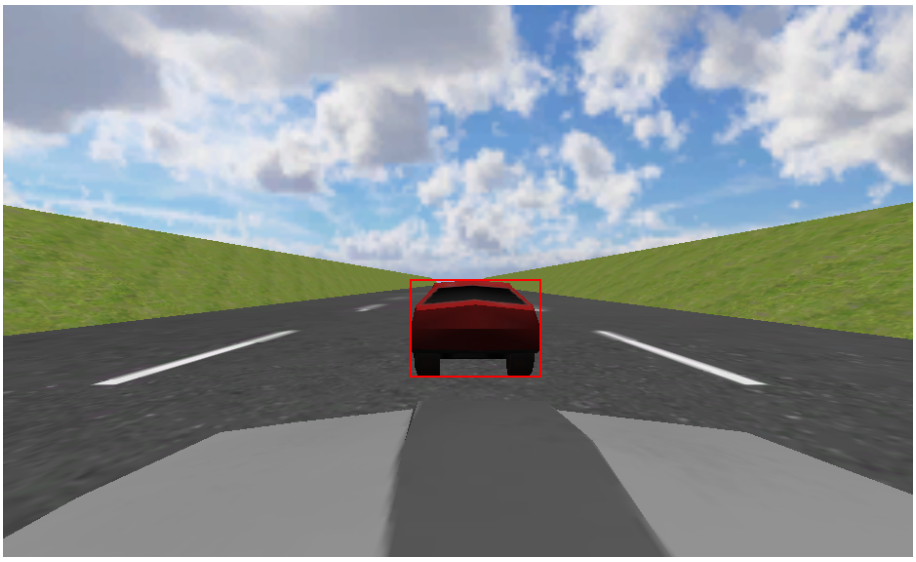


Abbildung 6.4: Der implementierte Algorithmus erkennt das Auto im Bild und schätzt dessen Entfernung.

stanz zur Kamera berechnet. Je nach Ergebnis wird dann die entsprechende Aktion zurückgegeben.

6.3 Ausführung

Nachdem nun die notwendigen Implementierungen vorgenommen wurden, kann das Framework verwendet werden. Zunächst müssen die beiden Engines gestartet und irgendwo im jeweiligen Programm die implementierten Klassen `GameEngineController` und `VisionEngineController` erstellt werden, damit diese auf Verbindungen horchen. Dann kann der *Virtual Environment Supervisor* über die Datei `ves.exe` gestartet werden. Nun erscheint die graphische Benutzeroberfläche und es wird versucht eine Verbindung zu den beiden Engines herzustellen. Im Output-Fenster werden entsprechende Informationen ausgegeben, ob der Verbindungsaufbau erfolgreich war. Falls dies nicht so ist, ist eventuell ein Neustart der Engines erforderlich, oder die Controller sind noch nicht initialisiert worden. Abbildung 6.5 zeigt die Benutzeroberfläche mit erfolgreich aufgebauten Verbindungen.

Nun können die Funktionalitäten über die graphische Benutzeroberfläche gesteuert werden. Die kleinen Buttons dienen zur allgemeinen Kontrolle und zum Überprüfen der implementierten Callback-Funktionen. Mit einem Klick auf den Button *Reset Game Engine* wird beispielsweise die Callback-Funktion `resetEngine()` in der Game Engine aufgerufen. Bei Betätigung der Buttons *Get Image* oder *Get Objects* werd jeweils die Callback-

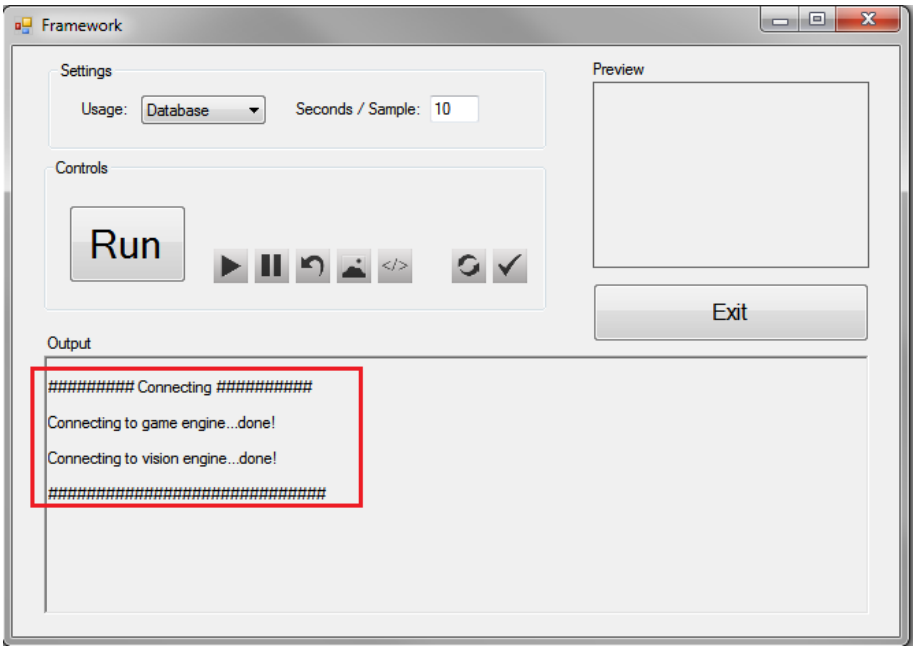


Abbildung 6.5: Output der Benutzeroberfläche bei erfolgreichem Verbindungsaufbau.

Funktionen `getImageData()` bzw. `getObjectData()` aufgerufen und es wird versucht, den Rückgabewert entweder in der Konsole auszugeben oder als Vorschaubild anzuzeigen. Wenn man nun die eigentliche Datengenerierung starten will, muss zunächst ausgewählt werden, in welcher Form das Framework verwendet werden soll. Im Bereich *Settings* kann eingestellt werden, ob die generierten Daten in einer Datenbank abgespeichert werden sollen – zu diesem Zweck ist keine Verbindung zu einer Vision Engine notwendig – oder ob das Framework als virtuelle Testumgebung verwendet werden soll. Des Weiteren kann man das Intervall einstellen, in welchen Sekundenabständen ein Sample genommen werden soll. Sobald man alles eingestellt hat, kann mit Klick auf den Button *Run* der durchlauf gestartet werden. Dadurch werden je nach Einstellung folgende Abläufe ausgelöst:

Einstellung *Database*: Im angegebenen Zeitintervall werden nacheinander folgende Schritte durchgeführt:

1. Die Kommandos zum Erstellen des aktuellen Bildes bzw. der aktuellen Objektdaten werden an die Library in der Game Engine geschickt.
2. Die Library ruft dann die entsprechenden, innerhalb der Game Engine enthaltenen Callback-Funktionen auf, in denen die Implementierungen des Benutzers die Daten aufbereiten und zurückgeben.
3. Diese werden dann von der Library an den Virtual Environment Su-

pervisor zurückgesendet, der sie dann abspeichert. Die jeweiligen Bild-Objektdaten-Paare erhalten denselben Zeitstempel, um sie zuordnen zu können. Derzeit werden die Daten im Filesystem im selben Ordner gespeichert, aus der der Virtual Environment Supervisor geöffnet wurde. Eine Anbindung an eine höherwertige Datenbank ist derzeit nicht implementiert.

Einstellung *Vision Engine*: Hier werden im angegebenen Zeitintervall folgende Schritte durchgeführt:

1. Das Kommando für das aktuell gerenderte Bild wird an die Library in der Game Engine gesendet.
2. Die Library ruft die entsprechende Callback-Funktion auf, wo die Implementierung des Benutzers das Bild erstellt und zurückgibt.
3. Das Bild wird dann von der Library an den Virtual Environment Supervisor zurückgesendet, welcher dieses gleich weiter an die Library in der Vision Engine schickt.
4. Dort wird implementierte Callback-Funktion `proceedImage()` aufgerufen, die das Bild verarbeitet und eine entsprechende Aktion zurückgibt.
5. Die Aktion wird dann über den Virtual Environment Supervisor an die Game Engine weitergeleitet.
6. In der Game Engine wird dann die erneut eine Callback-Funktion – `performAction()` – aufgerufen, die diese Aktion interpretiert und ausführt. Im implementierten Beispiel ist die Aktion entweder 0 oder 1. In der Callback-Funktion wird daher das Auto entweder gebremst oder beschleunigt.

Somit erfüllt das Framework sowohl die Funktionalität der reinen Datengenerierung als auch die der virtuellen Testumgebung für einen eigenen Computer Vision Algorithmus.

Kapitel 7

Fazit und Ausblick

In dieser Arbeit wurde gezeigt, dass es durchaus sinnvoll ist, Game Engines zur Erstellung von virtuellen Umgebungen und zur Generierung von Daten herzunehmen. Generell ist die Verwendung einer virtuellen Umgebung eine gute Alternative zu Setups in der realen Welt. Deren Erstellungsaufwand kann aber dennoch noch relativ hoch sein und meist ist ein gewisses Maß an computergraphischen Kenntnissen erforderlich. Die Verwendung einer Game Engine erleichtert aber die Erstellung einer solchen Umgebung um ein Vielfaches. Durch die bereits eingebauten Komponenten für Simulation, Physik, Beleuchtung, Steuerung etc. werden viele wichtige Funktionalitäten von der Engine übernommen. Des Weiteren sind die meist bereits in der Engine integrierten Metainformationen ein großer Vorteil. Diese können direkt zur Evaluierung hergenommen werden.

Das entwickelte Framework hatte das Ziel, den Benutzer dabei zu unterstützen, eine Game Engine als virtuelle Testumgebung aufzusetzen und möglichst viele Funktionalitäten zu kapseln, sowie eine möglichst hohe Austauschbarkeit der Komponenten zu gewährleisten. Über das Konzept der Callback-Funktionen müssen vom Benutzer nur die Funktionen implementiert werden, die je nach Engine unterschiedlich sind, der Rest wird vom Framework übernommen. Sollte die Game Engine oder die Vision Engine ausgetauscht werden, müssen so nur noch diese spezifischen Funktionen neu geschrieben werden. Die zwischen den Komponenten liegende Schnittstelle dient zur Kontrolle und Steuerung des Frameworks. Über diese kann der Benutzer in das Geschehen eingreifen und die Komponenten steuern. Die graphische Benutzeroberfläche bietet Funktionalitäten zur Verbindungsherstellung, zum Testen der Kommunikation und zum Starten des gesamten Ablaufs mit den gewünschten Einstellungen.

In Zukunft könnten vor allem in Hinblick auf das Framework noch einige Erweiterungen vollzogen werden. So wurde dieses im Zuge dieser Arbeit in der Programmiersprache C# entwickelt. Zum einen aufgrund der übersichtlichen API in Bezug auf Sockets und Windows-Formulare, zum anderen

weil die für das Beispiel verwendete Game Engine *Unity3D* ebenfalls C# unterstützt und daher das Framework hier gleich direkt verwendet werden konnte. Eine Implementierung in weiteren Programmiersprachen wir C++ oder Java wäre hier wünschenswert. Außerdem unterstützt das Framework im Moment ein paar fortgeschrittene Funktionalitäten noch nicht. Zum Beispiel arbeitet das Framework derzeit immer nur mit einem einzelnen Bild aus der Game Engine. Nachdem einige Computer Vision Algorithmen aber manchmal mit Stereo-Bildern arbeiten, könnte man hierfür eine entsprechende Funktionalität einbauen. Außerdem gibt es derzeit noch keine Anbindung an eine höherwertige Datenbank, um die generierten Daten abzuspeichern. Die Grundfunktionen des Frameworks, um eine Game Engine als virtuelle Umgebung verwenden zu können, sind jedoch allesamt vorhanden und die allgemeine Kommunikation der einzelnen Komponenten ist vollständig implementiert.

Anhang A

Inhalt der CD-ROM/DVD

Format: CD-ROM, Single Layer, ISO9660-Format

A.1 Masterarbeit

Pfad: /

Schmidt_Fabian_2013.pdf Masterarbeit (Gesamtdokument)

A.2 Projektdateien

Pfad: /project

/executables/veca.dll .	Kompilierte Library
/executables/ves.exe . .	Executable des Virtual Environment Supervisor
/source/*	Source Code der Library und des Virtual Environment Supervisor

A.3 Online-Quellen

Pfad: /web-pdfs

*.pdf	Kopien der Online-Quellen

Quellenverzeichnis

Literatur

[1] Luis von Ahn. „Games with a purpose“. In: *Computer* 39.6 (2006), S. 92–94.

[2] Luis von Ahn und Laura Dabbish. „Labeling images with a computer game“. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '04. ACM, 2004, S. 319–326.

[3] Luis von Ahn, Ruoran Liu und Manuel Blum. „Peekaboom: a game for locating objects in images“. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '06. ACM, 2006, S. 55–64.

[4] Martin Brunnhuber u. a. „Bridging the gap between visual exploration and agent-based pedestrian simulation in a virtual environment“. In: *Proceedings of the 18th ACM symposium on virtual reality software and technology*. VRST '12. ACM, 2012, S. 9–16.

[5] Wilhelm Burger und Mark Burge. *Principles of Digital Image Processing*. Springer-Verlag, 2009.

[6] Junqing Chen u. a. „Adaptive perceptual color-texture image segmentation“. In: *IEEE Transactions on Image Processing* 14.10 (Oct.), S. 1524–1536.

[7] Boguslaw Cyganek und Katarzyna Socha. „A multi-tool for ground-truth stereo correspondence, object outlining and points-of-interest selection“. In: *Proceedings of the 1st International Workshop on Visual Interfaces for Ground Truth Collection in Computer Vision Applications*. VIGTA '12. ACM, 2012, 4:1–4:4.

[8] D. Doermann und D. Mihalcik. „Tools and techniques for video performance evaluation“. In: *Proceedings of the 15th International Conference on Pattern Recognition*. Bd. 4, 167–170 vol.4.

[9] Andreas Fischer u. a. „Ground truth creation for handwriting recognition in historical documents“. In: *Proceedings of the 9th IAPR International Workshop on Document Analysis Systems*. DAS '10. Boston, Massachusetts: ACM, 2010, S. 3–10.

[10] David A. Forsyth und Jean Ponce. *Computer Vision – A Modern Approach*. Pearson, 2003.

[11] W.T. Freeman u. a. „Computer vision for computer games“. In: *Proceedings of the Second International Conference on Automatic Face and Gesture Recognition*. Oct, S. 100–105.

[12] E. Gamma u. a. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison Wesley Verlag, 2011.

[13] Cesar Isaza, Joaquin Salas und Bogdan Raducanu. „Synthetic ground truth dataset to detect shadows cast by static objects in outdoors“. In: *Proceedings of the 1st International Workshop on Visual Interfaces for Ground Truth Collection in Computer Vision Applications*. VIGTA '12. ACM, 2012, 11:1–11:6.

[14] Bernd Jähne und Horst Haußecker. *Computer Vision and Applications: A Guide for Students and Practitioners*. Academic Press, 2000.

[15] Sing Bing Kang, Yin Li und Xin Tong. *Image-Based Rendering*. Now Publishers, 2007.

[16] I. Kavasidis u. a. „A semi-automatic tool for detection and tracking ground truth generation in videos“. In: *Proceedings of the 1st International Workshop on Visual Interfaces for Ground Truth Collection in Computer Vision Applications*. VIGTA '12. ACM, 2012, 6:1–6:5.

[17] Gérard Medioni und Sing Bing Kang. *Emerging Topics in Computer Vision*. Prentice Hall, 2005.

[18] Julia Moehrmann und Gunther Heidemann. „Efficient annotation of image data sets for computer vision applications“. In: *Proceedings of the 1st International Workshop on Visual Interfaces for Ground Truth Collection in Computer Vision Applications*. VIGTA '12. Capri, Italy: ACM, 2012, 2:1–2:6.

[19] S. Nagabhushana. *Computer Vision and Image Processing*. New Age International, 2006.

[20] Oliver Schreer. *Stereoanalyse und Bildsynthese*. Springer-Verlag, 2005.

[21] Akhil R. Shah und Siddhartha R. Dalal. „Combinatorial enlargement of ground-truth datasets and efficient evaluation of segmentation algorithms“. In: *Proceedings of the 1st International Workshop on Visual Interfaces for Ground Truth Collection in Computer Vision Applications*. VIGTA '12. Capri, Italy: ACM, 2012, 12:1–12:4.

[22] Concetto Spampinato, Bas Boom und Jiyin He. „First International Workshop on Visual Interfaces for Ground Truth Collection in Computer Vision Applications“. In: *Proceedings of the International Working Conference on Advanced Visual Interfaces*. AVI '12. Capri Island, Italy: ACM, 2012, S. 812–814.

[23] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer-Verlag, 2010.

[24] Ákos Utasi und Csaba Benedek. „A multi-view annotation tool for people detection evaluation“. In: *Proceedings of the 1st International Workshop on Visual Interfaces for Ground Truth Collection in Computer Vision Applications*. VIGTA '12. ACM, 2012, 3:1–3:6.

Online-Quellen

[25] Phil Geib und Alex Bordens. *Cutting-edge sports statistics*. 2009. URL: http://www.sportvu.com/PDFS/Tribune112009.pdf.

[26] David Lowe. *The Computer Vision Industry*. 2012. URL: http://www.cs.ubc.ca/~lowe/vision.html.

[27] *Open Source Computer Vision Library*. 2013. URL: http://opencv.org/.

[28] *Unity3D Car Tutorial*. 2013. URL: http://www.gotow.net/andrew/blog/?page_id=78.

[29] Jeff Ward. *What is a Game Engine?* 2013. URL: http://www.gamecareerguide.com/features/529/what_is_a_game_.php.

[30] *mycryengine.com*. 2013. URL: http://mycryengine.com/.