



# Effiziente Modellierung von Computerspielen

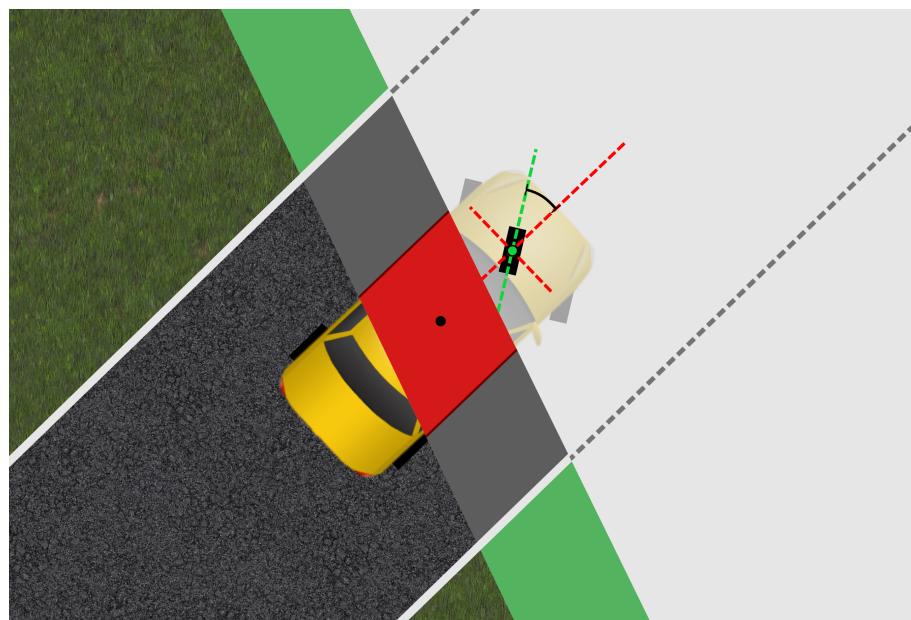
am Beispiel einer Autosimulation

---

Im Rahmen einer  
Besonderen Lernleistung im Fach Informatik

von  
Jan Frederik Bausch

betreut von  
Hr. Sommerbrodt und Hr. Dr. Ali Khan



Gymnasium Wöhlerschule  
Frankfurt am Main, 2015 - 2016

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>1</b>
1.1 Motivation.....	1
1.2 Konzept.....	1
1.2.1 Was ist ein Modell?.....	2
1.2.2 Grundsätzliche Ansätze der Modellierung.....	3
1.2.3 Was macht ein Modell effizient?.....	4
1.2.4 Beispielhafte Implementation eines Autorennspiels.....	5
<b>2 Wahl der Programmiersprache und Werkzeuge</b>	<b>6</b>
2.1 Programmiersprache.....	6
2.2 Bibliotheken, Frameworks und Engines.....	7
2.3 Weitere Werkzeuge.....	9
<b>3 Der Gameloop</b>	<b>9</b>
3.1 Unbekannte Framerate.....	11
3.2 Feste Framerate.....	12
3.3 Dynamische Framerate.....	12
3.4 Wahl des Gameloops.....	13
<b>4 Verwaltung von Entitäten</b>	<b>14</b>
4.1 Objektorientierter Ansatz mit flacher Hierarchie.....	14
4.2 Objektorientierter Ansatz mit erweiterter Hierarchie.....	16
4.3 Komponentenbasierter Ansatz.....	17
4.3.1 Kommunikation zwischen Systemen.....	23
4.3.2 Modellierung eines ECS in einer objektorientierten Programmiersprache.....	23
<b>5 Entwicklung einiger Module der Autosimulation</b>	<b>25</b>
5.1 Autophysik.....	25
5.1.1 Physik auf gerader Strecke.....	26
5.1.2 Kurven.....	28
5.2 Kollisionserkennung.....	30
5.3 Toneffekte.....	32
<b>6 Schluss</b>	<b>34</b>
6.1 Vergleich der beiden Architekturansätze.....	34
6.2 Einordnung in Entwicklungsprozesse und Designansätze.....	36
6.3 Fazit.....	38
<b>7 Anhang</b>	<b>40</b>

## **Erklärung**

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Besonderen Lernleistung, die anderen Quellen im Wortlaut oder dem Sinn nach entnommen wurden, sind durch Angaben der Herkunft kenntlich gemacht.

Frankfurt am Main, der 24.03.2016

Jan Frederik Bausch

# 1 Einleitung

## 1.1 Motivation

Mein Interesse für die Informatik liegt nicht zuletzt in der Faszination zu Videospielen. Während mich zunächst nur das Spielen selbst interessierte, wuchs mit der Zeit auch die Neugierde herauszu finden, wie ein solches Spiel aufgebaut ist und entwickelt wird. Dieser Frage möchte ich in dieser Arbeit nachgehen.

Dabei fasziniert mich besonders, dass die Spieleentwicklung einen großen Facettenreichtum besitzt, da sie viele Bereiche der Informatik, Kunst, Physik, Musik und Mathematik abdeckt. Allein als Programmierer kann man sich in größeren Studios auf zahlreiche Themen spezialisieren: Von der Engine-Entwicklung, über Netzwerkkommunikation und künstliche Intelligenz hinweg, bis hin zur Grafikprogrammierung gibt es ein breitgefächertes Aufgabenspektrum.

Vor dieser Arbeit habe ich schon Erfahrung mit der Spieleprogrammierung sammeln können. In den letzten Jahren habe ich bereits eine Handvoll Spiele programmiert. Allerdings fehlte es meist nach ein paar Wochen an der nötigen Motivation und keines dieser Projekte kam über den Status eines Prototypen hinaus.

Umso gespannter bin ich deswegen auf das Ergebnis dieser Arbeit, weil die Planung und Entwicklung eines Computerspiels von A bis Z für mich ein Novum ist.

## 1.2 Konzept

Im folgenden Abschnitt werde ich mein Konzept zu Beginn dieser Arbeit darlegen. Dies soll mir einerseits als Leitfaden bei der Entwicklung dienen und andererseits ermöglichen, ein abschließendes Resümee zu ziehen, bei dem ich mein ursprüngliches Konzept mit dem fertigen Modell und Spiel vergleiche.

Die Tatsache, dass meine ersten Projekte scheiterten ist nicht nur auf die fehlende Motivation, sondern auch auf die Planung zurückzuführen. Meistens schlich sich ein grundlegender Fehler in der Struktur des Programms ein, der eine reibungslose Entwicklung verhinderte. Aus diesem Grund möchte ich es mir in dieser Arbeit zum Ziel setzen, bekannte Entwurfsansätze von Videospielen zu verstehen und zu vergleichen.

Bevor wir tiefer in die Architektur eines Spiels einsteigen, machen wir uns zuerst mit grundlegenden Begriffen vertraut.

### 1.2.1 Was ist ein Modell?

Nicht nur in der Informatik, sondern auch in vielen anderen wissenschaftlichen Bereichen kommen Modelle zum Einsatz. Sie dienen dazu, meist reale Aspekte der Welt durch Abstraktion und Vereinfachung auf ein Abbild herunterzubrechen.<sup>1</sup> Jedes Modell ist dabei an einen bestimmten Zweck gebunden. Das bedeutet, dass es sich nur im Kontext dieses Zwecks verstehen und anwenden lässt.<sup>2</sup> So gibt es zum Beispiel viele verschiedene Atommodelle, die je nach physikalischem oder chemischen Kontext unterschiedliche Aspekte des Atoms darstellen.

Dank dieser Abstraktion sind Modelle in der Softwareentwicklung wichtige Hilfsmittel, um Softwareprobleme lösen zu können. Denn erst nachdem der Softwaredesigner ein Modell des Programms entworfen hat, können die Programmierer dieses in Quellcode umsetzen, ähnlich wie Bauarbeiter eines Hauses erst mit einem Architekturplan mit dem Bau beginnen können.

Bei der Entwicklung von Softwareprojekten gibt es verschiedene Vorgehensweisen, die man auch *Softwareentwicklungsprozesse* nennt.

Eine verbreitete Vorgehensweise ist es zum Beispiel, ein Projekt phasenweise anzugehen.

In dem sogenannten Wasserfallmodell wird der Entwicklungsprozess dazu in fünf Schritte unterteilt, die strikt nacheinander abgearbeitet werden. Nachdem die generellen Anforderungen des Projekts abgesteckt wurden, wird in der *Entwurfsphase* ein komplettes Modell des Programms entworfen. Das beinhaltet sowohl die Softwarearchitektur, sprich die Komponenten des Programms und ihre Beziehungen miteinander, als auch Programmablaufpläne, die detailliertere Prozesse eines Programms beschreiben. Solche Modelle lassen sich beispielsweise durch UML- und Nassi-Shneiderman-Diagramme grafisch darstellen. Erst wenn das Programm vollständig durchgeplant ist, folgen nach dem Wasserfallmodell die konkrete Umsetzung in Quellcode, Tests und schließlich Einsatz und Wartung.<sup>3</sup>

Das Wasserfallmodell ist für kleine Projekte ein intuitiver und leicht-verständlicher Ansatz. Je größer ein Projekt wird, desto wahrscheinlicher wird es, dass ungeahnte Probleme auftreten. So kann

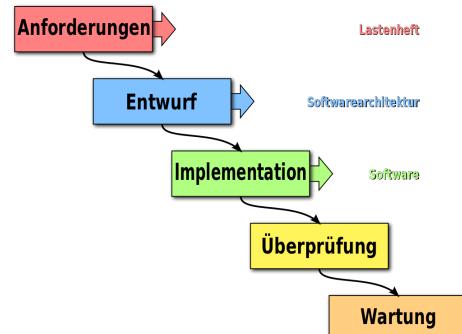


Abbildung 1: Wasserfallmodell

1 Vgl. <http://sgwww.cs.uni-magdeburg.de/~henry/publikation/Modellieren.pdf>, Zugriff am 26.02.2016

2 Vgl. [http://subs.emis.de/LNI/Proceedings/Proceedings08/DieVielfderModellInfo\\_14.pdf](http://subs.emis.de/LNI/Proceedings/Proceedings08/DieVielfderModellInfo_14.pdf), Seite 176, Zugriff am 22.02.2016

3 Vgl. <https://de.wikipedia.org/wiki/Wasserfallmodell>, Zugriff am 22.02.2016

Grafik: [https://commons.wikimedia.org/wiki/File:Waterfall\\_model-de.svg](https://commons.wikimedia.org/wiki/File:Waterfall_model-de.svg), Zugriff am 22.02.2016

zum Beispiel die Umsetzung einer Komponente schwierig sein oder der Kunde ist mit dem Endergebnis nicht zufrieden.

Ein anderer Entwicklungsprozess, der versucht, diesen Problemen früh entgegenzuwirken, ist das sogenannte *Prototyping*. Dabei werden Prototypen entwickelt, die nur zuerst einen Teil der Funktionalitäten des vollständigen Produkts darstellen. Auf Basis dieses Prototypen können Probleme frühzeitig erkannt werden, die bei der Entwicklung berücksichtigt werden. Entweder kann *evolutionär* auf dem vorherigen Prototypen aufgebaut werden, der Stück für Stück bis zum fertigen Produkt erweitert wird, oder es können *experimentelle* Prototypen entwickelt werden, um mögliche Lösungsansätze testen und vergleichen zu können.<sup>4</sup>

### 1.2.2 Grundsätzliche Ansätze der Modellierung

Da jedes Modell zweckgebunden ist, gibt es keinen allgemeingültigen Ansatz beim Entwurf eines Programms. Allerdings gibt es zwei unterschiedliche, fundamentale Perspektiven der Softwaremodellierung: *Top-Down- und Bottom-Up-Design*.

Beim *Top-Down-Design* liegt der Fokus zuerst darauf, die Softwarearchitektur zu planen und erst danach den Quellcode zu schreiben. Nach dem »teile und herrsche«-Prinzip wird das Programm dazu in kleinere Module aufgeteilt, die sich einem konkreten Teilproblem widmen. Jedes Modul wird solange unterteilt, bis das Problem leicht genug ist, um gelöst werden zu können.<sup>5</sup> Der Vorteil dieses Ansatzes ist es, dass die Grenzen des Modells vor der Implementation abgesteckt ist, was ein besseres Zeitmanagement erlaubt. Allerdings bietet der feste Rahmen wenig Raum für Flexibilität, wenn man das Modell später anpassen möchte.

Beim *Bottom-Up-Design* dagegen beginnt man schon früh mit der Programmierung von spezifischen Programmaspekten, die erst danach allgemeiner zusammengefasst werden. Deshalb spricht man davon, dass sich das Modell von unten nach oben entwickelt. Dadurch kann das Modell bei eventuellen Problemen bei der Implementation noch angepasst werden, macht aber die Planung schwieriger, weil »Auswüchse« im Modell entstehen können.<sup>6</sup>

Beide Ansätze vereint, dass man das Programm in Module aufteilt, die möglichst unabhängig voneinander sind. In der Realität kann es aber schwer sein, klare Trennlinien zwischen Modulen zu gewährleisten.<sup>7</sup>

4 Vgl. <http://www.enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/is-management/Systementwicklung/Vorgehensmodell/Prototyping/index.html>, Zugriff am 07.03.2016

5 Vgl. [https://de.wikipedia.org/wiki/Teile\\_und\\_herrsche\\_\(Informatik\)](https://de.wikipedia.org/wiki/Teile_und_herrsche_(Informatik)), Zugriff am 22.02.2016

6 Vgl. [https://en.wikipedia.org/wiki/Top-down\\_and\\_bottom-up\\_design](https://en.wikipedia.org/wiki/Top-down_and_bottom-up_design), Zugriff am 22.02.2016

7 Vgl. <http://ddi.cs.uni-potsdam.de/didaktik/Lehre/ADP1/Skriptum/kap6.pdf>, Seite 3, Zugriff am 22.02.2016

### 1.2.3 Was macht ein Modell effizient?

Um mögliche Modelle abschließend vergleichen und bewerten zu können, formuliere ich zuerst vier Kriterien, an denen ich ein Modell der Spieleentwicklung messe. Dazu habe ich mich auf die Punkte festgelegt, die meiner Meinung nach die größte Relevanz besitzen:

#### *Flexibilität*

Nicht nur die Informatik kennt das Problem, dass einem über die Zeit eines Projektes immer wieder neue Ideen und Funktionalitäten einfallen. Weil allein so viele Parteien, vom Grafikdesigner bis zum Vertriebsleiter, an größeren Spielen mitarbeiten, lässt es sich nicht nur auf dem Reißbrett planen. Umso dynamischer und flexibler ein Modell daher ist, desto besser kann man auf späte Probleme reagieren.

#### *Wiederverwendbarkeit von Quellcode*

In der Softwareentwicklung ist der Austausch zwischen Programmierern üblich. Wenn der Quellcode wiederverwendbar ist, dann müssen die gleichen Module nicht immer wieder neu entwickelt werden, sondern können aus anderen Softwareprojekten übernommen werden. Die Voraussetzung dafür ist die Unabhängigkeit der einzelnen Module, die nur über möglichst wenige Schnittstellen miteinander kommunizieren sollen.

#### *Kurzer Entwicklungszeitraum*

Besonders aus der Sicht von Spielestudios und Unternehmen ist der zeitliche Umfang des Entwicklungsprozesses von hoher Priorität, da eine längere Entwicklungszeit meist mit höheren Kosten verbunden ist. Das Top-Down-Verfahren etwa ist häufig zeit-effizienter, weil es eine klare Struktur vorgibt. So kann der Projektleiter genau festlegen, welche Programmierer bzw. welche Teams wie lange an einem Modul arbeiten sollen.

#### *Datenverarbeitungsgeschwindigkeit*

Auch die Laufzeit des Programms ist ein wichtiger Faktor vieler Spiele. Ein performantes Programm ist aus mehreren Gründen vorteilhaft. Einerseits benötigt ein Spiel dann weniger Ressourcen, was besonders auf leistungsschwachen, mobilen Plattformen relevant ist, andererseits können

dann auch mehr Rechenschritte pro Sekunde durchgeführt werden, wenn man etwa detailliertere Grafikeffekte zeichnen möchte.

Auch diese Kriterien sind natürlich nicht in allen Anwendungsfällen aussagekräftig, erfüllen aber die Anforderungen vieler Spiele. Denkbar wären darüber hinaus auch weitere Merkmale wie *Einsteigerfreundlichkeit für Programmieranfänger*, auf die ich in dieser Arbeit aber nicht eingehen werde.

### 1.2.4 Beispielhafte Implementation eines Autorennspiels

Insbesondere der letzte Gesichtspunkt, die *Flexibilität* während der Entwicklung, ließ sich für mich am besten bewerten, nachdem ich versucht habe, das Modell tatsächlich als Quellcode umzusetzen. Durch eine praktische Implementation eines Autorennspiels bin ich mir der Probleme und Hindernisse, die über das Projekt hinweg aufkommen können, bewusst geworden.

Als Projekt habe ich mir eine Autosimulation aus der Vogelperspektive vorgenommen, d.h. die Kamera blickt von oben auf die Straße und das Auto. Das Auto soll eine naturgetreue Fahrphysik besitzen, die auf die typischen Manöver aus anderen Rennspielen – *Lenken, Gas, Bremse* und *Rückwärtsfahren* – reduziert ist.

Die Umgebung besteht aus einem fest vorgegebenen Parcours, den der Spieler durchqueren muss. Neben einem Grasboden und der asphaltierten Straße können zudem Bäume, Mauern und Holzkisten als Hindernisse auf dem Boden oder der Straße stehen, denen der Spieler geschickt ausweicht.

Das Spiel besteht aus verschiedenen *Levels* bzw. *Welten*, die es zu bewältigen gilt. Ein Level ist dann gewonnen, wenn man die markierte Strecke unter einer gewissen Zeit gefahren ist. Erst wenn ein Level erfolgreich bestanden ist, wird das nächste freigeschaltet.

Neben dem Spiel selbst soll es ein Hauptmenü und ein Optionsmenü geben, in dem man verschiedene Einstellungen wählen kann.

Sollte ich nach diesen genannten Punkten noch mehr Zeit haben, könnte ich das Spiel um ein *Drift*-Manöver oder Partikel- und Spezialeffekte, wie zum Beispiel Reifenabriebspuren beim Bremsen ergänzen. Weiterhin wäre es denkbar ein Geldsystem einzuführen, mit dem man das Auto »tunen« oder neue Wagen kaufen kann. Geld könnte man gewinnen, wenn der Spieler ein Level besonders schnell bewältigt.

Das Programm soll auf den gängigen Windows-Betriebssystemen und einem mittelklassigem Computer laufen.

## 2 Wahl der Programmiersprache und Werkzeuge

Eine der ersten, aber auch weitreichendsten Entscheidungen, die man zu Beginn eines Softwareprojekts treffen muss, ist die Wahl der Programmiersprache und damit verbunden auch der Bibliotheken und Werkzeuge. Sie entscheiden darüber, auf welchen Geräten das Spiel läuft, wie viele Ressourcen es verbraucht und wie anspruchsvoll die Programmierung ist.

Die Wahl der Sprache kann einen großen Einfluss auf die Entwicklung haben, so dass ein einfacher Wechsel in der Mitte des Projekts mit großem Aufwand verbunden wäre. Umso wichtiger ist es daher, sich im Vorhinein ausführlich zu informieren.

### 2.1 Programmiersprache

Da glücklicherweise eine Vielzahl der eingangs erwähnten Prototypen mit unterschiedlichen Programmiersprachen verwirklicht habe, fiel es mir leichter eine Sprache zu finden, die auf meine Bedürfnisse zugeschnitten ist.

Eines meiner ersten Spiele habe mich Microsofts *C#* programmiert. Der Vorteil dieser Sprache ist die verständliche API, sprich die Schnittstelle zwischen Programm und Betriebssystem, mit der man beispielsweise auf den Bildschirm zeichnen, auf Dateien zugreifen oder Tasteneingaben empfangen kann.<sup>8</sup> Weitere Stärken sind die mächtige Entwicklungsumgebung »Visual Studio« und eine breite Nutzerbasis. Allerdings unterstützt Microsoft offiziell nur ihr eigenes Betriebssystem Windows - andere Desktopsysteme oder Smartphones können nur mit teils kostenpflichtigen Diensten von Drittanbietern erreicht werden.

Aus dem Informatikunterricht bin ich weiterhin mit *Delphi* bzw. *Free Pascal* vertraut. Im Gegensatz zu *C#* laufen mit Lazarus geschriebene Programme auch unter Mac OS und Linux, wiederum aber nicht auf mobilen Systemen. Hinzu kommt, dass man mit Delphi näher an der Hardware programmiert. Das hat zwar den Vorteil, dass man noch mehr Leistung aus dem Computer herausholen kann, ist aber aufwendiger in der Programmierung.

---

<sup>8</sup> Vgl. <https://de.wikipedia.org/wiki/Programmierschnittstelle>, Zugriff am 29.07.2015

Ich habe mich letztendlich für die Sprache *Haxe*<sup>9</sup> entschieden. Haxe hat seinen Ursprung in Adobes Programmiersprache *ActionScript*, die gerade bei Spieleentwickler für Adobe Flash beliebt war. Als Flash mehr und mehr von Javascript und Html5 verdrängt wurde, bildete sich Haxe als freie und quelloffene Alternative.

Der große Vorteil von Haxe ist die Möglichkeit, das Programm für eine Vielzahl von Plattformen zu entwickeln. Nach dem Prinzip »write once, run anywhere« (übersetzt »einmal schreiben, überall ausführen«) unterstützt Haxe alle Desktopsysteme, Android, iOS und das Web. Dies wird erreicht, indem der originale Quellcode des Programmierers automatisch in eine systemnahe Programmiersprache kompiliert wird, die dann nativ auf der Plattform läuft.

Verglichen zu den verbreiteten Programmiersprachen hat Haxe noch eine kleine Entwickler-Community. Da sich diese aber vor allem aus Videospielentwicklern zusammensetzt, gibt es in meinem Fall dennoch genügend Ressourcen und Hilfestellungen.

Neben den gerade genannten gibt es noch einige weitere Sprachen, wie zum Beispiel C++, Java oder Python, die bei Spieleentwicklern weit verbreitet sind. Meine Wahl fiel auf Haxe, weil ich mit der Syntax und API am besten vertraut bin, und ich das Modell demnach schnell in Quellcode umsetzen kann.

## 2.2 Bibliotheken, Frameworks und Engines

Nachdem man die passende Programmiersprache gefunden hat, kann man damit beginnen, die Software zu programmieren. Man hat nun zwar die Möglichkeit, das Spiel von 0 bis 100 komplett selbst umzusetzen, wird aber nicht in jedem Teilgebiet die nötige Expertise besitzen.

Softwareprobleme, die besonders häufig vorkommen, werden daher schon von anderen Programmieren gelöst. Ich kann diese »Unterprogramme« dann einbetten und sie über eine Programmierschnittstelle erreichen. Bei der genaueren Einordnung dieser Unterprogramme unterscheidet man zwischen *Bibliotheken* und *Frameworks*.

Bibliotheken sind kleinere Hilfsmodule, die ein sehr konkretes Softwareproblem lösen.<sup>10</sup> Frameworks dagegen nehmen eine aktiver Rolle ein. Sie geben eine klare Softwarearchitektur vor und legen einen einheitlichen Ordnungsrahmen fest.<sup>11</sup> Natürlich ist auch bei dieser Definition der Über-

---

9 Siehe <http://haxe.org/>

10 Vgl. <https://de.wikipedia.org/wiki/Programmbibliothek>, Zugriff am 29.07.2015

11 Vgl. <https://de.wikipedia.org/wiki/Framework>, Zugriff am 30.07.2015

gang fließend, da auch jede Bibliothek die Programmstruktur beeinflusst.

Auch die Wahl der Bibliotheken und Frameworks bedürfen einer abgewogenen Entscheidung. Im besten Fall nehmen sie dem Programmierer komplexe Probleme ab und erlauben ihm die reibungslose Entwicklung – im schlechtesten Fall nehmen sie ihm an den entscheidenden Stellen die Gewalt über das Programm.

Ich habe mich für *OpenFL*<sup>12</sup> entschieden. OpenFL ist ein Framework, das für die Entwicklung von Videospielen bestimmt ist und erleichtert mir die Arbeit bei zum Beispiel dem Laden von Bildern, Abspielen von Soundeffekten, Zeichnen der Grafikelemente und der Netzwerkkommunikation.

Weiterhin benutze ich *HaxeUI*<sup>13</sup> für die Benutzeroberfläche und *Ash*<sup>14</sup>, um das Modell eines *Entity-Component-System* umzusetzen. Eine genaue Erklärung des Entity-Component-Systems folgt in Kapitel 4.

Ein ganz anderer Ansatz wäre es, eine Spieleengine wie beispielsweise *Unity*, *Unreal Engine* oder die *Blender Game Engine* zu verwenden. Die meisten Engines unterscheiden sich von Bibliotheken und Frameworks in der Tatsache, dass sie Entwicklungsumgebung, Framework und Leveleditor in einem großen Softwarepaket vereinen. Sie mögen zwar für manche Unternehmen der effektivste Weg für die Entwicklung eines marktreifen Videospiels sein, unterschlagen aber den Aspekt einer effizienten Modellierung. Denn in den meisten Fällen muss ein Programmierer, der diese Engines benutzt, aus der Sicht der Softwarearchitektur nur sehr triviale Probleme lösen.

Deshalb habe ich mir bewusst gegen eine Engine und für die oben genannten Frameworks und Bibliotheken entschieden. Denn sie geben mir einerseits die Freiheit, mich nicht mit Hardware- oder Plattform-spezifischen Problemen zu befassen, lassen mir andererseits aber die Kontrolle über die entscheidenden Aspekte der Softwarearchitektur, auf die ich in den folgenden Kapiteln eingehe.

Im Anhang ist ein Kurzanleitung zur Installation von Haxe, OpenFL und der weiteren Bibliotheken zu finden.

## 2.3 Weitere Werkzeuge

Auch außerhalb der Programmierung selbst fallen Arbeiten an, beispielsweise bei der Bearbeitung von Bildern oder dem Erstellen von Levels, bei denen man externe Programme bzw. Werkzeuge be-

<sup>12</sup> Siehe <http://www.openfl.org/>

<sup>13</sup> Siehe <http://haxeui.org/>

<sup>14</sup> Siehe <http://www.ashframework.org/>

nötigt. Bei besonders ausgefallenen Spielen werden solche Werkzeuge vom Spieleentwickler selbst gebaut, damit sie genau auf Ansprüche des Spiels zugeschnitten sind. Im Falle meines einfachen Rennspiels bietet es sich an, auf bereits existierende Werkzeuge zurückzugreifen, um wieder Arbeit zu sparen.

Für die Fotobearbeitung verwende ich *Gimp*<sup>15</sup> und *Inkscape*<sup>16</sup>. Außerdem greife ich auf Bilder und Soundeffekte von *OpenGameArt*<sup>17</sup> zurück, die ich mit der entsprechenden Kennzeichnung kostenlos verwenden darf. Die Level und Rennstrecken erstelle ich mit dem Editor *GLEED2D*<sup>18</sup> und ich benutze *Sublime Text*<sup>19</sup> als Texteditor.

### 3 Der Gameloop

Nachdem die Rahmenbedingungen des Projekts abgesteckt sind, können wir uns mit der Struktur eines Spieles befassen.

Ein Spiel unterscheidet sich in einigen Aspekten von klassischen Anwendungen, wie man sie zum Beispiel aus dem Informatikunterricht kennt. Der vermutlich größte Unterschied ist meiner Meinung nach die Interaktivität zwischen Mensch und Computer. Viele Spiele der heutigen Zeit erinnern in ihrem Aussehen und ihrem Inhalt an Kinofilme - mit dem Unterschied, dass der Spieler jede Bewegung des Charakters steuert. Anstatt, dass der Benutzer wie bei typischen PC-Anwendungen alle paar Sekunden mit der Maus auf einen Knopf drückt, kann sich bei einem Videospiel schon die kleinste Mausbewegung auf das Spielgeschehen auswirken.

Dazu kommt, dass das Programm einen Überblick über die zahlreichen Spielobjekte behalten muss, die auf verschiedene Eingaben hören, sich bewegen und Abhängigkeiten untereinander besitzen.

Aufgrund dieser und vieler weiterer Besonderheiten haben sich in der Spieleentwicklung unterschiedliche Ansätze der Softwarearchitektur entwickelt, von denen ich ein Paar im Rahmen dieser Arbeit ausführen möchte. Unser erster Blick fällt dabei auf das Herzstück eines jeden Videospiels: Den Gameloop.

---

15 Siehe <http://www.gimp.org/>

16 Siehe <https://inkscape.org/de/>

17 Siehe <http://opengameart.org/>

18 Siehe <http://gleed2d.codeplex.com/>

19 Siehe <http://www.sublimetext.com/>

Klassischerweise orientiert sich der Programmablauf einer Anwendung an der Eingabe des Benutzers. Beobachten können wir das zum Beispiel an einer einfachen Taschenrechneranwendung: Wenn wir auf den »Ergebnis«-Button drücken oder die Enter-Taste auf der Tastatur betätigen, wird das Ergebnis berechnet und dem Benutzer angezeigt. Dieser Programmablauf folgt dem sogenannten *EVA-Prinzip*, d.h. zuerst erfolgt die **Eingabe** des Benutzer, die danach vom Programm **verarbeitet** und schließlich dem User **ausgegeben** wird.<sup>20</sup>

Das EVA-Prinzip lässt sich ebenso auch auf Spiele übertragen. Drückt man die Pfeilhoch-Taste, um Gas zu geben, berechnet das Spiel die neue Position des Autos und zeichnet zuletzt das Auto an die neue Position auf den Bildschirm.

Doch es gibt einen entscheidenden Unterschied zwischen Videospielen und klassischen Anwendungen. Auch wenn es keine Eingabe gibt, muss der Computer dennoch verarbeiten und die Spielwelt zeichnen. Das Auto kann weiter rollen, ohne dass wir Gas geben oder ein gegnerisches Monster kann uns angreifen, ohne dass der Spieler eine Eingabe macht. Würde sich der Programmablauf nur nach den Eingaben richten, wären jene Dinge schwer umsetzbar.

Daher erweitern wir das EVA-Prinzip um den sogenannten Gameloop, der auch Mainloop genannt wird. Der Gameloop ist sozusagen der Herzschlag des Programms, denn er gibt alle paar Millisekunden das Signal dazu, die Eingabe des Benutzers auszulesen, zu verarbeiten und schließlich die Spielwelt auf den Bildschirm zu malen. Wie der Name vielleicht schon verrät, handelt es sich dabei um eine Schleife, die allerdings endlos lang läuft. Diese Taktrate weist wieder Ähnlichkeiten zum Film auf, denn auch das Spiel generiert eine gewisse Anzahl von Bildern, die schnell abgespielt die Illusion einer Bewegung erzeugen.

Man kann sich vorstellen, was für einen großen Einfluss der Gameloop auf die Leistung eines Spiels hat. Wenn der Takt der Schleife verdoppelt wird, verbraucht auch das Programm die zweifache Rechenleistung. Daher unterscheidet man zwischen verschiedenen Typen von Gameloops, die ihre eigenen Stärken und Schwächen besitzen. Um die Auswirkung der unterschiedlichen Techniken besser zu verstehen, setzen wir jeweils immer das gleiche Fallbeispiel um: Ein Auto soll sich um 10 Pixel nach rechts bewegen, wenn wir die Pfeiltaste nach oben drücken.

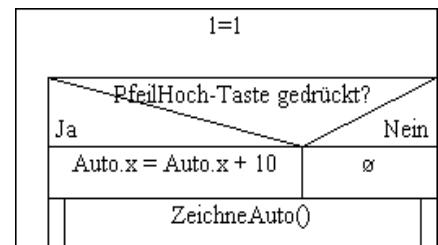
### 3.1 Unbekannte Framerate

Der erste Ansatz ist der Gameloop in seiner Reinform. In der Endlosschleife wird bei jedem Schleifendurchlauf geprüft, ob der Spieler die Pfeilhoch-Taste drückt und bewegt das Auto, falls die Be-

<sup>20</sup> Vgl. <http://www.grundlagen-computer.de/allgemein/eva-prinzip-eingabe-verarbeitung-ausgabe>, Zugriff am 01.08.2015

dingung wahr ist. Danach wird das Auto an der neuen Position gezeichnet und die Schleife wieder erneut durchlaufen. Kurz gesagt, der Gameloop wird so schnell wie möglich ausgeführt.

Dies führt zu einem ungewollten Effekt. Ein schneller Computer verarbeitet diese Schritte in einer kurzen Zeit. Als Folge würde das Auto ebenso schnell über den Bildschirm fahren. Führt man das selbe Programm auf einem leistungsschwächeren Gerät, beispielsweise einem Smartphone aus, fährt das Auto spürbar langsamer, da das Gerät für jeden Einzelschritt mehr Zeit benötigt. Eine solche Einschränkung ist bei einem Rennspiel, dessen Hauptmerkmal ein realistisches Fahrgefühl ist, nicht tolerierbar.



*Struktogramm eines einfachen Gameloop*

Diese Art eines Gameloop war fast ausschließlich auf dedizierten Plattformen, wie den ersten Homecomputern oder Spielekonsolen vertreten. Als Entwickler wusste man genau, wie schnell der Computer lief und man konnte sicher sein, dass das Programm nur auf diesem Computermodell läuft. Dies war auch der Grund dafür, dass spätere Modelle einer Gerätefamilie einen »Turbo«-Schalter besaßen. Wenn man auf einem neueren Gerät ein älteres Spiel spielen wollte, würde man den »Turbo«-Modus abschalten, um den Computer zu verlangsamen.<sup>21</sup>

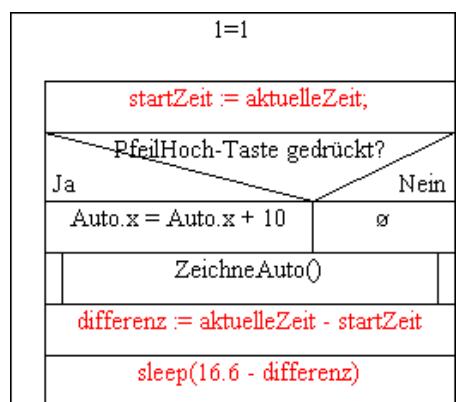
Spätestens aber mit dem Aufschwung von modularen PCs war die Rechenleistung so breit gefächert, dass diese Methode angepasst werden musste.

## 3.2 Feste Framerate

Die zweite Art versucht dieses Problem zu beheben. Anstatt, dass die Schleife so schnell wie möglich ausgeführt wird, soll sie nun in einer festen Taktfrequenz laufen. Ganz konkret bei zum Beispiel 60 Frames bzw. Schleifendurchläufe in der Sekunde dürfte ein Durchlauf nur  $1000\text{ms} / 60 \approx 16,6\text{ms}$  dauern.

Um dies zu erreichen, pausiert man die Schleife, wenn ein schneller Computer weniger Zeit für einen Durchgang benötigt. Braucht der Computer beispielsweise 6,6ms für die Abfrage der Tastatur, Verschieben des Autos und Zeichnen auf den Bildschirm, muss man noch weitere 10ms warten bis man mit dem nächsten Schleifendurchlauf beginnen kann.

Dazu misst man die Zeit ganz zu Beginn eines Schleifendurch-



*Feste Framerate*

<sup>21</sup> <http://gameprogrammingpatterns.com/game-loop.html#seconds-per-second>, Zugriff am 03.08.2015

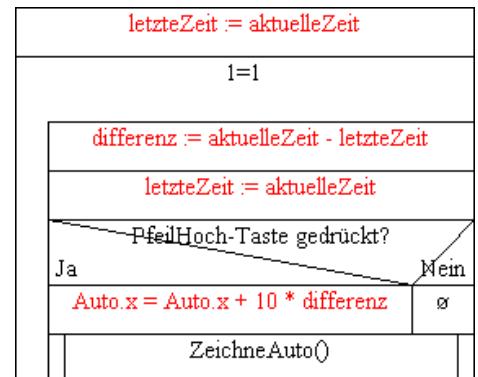
lauf und kurz nach den drei gerade genannten Arbeitsschritten. Die Differenz der beiden Zeiten abgezogen von der gewünschten Durchlaufdauer – hier 16,6ms – ergibt dann die Zeit, die man warten muss. Damit haben wir das Problem der ersten Art behoben. Da die Taktfrequenz auf allen Geräten gleich ist, fährt das Auto überall gleich schnell.

Natürlich ist es klar, dass wir mit dieser Methode nicht die ganze Leistung eines Computers ausschöpfen – schließlich verlangsamen wir ihn ja bewusst. Meistens reicht eine niedrige Framerate, wie 30 oder 60 FPS (»Frames per second«) aber auch schon, um ein realistisches Gefühl von Bewegung zu vermitteln. Zum Vergleich: Die meisten Kinofilme werden nur in 24 FPS<sup>22</sup> ausgestrahlt.

### 3.3 Dynamische Framerate

Entwickelt man aber traditionell für den Desktop-PC-Markt, möchte man die volle Rechenleistung ausschöpfen. Wir brauchen also einen Hybrid der die beiden vorherigen Ansätze miteinander verbindet, sodass die Schleife so schnell wie möglich ausgeführt wird, die Bewegung aber nicht verfälscht wird. Dazu passen wir diesmal nicht die Taktrate an die Bewegung, sondern die Bewegung an die Taktrate an.

Wieder messen wir die Dauer, die der Computer für Eingabe,



Dynamische Framerate

Verarbeitung und Ausgabe benötigt. Bei einer kurzen Zeit, also einem schnellen Computer, verlangsamen wir die Bewegung - ist die Bearbeitungszeit länger, wird auch das Auto mit größeren Schritten bewegt. Anders ausgedrückt: Bei 60 FPS wird das Auto jeden Schleifendurchlauf um 5 Pixel verschoben, bei 30 FPS um 10 Pixel.

Die Implementation sieht so aus: Man speichert die Bearbeitungszeit des *letzten* Durchlaufs (die Zeit des aktuellen Durchlaufs kann man ja anfangs nicht wissen) und multipliziert jede **Veränderung der Position** mit dieser Zwischenvariable. Allerdings ist es wichtig, dass man auch bei der weiteren Programmierung alle Bewegungsänderungen ebenso mit der Differenz/Bearbeitungszeit multipliziert. Als Folge ist jede Bewegungsänderung proportional zur Rechenleistung des Computers.

<sup>22</sup> <https://de.wikipedia.org/wiki/Bildfrequenz>, Zugriff am 04.08.2015

## 3.4 Wahl des Gameloops

Während der erste Typ des Gameloops aus offensichtlichen Gründen veraltet ist, gibt es zwischen Typ 2 und 3 nach meiner Einschätzung keinen qualitativen Unterschied. Hier muss man schlicht abhängig vom Anwendungsbereich abwägen, welcher Gameloop am besten geeignet ist.

Der Vorteil einer festen Taktrate ist, dass er dem Computer weniger Ressourcen abverlangt - schließlich reduzieren wir die Rechenoperationen unseres Programms ja bewusst. Bei Smartphone-Applikationen ist der Rechenaufwand entscheidend, da er erhebliche Auswirkungen auf den Stromverbrauch und damit auch auf die Akkulaufzeit des Geräts hat. Aus diesem Grund laufen viele Spiele auf mobilen Plattformen mit fester Framerate. Denn als App-Entwickler nimmt man also ein paar Bilder pro Sekunde weniger gerne in Kauf, wenn der Benutzer dafür länger spielen kann.

Der dritte Typ eignet sich dagegen besonders für Desktop-PCs, deren Stromverbrauch weniger relevant ist. Da man als Gamer mit einem 1500€-PC nicht auf die Framerate eines Smartphones begrenzt werden möchte, kann dieser Ansatz das ganze Potential des Rechners ausnutzen. Da die Rennsimulation nur auf Desktop-PCs laufen soll, habe ich mich dafür entschieden, den dritten Typ mit einer dynamischen Taktrate umzusetzen.

Die Implementation in der Simulation findet man in der Datei `app\Game.hx` ab Zeile 93. Hierzu sei aber noch folgendes gesagt:

Auf den ersten Blick fällt vielleicht auf, dass es in diesem Quellcode keine charakteristische Endlossschleife gibt, die ja eigentlich einen Gameloop ausmacht. Tatsächlich gibt es eine Schleife, diese wird aber nicht von meinem Quellcode sondern von dem Framework OpenFL initiiert. OpenFL ruft diese Routine dann in jedem Schleifendurchlauf über die Methode `onEnterFrame()` auf. Das hat den Grund, dass dieses synchron zu meinen Befehlen auch selbst Code ausführen will, beispielsweise um im Hintergrund Ressourcen zu laden.

Hier offenbart sich auch nochmal die Natur eines Frameworks (siehe Kapitel 2.2), einen Teil der Softwarearchitektur festzulegen. Nicht ich bestimme den Programmablauf, sondern das Framework ruft meinen Code auf.

## 4 Verwaltung von Entitäten

Mit unserem einfachen Gameloop haben wir jetzt eigentlich schon alle Mittel, um ein fertiges Spiel zu programmieren. Wir können jetzt neben unserem Auto auch Hindernisse und Straßen zeichnen oder in einer Bedingung vor jeder Bewegung des Autos prüfen, ob das Auto mit Hindernissen kollidiert.

dert.

Was in der Theorie so klar und einfach klingt, zeigt sich in der Praxis als Problem. Mit jeder weiteren Idee wird unser Gameloop um dutzende Zeilen länger bis wir uns irgendwann nicht mehr in unserem Dokument zurechtfinden können. Es bietet sich an, das Programm auf mehrere Dateien aufzuteilen.

An dieser Stelle führen wir einen neuen Begriff ein: »Eine Entität« oder engl. »an entity«. Eine Entität ist ein Sammelbegriff für ein Spieleobjekt eines Levels. Im Beispiel der Autosimulation sind Spieler, Straßen, Bäume, Mauern oder Grasfelder alles Entitäten – ganz unabhängig davon, ob sie statisch oder bewegt, sichtbar oder unsichtbar sind.

Wie auch im Fall des Gameloop gibt es bei der Verwaltung und Strukturierung dieser Entitäten verschiedene Ansätze, die ihre eigenen Stärken und Schwächen haben. Um die verschiedenen Typen besser voneinander abgrenzen zu können, setze ich ein sehr vereinfachtes Modell einer Autosimulation um.

## 4.1 Objektorientierter Ansatz mit flacher Hierarchie

Das Mittel der Wahl eines Programmierers, um ein Programm zu strukturieren, ist die *Objektorientierte-Programmierung* (kurz »OOP«). Das Programm wird in verschiedene *Klassen* unterteilt, die unterschiedliche Eigenschaften – *Attribute* genannt – und Prozeduren bzw. Funktionen – die *Methoden* heißen – besitzen. Klassen des Rennspiels könnten beispielsweise »Mauer« oder »Auto« sein.

Möchte man eine Klasse benutzen, erstellt man sich ein *Objekt* dieser Klasse. Diese Objekte besitzen dann tatsächlich Daten und Werte für die einzelnen Attribute, die kann man entsprechend beschreiben und auslesen kann. Ein Objekt der Klasse »Auto« könnte »PetrasAudi« mit einem Gewicht von 800 kg sein.

Darüber hinaus kann man Klassen auch Vererben. Diese Unterklassen – *Kinder* genannt – übernehmen die Attribute und Methoden ihrer *Eltern* und können darüber hinaus auch wieder eigene Attribute und Methoden hinzufügen. Die Klasse »LKW« könnte ein Kind der Klasse »Auto« sein und zudem das Attribut »Ladungsart« besitzen. Ein Objekt eines »LKW« könnte »ManisMAN« sein, dessen Gewicht 15000 kg und dessen Ladungsart »Möbel« ist.<sup>23</sup>

Mithilfe von Klassen und Vererbung können wir viele Fälle, sowie deren Daten und Beziehungen strukturiert darstellen. Und so auch unser Rennspiel.

23 Vgl. <http://www.itwissen.info/definition/lexikon/Objektorientierte-Programmierung-OOP-object-oriented-programming.html>, Zugriff am 06.08.2015

Wenn wir unser Spiel mit einem objektorientierten Ansatz umsetzen wollen, müssen also darüber Gedanken machen, wie wir die verschiedenen Spielobjekte in Klassen einsortieren können. Alle Spielobjekte haben gemeinsam, eine Position im Level zu besitzen und eine Größe zu haben. Anstatt nun also allen Klassen Spieler, Straße, Mauer, usw. Attribute für X, Y, Höhe und Breite zu geben, erstellen wir die Elternklasse »Entität« (siehe Abb. 2 unten). Spezifische Attribute, beispielsweise die Anzahl der Spuren einer Straße, werden aber nur in der Kinderklasse gespeichert.

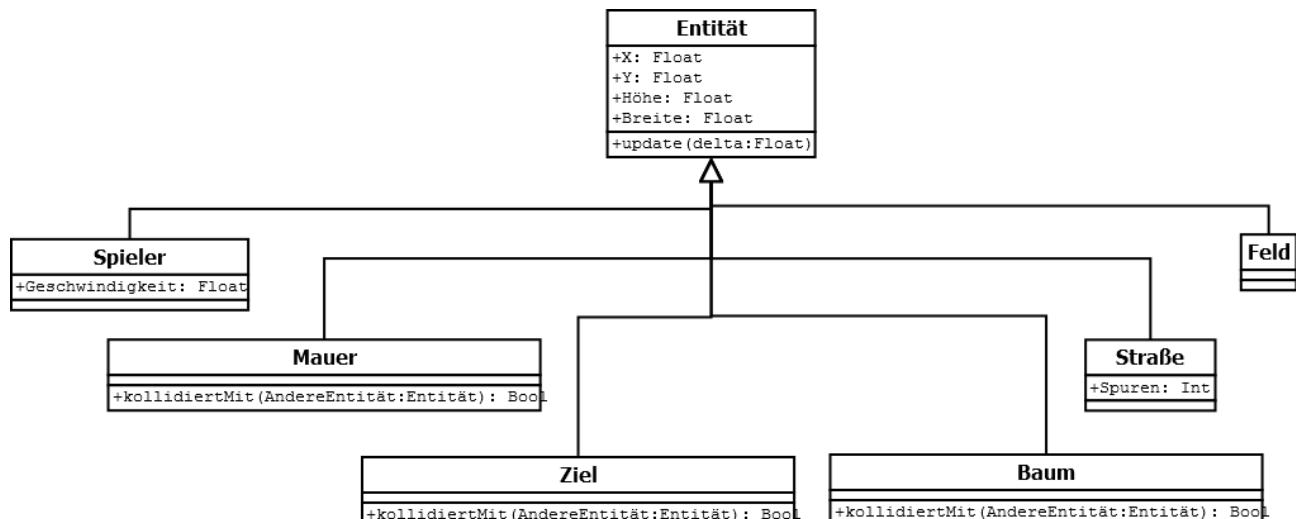
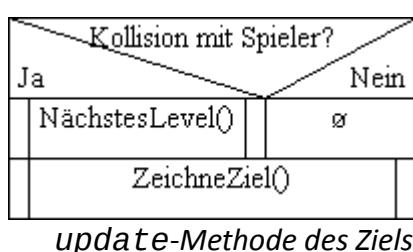
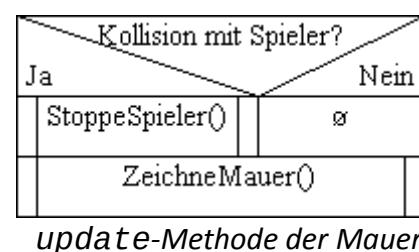


Abbildung 2: UML-Diagramm einer flachen Hierarchie

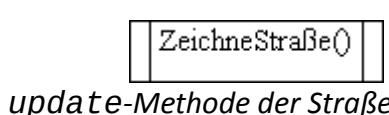
Neben den Attributen besitzt jede Entität aber auch noch eine **update**-Methode, die als Parameter die Zeitdifferenz besitzt, die wir im vorherigen Kapitel berechnet haben. Diese Methode wird jeden Schleifendurchlauf aufgerufen. Darin wird der Quellcode spezifisch für diese Entität ausgeführt. Hier ein paar Beispiele für **update**-Methoden:



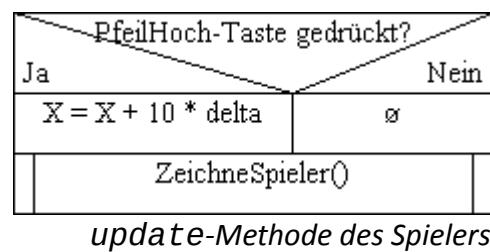
*update-Methode des Ziels*



*update-Methode der Mauer*



*update-Methode der Straße*



*update-Methode des Spielers*

Die Straße hat in diesem Beispiel selbst keine Aktion, deshalb besteht ihre **update**-Methode nur aus

einer Zeichenanweisung.

## 4.2 Objektorientierter Ansatz mit erweiterter Hierarchie

Mit der flachen Hierarchie kommt aber auch ein Problem, das wir hier bei zum Beispiel dem Ziel und der Mauer sehen können. Beide Entitäten prüfen ob, eine andere Entität mit ihnen kollidiert. Das bedeutet auch, dass in beiden Klassen eine Methode zu Kollisionserkennung stehen muss.

Für den Programmierer bedeutet das, dass er den Code immer hin und her kopieren muss. Findet er einen Fehler in der einen Methode, muss er ihn gegebenenfalls ebenfalls an einer Handvoll weiterer Stellen beheben. Generell ist es also nicht wünschenswert, wenn der selbe Code doppelt vorkommt.

Um diesem Problem entgegenzuwirken, suchen wir also wieder gemeinsame Eigenschaften und fassen diese in weitere Elternklassen zusammen. Damit hat unser Spiel folgende Struktur:

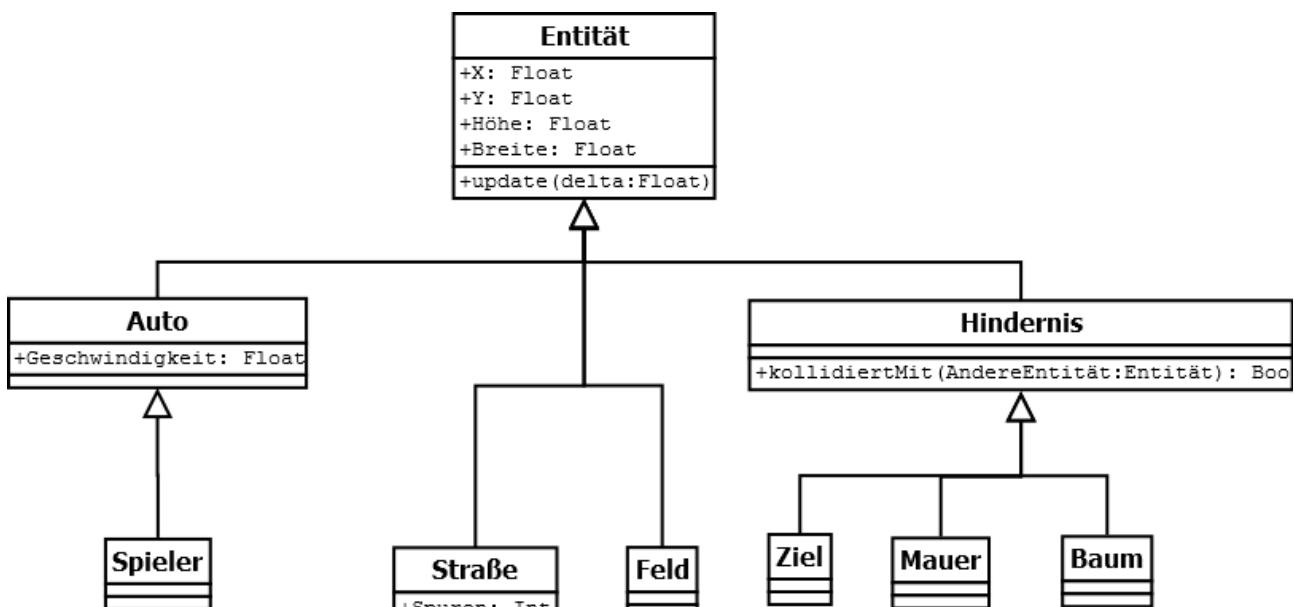


Abbildung 3: UML-Diagramm einer erweiterten Hierarchie

Entdeckt der Programmierer nun einen Fehler in der **kollidiertMit**-Funktion, muss er ihn nur in der Klasse Hindernis beheben. Klassen die schon spezifisch genug sind, wie hier Straße und Feld, erben weiterhin direkt von der Elternklasse Entität.

## 4.3 Komponentenbasierter Ansatz

Nicht nur die Informatik kennt das Problem, dass einem über die Zeit eines Projektes immer wieder neue Ideen und Funktionalitäten einfallen. Natürlich kann man auch ein Spiel nicht nur auf dem Reißbrett planen, sondern muss auch die Möglichkeit besitzen, im Laufe der Entwicklung neue Ideen zu verwirklichen.

Spielen wir einen solchen Fall einmal an unserem bisherigen Ansatz durch. Wir möchte zum Beispiel neben dem Spieler auch gegnerische Autos haben, die durch eine künstliche Intelligenz gesteuert werden. Selbstverständlich sollen die gegnerischen Autos die gleiche Fahrsimulation wie der Spieler besitzen, und der Spieler soll mit dem Gegner kollidieren können.

Damit haben wir ein Problem, denn die Klasse **Gegner** kann nicht gleichzeitig Kind von Auto und Hindernis sein.

Eine Möglichkeit wäre es, wieder die **kollidiertMit**-Funktion aus der Klasse **Hindernis** in die Klasse **Auto** zu kopieren:

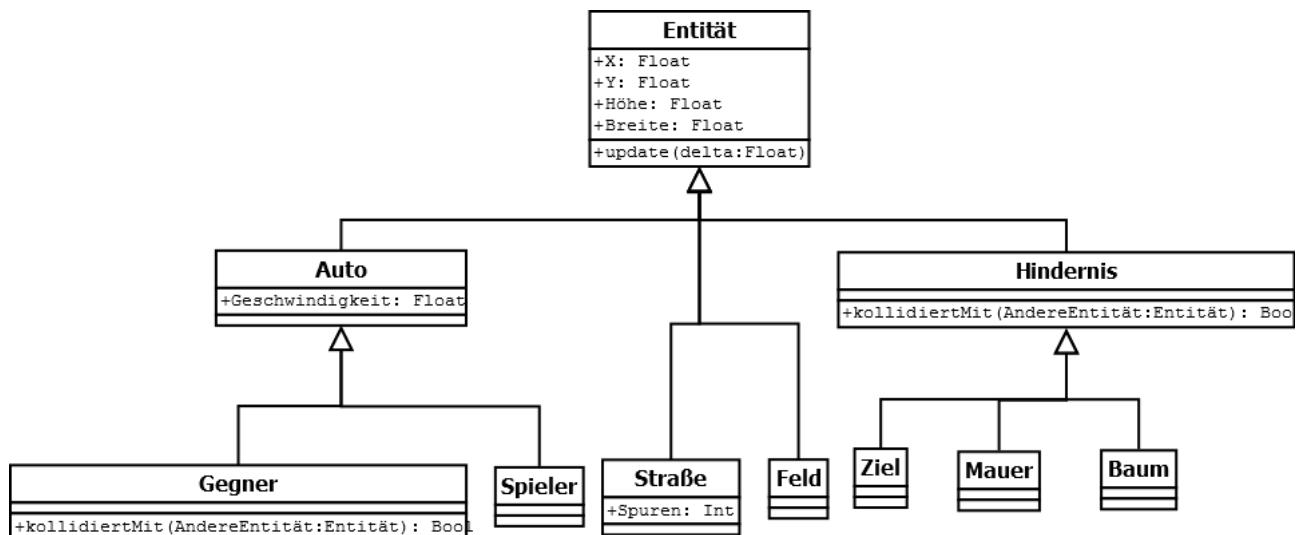


Abbildung 4: Quellcodedopplung

Damit stehen wir aber wieder vor der selben Problematik wie im ersten Fall, an mehreren Stellen den gleichen Quellcode pflegen zu müssen.

Eine weitere Möglichkeit ist, die **kollidiertMit**-Funktion in einer Hierarchie-Stufe höher, sprich in der **Entität**-Klasse, unterzubringen:

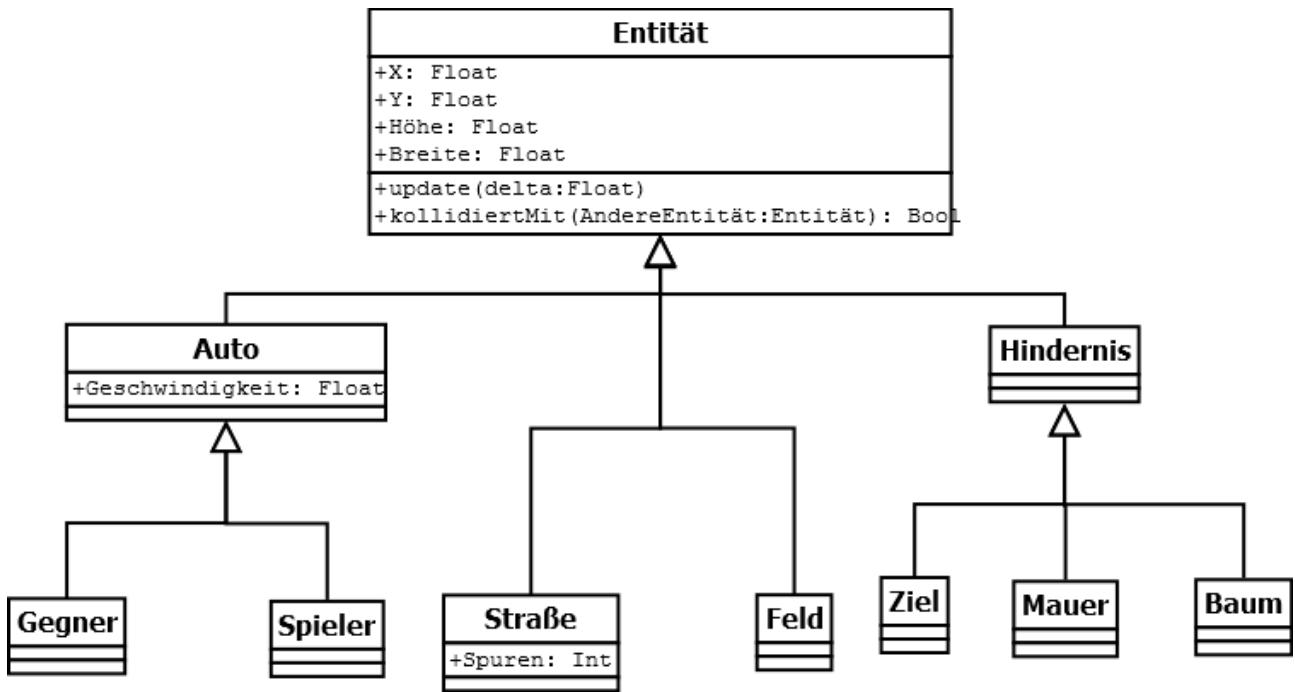


Abbildung 5: Höhere Hierarchie

Damit haben die Gegner-Klasse und alle Hindernisse darauf Zugriff und kein Quellcode kommt mehrfach vor. Allerdings erben diese Methode jetzt auch Entitäten, die diese Funktion nicht brauchen, wie hier der Spieler, die Straße und das Feld.

Je größer ein Spiel wird, desto komplexer wird das Entitätenmodell und die Beziehungen untereinander. Als Folge ist ein häufiges Phänomen von Spielen mit diesem Ansatz, dass Attribute und Methoden immer höher in der Hierarchie wachsen, bis sie schließlich in der Basisklasse Entität landen. Die Basisklasse der populären Spieleengine *Unreal Engine 3* besteht zum Beispiel aus ca. 3700 Zeilen Quellcode<sup>24</sup>.

Damit ist der eigentliche Hintergedanke, den Quellcode modularer und übersichtlicher zu gestalten, zunichtegemacht.

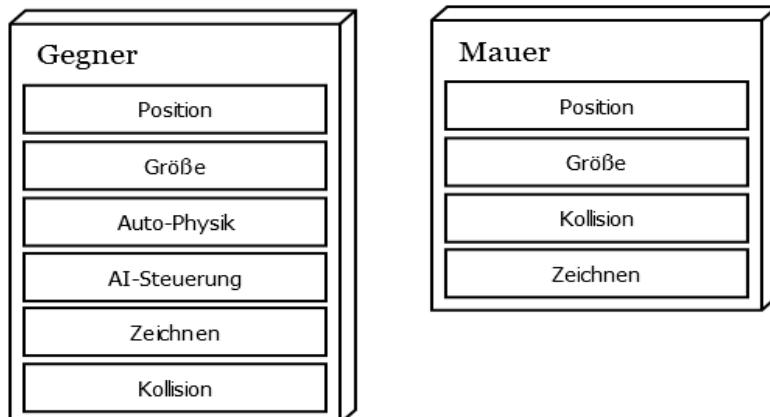
Aus dieser Problematik heraus bevorzugen immer mehr Spieldesigner in den letzten Jahren einen *komponentenbasierten* Ansatz. Eine weitverbreitete komponentenbasierte Softwarearchitektur ist das sogenannte *Entity-Component-System* (kurz ECS). Dabei teilt man eine Entität nach ihren Eigenschaften auf, die Komponenten genannt werden. Ein Spieler bestünde beispielsweise aus folgenden Komponenten:

<sup>24</sup> <http://www.heise.de/developer/artikel/Component-Based-Entity-Systems-in-Spielen-2262126.html>, Zugriff am 09.08.2015



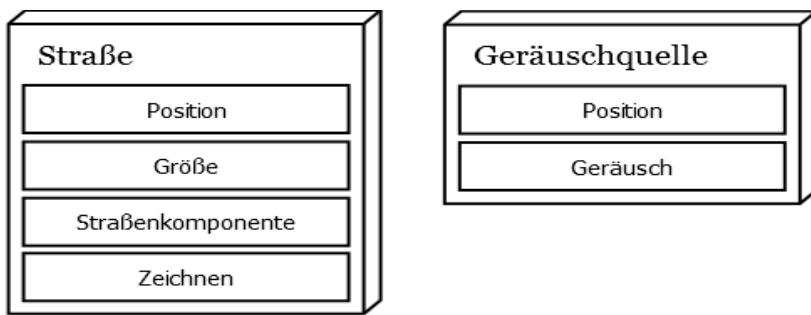
Komponenten bestehen selbst nur aus Attributen. Die Komponente Position besitzt die Attribute *X* und *Y*, die Komponente Größe besitzt die Attribute *Höhe* und *Breite* und in der Zeichen-Komponente könnte ein Bild speichern, das auf den Bildschirm gezeichnet wird.

Diese Komponenten kann man je nach Entitätentyp beliebig zusammenstellen. Der Gegner wird nicht mit der Tastatur, sondern durch eine AI gesteuert. Demnach wird diese Komponente ausgetauscht. Da er darüber hinaus mit dem Spieler kollidieren soll, bekommt er noch eine Kollisionskomponente.



Der große Vorteil eines Entity-Component-Systems ist aber, dass man sich auch ungewöhnlichere Entitäten zusammenbauen kann. Generell gilt aber, dass alle Daten einer Entität in Komponenten abgespeichert werden müssen und nicht, wie beim objektorientierten Ansatz, in der Klasse selbst. So muss etwa das Attribut *Spuren* der Straßen-Entität in eine eigene Komponente ausgelagert werden.

Natürlich ist keine Komponente per se notwendig. Wir können beispielshalber eine »Geräuschquelle«-Entität entwerfen, die an einer bestimmten Position einen Audioton abspielt. Nötig ist hier nur die Position- und eine Geräuschkomponente, nicht aber Größe oder eine Zeichenkomponente.



Im Moment steht das Entity-Component-System aber nur als abstrakte Vorstellung im Raum. Nun kommen wir dazu, wie es praktisch modelliert wird, um eine bessere Vorstellung davon zu gewinnen.

Das Entity-Component-System trägt seinen Namen nicht grundlos, denn es besteht aus **Entitäten**, **Komponenten** und **Systemen**. In einem ECS wird strikt zwischen Daten und Funktionen getrennt – nicht wie beim objektorientierten Ansatz, wo eine Klasse aus Attributen und Methoden besteht.

**Komponenten** speichern nur Daten. Eine Komponente kapselt einen Teilaспект bzw. eine Eigenschaft eines Spielobjekts ab. Eine Positions-Komponente spiegelt so etwa den Standort der Entität im Level wider, indem sie die Daten über X- und Y-Koordinate speichert. Der Teilaспект, den sie ab-kapselt, muss aber sinnvoll gewählt werden. Im Fall unseres konstruierten Beispiels oben sollte man nicht etwa auch noch Höhe und Breite in der Positions-Komponente speichern, da die »Geräuschquelle« nur X- und Y-Koordinate benötigt.

**Entitäten** sind kurz gesagt nur Behälter von Komponenten. Das heißt, dass eine Entität schlichtweg nur eine Liste von Komponenten speichert, die sie ausmacht. Wie schon vorher erwähnt, darf die Entität selbst keine Attribute tragen, sondern muss diese in Komponenten auslagern.

**Systeme** wiederum implementieren die funktionelle Seite. Auch sie sind wieder auf ganz bestimmte Funktionen des Spiels spezialisiert und tragen die eigentliche Logik des Spiels. Ein Beispiel im Rahmen unseres ausgedachten Rennspiels ist ein »Zeichen-System«, dass die Positions- und Zeichen-Komponenten nimmt und die Spielwelt damit auf den Bildschirm zeichnet. Jedes System besitzt daher die uns bekannte **update**-Methode.

Das ganze ECS wird meistens von einer **Engine** zusammengehalten, die alle Systeme und Entitäten mit ihren Komponenten speichert und verwaltet.

Besser versteht man das Konzept noch, wenn wir den Ablauf eines Spiels einmal im Kopf durchge-

hen.

Zu Beginn des Spiels werden der Engine zuerst alle Entitäten mitgeteilt. Ein Spieler also setzt sich aus einer Positions-Komponente mit X und Y gleich null, einer Größen-Komponente mit der Höhe 100 und der Breite 50, einer Auto-Physik-Komponente, einer Tastatur-Komponente und zuletzt einer Zeichen-Komponente mit dem Bild aus der Datei »images/auto.png« zusammen.

So fügt man nach und nach alle Entitäten eines Level ein. Damit ist die Vorbereitung beendet und wir können den Gameloop starten. Der Gameloop ruft jeden Schleifendurchlauf alle **update**-Methoden der Systeme auf, damit diese die Logik speziell für ihre Aufgabe ausführen können.

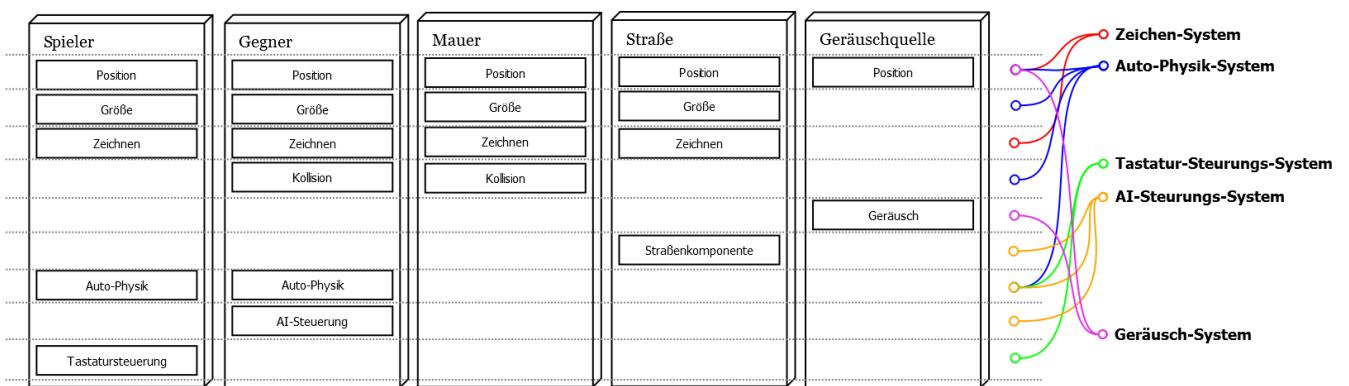
Die **update**-Methode des Zeichen-Systems würde folgendes machen: »Gib mir alle Entitäten mit einer Positions- und Zeichen-Komponente. Zeichne dann das Bild der Zeichen-Komponente an die Koordinate der Positions-Komponente.«

Das Tastatur-System würde diesen Quellcode ausführen: »Gib mir alle Entitäten mit einer Auto-Physik-Komponente und einer Tastatur-Steuerungs-Komponente. Wenn die Pfeilhoch-Taste gedrückt ist, setze die Variable Gas der Auto-Physik-Komponente auf true.«

Das Auto-Physik-System könnte folgendes ausführen: »Gib mir alle Entitäten mit einer Auto-Physik-Komponente, einer Positions-Komponente und einer Größen-Komponente. Gib mir außerdem eine Liste derer Entitäten mit einer Positions-Komponente, einer Größen-Komponente und einer Kollisions-Komponente und prüfe, ob jene Entität mit einer Entität dieser Liste kollidieren. Falls keine Kollision vorliegt, addiere die X-Koordinate der Positions-Komponente um 10 \* delta.«

Der Gameloop läuft solange, bis die Applikation beendet wird. Ebenso lange werden also auch die Systeme aufgerufen und können ihre Logik ausführen.

Einen besseren Überblick darüber, welche Beziehung die verschiedenen Komponenten und Systeme in unserem Beispiel haben könnten, bietet das folgende Diagramm:



### 4.3.1 Kommunikation zwischen Systemen

Auch wenn die einzelnen Systeme sehr voneinander abgekapselt werden sollen und nur sehr spezifische Aspekte der Spiellogik übernehmen, ist es trotzdem notwendig, dass sie hin und wieder miteinander kommunizieren. Im Kontext unseres Fallbeispiels könnte das *Auto-Physik-System* dem *Geräusch-System* Bescheid geben, wenn das Auto kollidiert, damit dieses ein Schadengeräusch abspielt.

In einem ECS geschieht das meistens über *Ereignisse* (engl. *Events*). Dazu definiert man zuerst verschiedene Ereignisse, beispielsweise »Auto kollidiert«, »Spiel gestartet« oder »Checkpoint aktiviert«. Die *Empfänger*, wie hier das Geräusch-System, *abonniert* dann ein bestimmtes Ereignis und wird benachrichtigt, wenn dieses vom *Sender* – dem Auto-Physik-System – ausgelöst wurde.

Natürlich kann es auch mehrere Sender und Empfänger für ein Ereignis geben. Vorstellbar wäre es zum Beispiel, dass auch noch das Zeichen-System ein Empfänger für das Event »Auto kollidiert« ist und bei einem Zusammenprall eine Explosion zeichnet.<sup>25</sup>

### 4.3.2 Modellierung eines ECS in einer objektorientierten Programmiersprache

Dennoch ist Haxe, so wie alle anderen zu Beginn vorgestellten Sprachen, eine objektorientierte Programmiersprache. Wie also kann man in ihnen ein Entity-Component-System modellieren?

An dieser Stelle müssen wir begrifflich genauer differenzieren, wo der Unterschied der beiden Ansätze liegt. Das Entity-Component-System und OOP widersprechen sich nicht, da das ECS eine Architektur ist, die auf dem objektorientierten Modell aufbaut. Sprachlich präziser ist deshalb, dass wir beim ersten Ansatz durch *Vererbung* und beim zweiten Ansatz durch *Komposition* modellieren.

Unser vereinfachtes Beispiel lässt sich folgendermaßen modellieren:

*Alle Grafiken sind auf der beigelegten CD in hoher Auflösung abgespeichert.*

---

<sup>25</sup> Vgl. <http://www.heise.de/developer/artikel/Component-Based-Entity-Systems-in-Spielen-2262126.html?artikelseite=2>, Zugriff am 21.10.2015

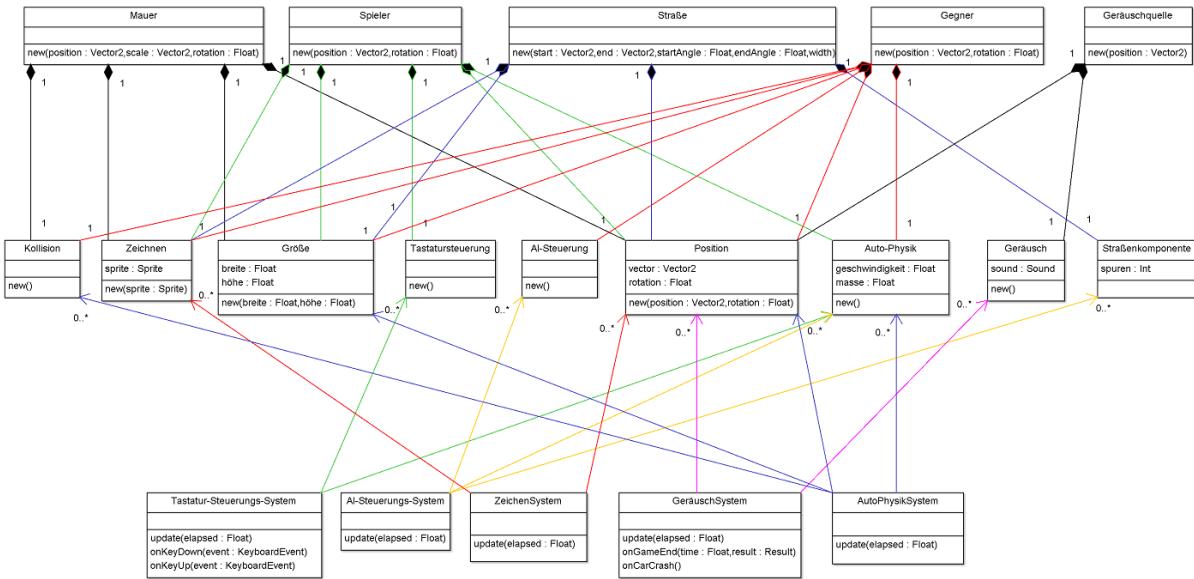


Abbildung 7: ECS in einer objektorientierten Programmiersprache

Auf den ersten Blickt wirkt diese Architektur deutlich komplizierter als der hierarchische Ansatz, al-lerdings lässt sie sich bei genauerer Betrachtung intuitiv verstehen.

Der Spieler zum Beispiel setzt sich aus den Komponenten *Zeichnen*, *Größe*, *Tastatursteuerung* und *Position* zusammen. Jede Entität ist also mit einer Aggregations-Beziehung mit den Komponenten verknüpft, die sie ausmacht.

Die Systeme dagegen besitzen eine assoziative Beziehung zu den Komponenten. Das Zeichensys-tem beispielsweise kennt alle Zeichen- und PositionsKomponenten und rendert die Textur der Zei-chenkomponente jeden Schleifendurchlauf an der Position der PositionsKomponente auf den Bild-schirm.

## 5 Entwicklung einiger Module der Autosimulation

Nachdem wir uns ausführlich mit Gameloops und der Verwaltung von Entitäten beschäftigt haben, widmen wir uns nun der speziellen Module der Autosimulation.

Auch den Modulen liegen Modelle zugrunde, die hier aber weniger die Softwarearchitektur, son-dern auch die Mathematik und Physik tangieren.

Da es allerdings schnell den Rahmen dieser Arbeit sprengt, würde ich Modul für Modul und Zeile für Zeile durchgehen, gehe ich besonders auf die interessanten Aspekte des Programms ein und kann diese dafür umso detaillierter beschreiben.

## 5.1 Autophysik

Bei der Entwicklung des Spiels hat sich es mir sich als große Schwierigkeit gezeigt, eine realistische Fahrphysik zu entwickeln. Klar ist, dass kein Computer der Welt in Echtzeit alle physikalischen Kräfte und Effekte simulieren kann, die auf das Auto wirken. Wieder einmal müssen wir also die Realität auf einfacheres Modell herunterbrechen, das weder dem Rechner noch dem Programmierer zu viel abverlangt.

Dabei stütze ich mich in diesem Kapitel auf das Paper »Car Physics for Games«<sup>26</sup> von Marco Monsler, sowie den Artikel »Simple 2D car steering physics in games« von *Engineering .NET*<sup>27</sup>.

Alle Entitäten, bei denen eine Autophysik simuliert werden soll, besitzen die Komponente *Vehicle* (siehe `app/components/Vehicle.hx`). In ihr sind Daten des Autos, wie Masse oder Radstand (der Abstand beider Achsen), Kräftekonstanten und Eingaben des Spielers - sprich Gas, Bremse und Lenkwinkel – gespeichert. Eine Liste mit den Werten der Konstanten befindet sich im Anhang. Die eigentliche Simulation geschieht dann aber im *VehicleSystem*, das mithilfe dieser Daten die neue Position und Rotation des Autos berechnet.

### 5.1.1 Physik auf gerader Strecke

Im ersten Schritt berechnen wir die Geschwindigkeit auf gerader Strecke, berücksichtigen also noch keine Kurven. In diesem Modell wird das Auto vereinfacht von zwei Kräften beeinflusst: Von Antriebskräften in Fahrtrichtung und von Widerstandskräften entgegen der Fahrtrichtung.

Die Antriebskraft  $F_{Antrieb}$  wird von der Eingabe des Spielers beeinflusst. Drückt der Spieler die Gas-Taste wird  $F_{Antrieb}$  auf eine konstante Zahl, die *Motorkraft*, gesetzt. Wenn die Brems-Taste gedrückt wird und die Geschwindigkeit des Autos über 0m/s liegt, wird die Antriebskraft um die *Bremskraft* subtrahiert. Wenn die Brems-Taste gedrückt ist und die Geschwindigkeit nicht über 0m/s

<sup>26</sup> Zu finden im Ordner »Referenzen« oder auf <http://www.asawicki.info/Mirror/Car%20Physics%20for%20Games/Car%20Physics%20for%20Games.html>, Zugriff am 28.10.2015

<sup>27</sup> <http://engineeringdotnet.blogspot.de/2010/04/simple-2d-car-physics-in-games.html>, Zugriff am 28.10.2015

liegt, wird die Antriebskraft um die *Rückwärtsgangkraft* subtrahiert. Je nach Eingabe kann  $F_{Antrieb}$  demnach positiv oder negativ sein.

Nur mit der Antriebskraft könnte das Auto aber unendlich beschleunigen. Als nächsten berücksichtigen wir deshalb zwei Widerstandskräfte, die relativ zur Geschwindigkeit des Autos wachsen: Der Rollwiderstand ist proportional zur Geschwindigkeit des Autos. Wir approximieren ihn mit

$$F_{Roll} = -C_{Roll} * v \quad \text{Zeile 58}$$

Der Luftwiderstand ist proportional zum Quadrat der Geschwindigkeit und wird deshalb besonders bei höheren Geschwindigkeiten spürbar. Er lässt sich vereinfacht durch

$$F_{Luft} = -C_{Luft} * v^2 \quad \text{Zeile 60}$$

beschreiben, wobei die Konstante  $C_{Luft}$  von der Fläche des Autos von vorne und der Luftpumpe beeinflusst wird.<sup>28</sup>

In einem Graphen kann man die Wirkung der Antriebs- und Widerstandskräfte am besten visualisieren:

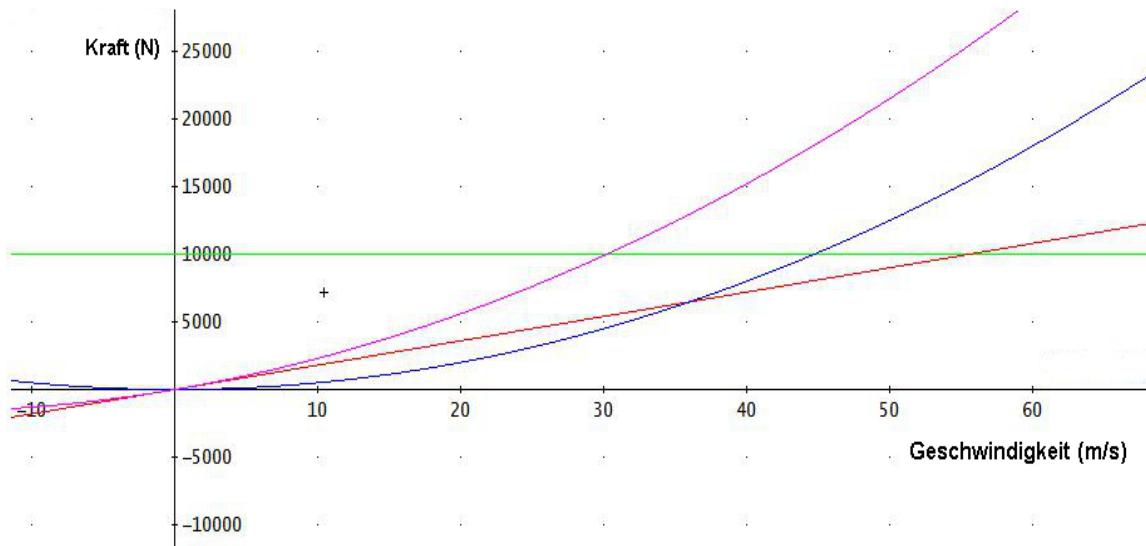


Abbildung 8: Kräfteausgleich bei höherer Geschwindigkeit

● Antriebskraft ● Rollwiderstand ● Luftwiderstandskraft ● Roll- und Luftwiderstand

Bei einer niedrigen Geschwindigkeit überwiegt der Rollwiderstand (rot), ab 35 m/s hat der Luftwiderstand (blau) eine größere Wirkung. Bei ungefähr 30 m/s = 108 km/h kreuzen sich die Graphen der summierten Widerstandskräfte (pink) und der Antriebskraft (grün). Das ist also die Höchstge-

schwindigkeit unseres Autos.<sup>29</sup>

Eine letzte Kraft, die im Spiel relevant ist, ist die »Booster«-Kraft  $F_{Boost}$ . Das ist die Kraft, die auf das Auto wirkt, wenn es über eine Boosterfläche fährt. Sie kommt aber natürlich nicht in der Realität vor. Zusammen wirken auf das Auto demnach folgende Kräfte:

$$F_{long} = F_{Antrieb} + F_{Roll} + F_{Luft} + F_{Boost} \quad \text{Zeile 69}$$

Mithilfe des zweiten Newtonschen Gesetzes lässt sich die Beschleunigung berechnen:

$$a = \frac{F_{long}}{m} \quad \text{Zeile 78}$$

Da diese Berechnungen in unserem Gameloop (siehe Kapitel 3.3) durchgeführt werden, kennen wir auch den Zeitunterschied zwischen den einzelnen Frames  $\Delta t$ , mit dem wir dann die Geschwindigkeit ermitteln.

$$v_{neu} = v_{alt} + \Delta t * a \quad \text{Zeile 69}$$

## 5.1.2 Kurven

Mit diesen paar Formeln haben wir bereits eine erstaunlich realistische Simulation für das Beschleunigungsverhalten eines Autos. Im nächsten Schritt befassen wir uns daher mit der Lenkung unseres Autos.

Je nach dem, wie realistisch die Fahrsimulation sein soll, ergeben sich bei Kurven ganz neue Problematiken. Denn bei hohen Geschwindigkeiten ist das Auto längst nicht mehr so stabil wie auf gerader Strecke. Beispielsweise kann es sein, dass das Auto über- oder untersteuert, sprich in eine Richtung fährt, obwohl die Räder in eine ganz andere zeigen.<sup>30</sup>

Noch komplizierter wird es beim Driften, was erreicht werden kann, indem man die Hinterräder kurzzeitig blockiert. Wenn man solche Phänomene naturgetreu abbilden möchte, muss man sich schnell mit Reifengrip, Winkelgeschwindigkeit, Schräglaufwinkeln und Gewichtsverlagerung befassen.

Spätestens hier sind meine physikalischen Kenntnisse aber bei weitem überschritten. Deshalb habe ich mich dazu entschieden, ein vereinfachtes Modell umzusetzen. Der Vorteil ist, dass es einfach zu verstehen ist und weniger Rechenkapazitäten benötigt – allerdings muss das Feature Driften leider

---

29 Vgl. »Car Physics for Games« Seite 3

30 Vgl. »Car Physics for Games« Seite 14

wegfallen. Da das Auto nicht ausbrechen kann, scheint es bei höheren Geschwindigkeiten praktisch auf der Straße zu kleben.

Letztendlich geht es mir in diesem Projekt aber nicht darum, dass jede noch so kleine Kraft physikalisch-perfekt simuliert wird, sondern dass der Spieler unterhalten wird. Deswegen bin ich mit diesem Modell sehr zufrieden.

Die erste Vereinfachung bei diesem Ansatz ist, das Auto auf das sogenannte *Bicycle Model* herunterzubrechen. Dabei geht man davon aus, dass das Auto wie ein Fahrrad nicht vier, sondern zwei Räder besitzt. Ausgehend von der Position des Autos, die immer den Mittelpunkt der Entität markiert, bestimmt man zuerst die Position des Vorder- und Hinterrads.<sup>31</sup>

Das Vorderrad befindet sich logischerweise in Fahrtrichtung des Autos (sprich der Rotation der Entität) in einer Entfernung von  $\frac{1}{2}$  Radstand:

```
var frontwheel:Vector2 = position.vector + Vector2.fromPolar(position.rotation, vehicle.WHEELBASE/2);
```

Das Hinterrad dementsprechend gegen die Fahrtrichtung:

```
var backwheel:Vector2 = position.vector + Vector2.fromPolar(position.rotation+180, vehicle.WHEELBASE/2);
```

Die Funktion `Vector2.fromPolar(a,b)` gibt einen Vektor zurück, der mit der Y-Achse den Winkel a einnimmt und die Länge b besitzt.

Im nächsten Schritt bewegen wir beide Räder um die eben ausgerechnete Geschwindigkeit (blauer Pfeil). Das Hinterrad wird in Fahrtrichtung des Autos und das Vorderrad in Fahrtrichtung addiert mit dem Lenkwinkel bewegt.

```
frontwheel += Vector2.fromPolar(position.rotation+vehicle.steerAngle, vehicle.velocity * 200/5 * elapsed);
backwheel += Vector2.fromPolar(position.rotation, vehicle.velocity * 200/5 * elapsed);
```

Verwunderlich ist die Multiplikation mit  $200/5$ . Alle bisherigen Variablen, so auch die Geschwindigkeit des Autos, wurden im Einheitsmaß Meter berechnet. Die Position aller Entitäten werden im

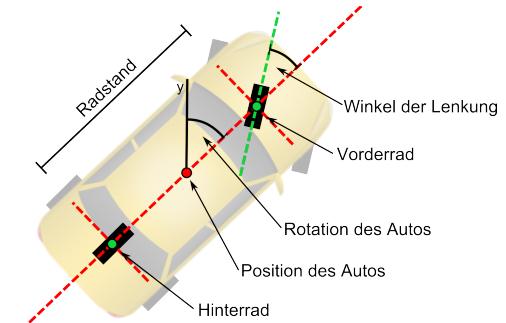


Abbildung 9: Bicycle Model

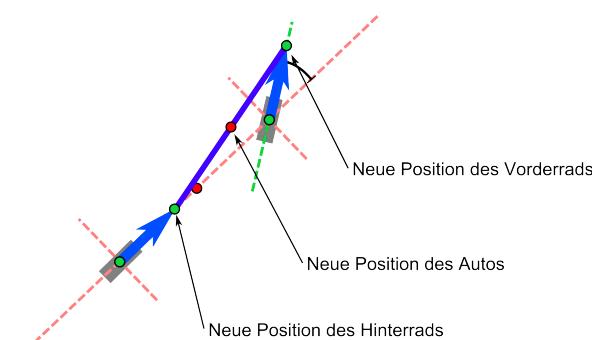


Abbildung 10: Verschiebung der Räder

<sup>31</sup> Vgl. <http://engineeringdotnet.blogspot.de/2010/04/simple-2d-car-physics-in-games.html>, Zugriff am 28.10.2015

Spiel aber in Pixel gespeichert. Deswegen muss die Geschwindigkeit von m/s in »Pixel/s« umgerechnet werden und wird deshalb mit 200/5 multipliziert (da 5 Meter ungefähr 200 Pixeln entsprechen).

Zum Abschluss berechnen wir den Vektor zwischen der neuen Position des Hinterrads zur neuen Position des Vorderrads. Tatsächlich entspricht der Winkel dieses Vektors der neuen Rotation unseres Autos:

```
var rotation: Float = backWheel.angleTo(frontWheel);
```

und die Mitte zwischen der neuen Position des Hinterrads und der neuen Position des Vorderrads ist die neue Position des Autos<sup>32</sup> (siehe Abb. 10):

Mit diesen einfachen geometrischen Tricks haben wir das Lenkverhalten eines Autos nachgebildet.

## 5.2 Kollisionserkennung

Nun kann sich unser Auto naturgetreu in der Spielwelt bewegen.

Nur würde kein Rennspiel richtig Spaß machen, wenn man einfach durch Mauern, Hindernisse und Ziele hindurchfahren kann. Deswegen müssen wir als nächstes erkennen können, ob zwei Spielobjekte aufeinanderprallen oder nicht.

Neben unserem bisherigen *VehicleSystem* brauchen wir demnach ein *CollisionSystem*. Immer wenn das VehicleSystem das Auto bewegen möchte, fragt es das CollisionSystem, ob die Bewegung möglich ist.

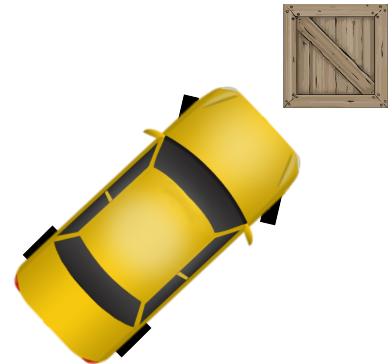


Abbildung 11: Kollidieren Auto und Box?

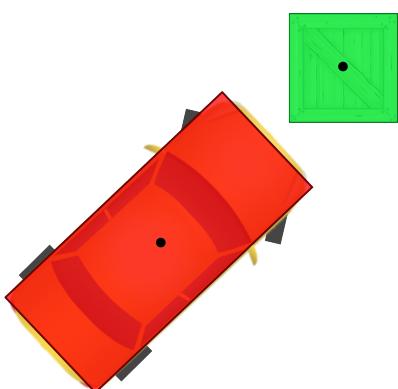


Abbildung 12: Spielobjekte als rotierte Rechtecke darstellen

Schauen wir nun, wie das CollisionSystem eine Kollision erkennt.

Jede Entität, die kollidieren kann, besitzt die Komponente *Collision* (siehe `app/components/Collision.hx`). In dieser Komponente ist die Form der Entität beschrieben. Da eine sehr komplexe Form ebenso komplizierte Algorithmen und aufwendige Rechenoperationen nach sich ziehen würde, habe ich mich dafür entschieden, die Entitäten durch **rotierte Rechtecke** darzustellen. Dies bedeutet allerdings, dass beispielsweise die Form des Autos

32 Vgl. <http://engineeringdotnet.blogspot.de/2010/04/simple-2d-car-physics-in-games.html>, Zugriff am 28.10.2015

nicht perfekt repräsentiert werden kann. Dennoch ist diese Darstellung genau genug und kostet weniger Rechenleistung, denn man muss bedenken, dass man mehr als 30 mal in der Sekunde auf Kollisionen überprüft.

Da aber selbst der Erkennungsalgorithmus für zwei rotierte Rechtecke rechenintensiv ist, wird die Form im ersten Schritt noch weiter abstrahiert. Wenn sich schon diese noch größeren Formen nicht schneiden, muss gar nicht erst weiter gerechnet werden. Dazu werden die zwei Objekte als Kreise dargestellt. Der Kreismittelpunkt ist die Position des Autos und der Radius entspricht dem Abstand zwischen dem Mittelpunkt und dem entferntesten Punkt des Rechtecks.

Die Vorteile: Erstens ist die Kollisionsprüfung sehr schnell (Die Radien beider Kreise addiert müssen kleiner sein als der Abstand der Mittelpunkte) und zweitens muss dieses Modell noch nicht die Rotation des Spielobjektes berücksichtigen.

Tatsächlich schneiden sich die Kreise in der Abbildung nicht, deswegen könnte das Spiel die nächsten Schritte einfach überspringen.

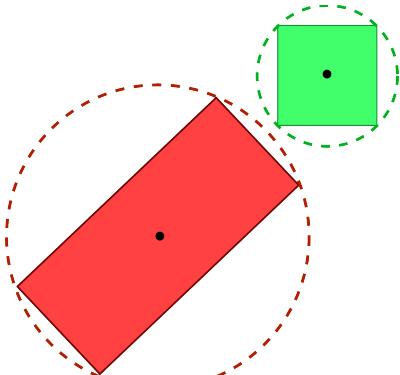


Abbildung 13: Grobe Kollisionsprüfung

Wenn sich die Kreise aber überlappen, kommt man um eine genauere Prüfung nicht herum. Im Falle der rotierten Rechtecke habe ich mich für einen Algorithmus entschieden, der auf das **Separating Axis Theorem** aufbaut. Es besagt, dass sich zwei Rechtecke nicht schneiden, wenn man eine Gerade findet, die zwischen beiden liegt, bzw. die beide trennt.<sup>33</sup>

Zuerst findet man dazu die *Achsen* der beiden Rechtecke. Diese Achsen sind schlicht Richtungsvektoren, die parallel zu je einer Seite des Rechtecks stehen. Ein Dreieck beispielsweise hätte drei und ein n-Eck hätte dementsprechend n Achsen.

Das Rechteck hat im Gegensatz die Besonderheit, dass sich immer zwei Seiten parallel gegenüber liegen. Aus diesem Grund brauchen wir nur zwei Achsen. Für unsere zwei Rechtecke haben wir deshalb vier Achsen gefunden (hier  $x_1$ ,  $y_1$ ,  $x_2$  und  $y_2$  genannt):

<sup>33</sup> [http://wiki.delphigl.com/index.php/Tutorial\\_Separating\\_Axis\\_Theorem](http://wiki.delphigl.com/index.php/Tutorial_Separating_Axis_Theorem), Zugriff am 22.10.2015

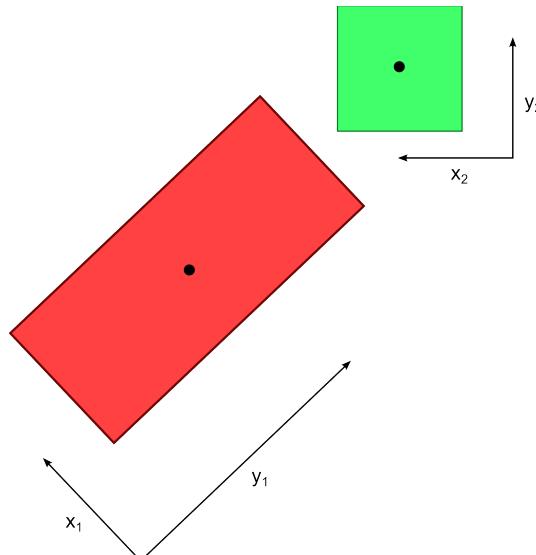
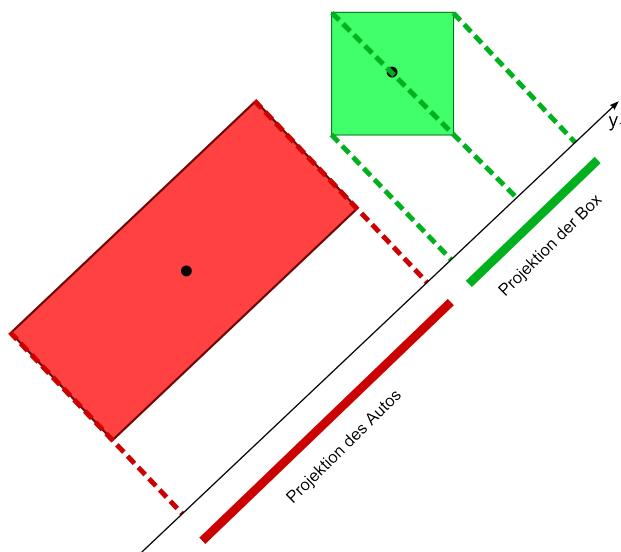


Abbildung 14: Achsen finden

Den darauffolgenden Schritt müssen wir dann **für jede Achse** – sprich viermal – ausführen: Wir projizieren alle Eckpunkte beider Rechtecke auf die aktuelle Achse. Das machen wir, indem wir den *Ortsvektor* eines Eckpunktes mit dem *Skalarprodukt* auf den normierten *Richtungsvektor* der Achse projizieren. Als Ergebnis bekommt man für beide Rechtecke je eine Menge an Stellen auf der aktuellen Achse, die die Projektion der Rechtecke abgrenzt.

Abbildung 15: Projektion auf Achse  $y_1$ 

Hat man **eine einzige Achse** gefunden, bei der sich die Projektionen beider Rechtecke nicht überschneiden (wie hier die Achse  $y_1$ ), so ist bestätigt, dass die **beiden Rechtecke nicht kollidieren**.

Unser Ergebnis können wir dann wieder an das VehicleSystem zurückgeben, das damit entscheidet, ob es das Auto bewegt.

## 5.3 Toneffekte

Mit einer guten Physiksimulation kommt man einem realistischen Spielgefühl schon sehr nahe. Viel unbewusster, aber kaum weniger effektiv, wirken daneben Toneffekte, weil sie dem Spieler schnell und direkt Rückmeldungen geben können. Zum Beispiel kann der Spieler kaum wahrnehmen, wenn Spielobjekte etwas schneller an der Kamera vorbeifliegen. Wenn er aber das »Aufheulen« des Motors hört, versteht er sofort, dass das Auto beschleunigt.

Selbst kann man dieses Phänomen einmal erfahren, wenn man die Toneffekte in den Optionen ausstellt. Ohne Motorengeräusche wirkt das Auto dann langsamer, ohne Blechschadentöne ist die Kollision nur halb so gefährlich und ohne Siegesfanfare ist selbst die Bestzeit kaum mehr zufriedenstellend.

Falls das noch nicht überzeugend genug sein sollte, ist zudem auch die Implementation vieler grundlegender Soundeffekte sehr einfach. Da das Spiel schon eine Handvoll Ereignisse (siehe Kapitel 4.3.1) kennt – so beispielsweise »Entity kollidiert« oder »Countdown startet« – kann das *Sound-System* diese ganz einfach abonnieren und dann entsprechende Töne abspielen.

Komplizierter dagegen sind die Motorengeräusche, weil sie erst realistisch klingen, wenn sie sich abhängig zur Geschwindigkeit verändern. Aber auch hier konnte ich mich eines einfachen Tricks bedienen, um das Gehör zu täuschen. Dafür brauche ich vier kurze Audioschnipsel, die sich in der Endloswiederholung abspielen lassen können: Einmal das Motorengeräusch im Standgas, bei niedriger Geschwindigkeit, bei mittlerer Geschwindigkeit und bei der Höchstgeschwindigkeit. Im Spiel stelle ich die Lautstärke jeder Spur dann abhängig zur Geschwindigkeit des Autos ein. Steht es zum Beispiel still, wird allein die Standgas-Spur mit voller Lautstärke abgespielt. Bei 5 m/s wird die Standgas-Spur zu 50 % und die Spur der niedrigen Geschwindigkeit zu 50 % Lautstärke abgespielt, während die anderen Spuren noch nicht zu hören sind. Damit gibt es einen glatten Übergang von einem Motorengeräusch zum nächsten.

Die Parameter bei der Tonabmischung, sprich zu wie viel Prozent eine jeweilige Spur bei einer bestimmten Geschwindigkeit hörbar ist, habe ich durch schlichtes Ausprobieren gefunden. Am besten lassen sie sich durch folgendes Diagramm visualisieren:

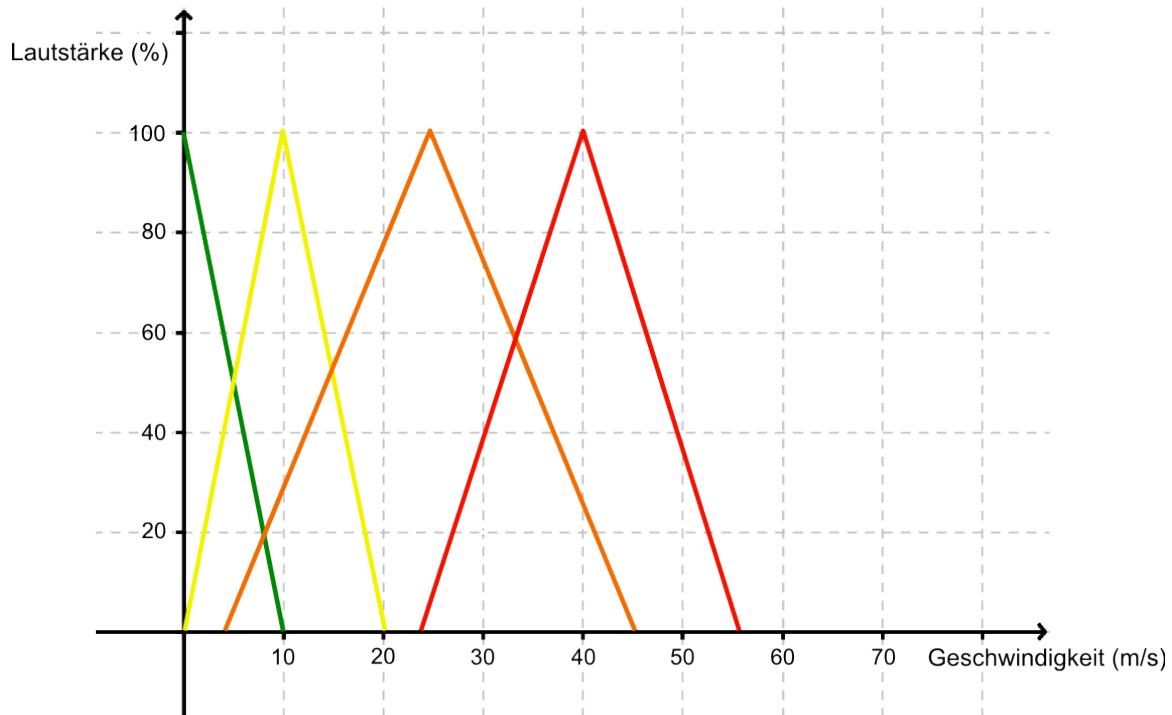


Abbildung 14: Tonmischung der Motorengeräusche  
 • Standgas • Niedrige Geschw. • Mittlere Geschw. • Hohe Geschw.

## 6 Schluss

Ich habe während meiner Besonderen Lernleistung verschiedene Architekturansätze und Entwurfsmodelle kennengelernt, die ich abschließend vergleichen und nach ihrer Effizienz bewerten möchte. Mithilfe der Erfahrungen, die ich bei der Arbeit an diesem Projekt sammeln konnte, messe ich sie an den eingangs genannten Kriterien.

### 6.1 Vergleich der beiden Architekturansätze

Um wieder auf die Ausgangsfrage dieser Arbeit zurückzukommen, wie man ein Videospiel effizient modelliert, stelle ich nun die beiden Architekturen, also den komponentenbasierten und den hierarchischen Ansatz, unter den in Kapitel 1.2.3 gewählten Gesichtspunkten gegenüber.

#### *Flexibilität*

Die große Schwäche des hierarchischen Ansatzes ist die Flexibilität während der Entwicklung, weil die Vererbung einen starren Rahmen vorgibt. Wir konnten in Kapitel 4 zum Beispiel feststellen,

dass es schwer ist, eine neue Entität »Gegner« in unsere bestehende Architektur einzufügen. Je größer ein Projekt ist, desto größer wird auch der »Hierarchie-Baum« des Modells. Und je höher eine Entität in der Hierarchie steht, desto mehr Entitäten unter ihr sind von ihr abhängig. Ändert man im Nachhinein also auch nur ein paar kleine Details einer Klasse, kann es gut sein, dass man auch deren Unterklassen anpassen muss.

Dem Entity-Component-System dagegen gelingt es besser, die verschiedene Teilprobleme eines Spiels in unabhängige Module zu unterteilen. Dadurch kann man einerseits die verschiedenen Module anpassen, ohne befürchten zu müssen, dass sich die Änderungen auf andere Module auswirken, andererseits kann man auch die Zusammensetzung einer Entität noch im Nachhinein beliebig zusammenstellen. Ganz nach dem Prinzip der Komposition kann ich beispielsweise auch einem Baum die *Auto-Physik-Komponente* geben, sodass er wie ein Auto durch die Gegend fährt.

Das ist also die Flexibilität während der Entwicklung bzw. während der Modellierung. Man kann darüber hinaus aber auch noch die *Flexibilität zur Laufzeit* anführen, sprich inwiefern man die Daten und Logik der Entitäten verändern kann, während das Programm ausgeführt wird. Die Attribute und Methoden einer Klasse kann man zur Laufzeit nicht verändern, weshalb sich also auch die Entitäten eines hierarchischen Ansatz nicht verändern können. Beim ECS dagegen kann man Systeme und Komponenten bei der Ausführung dynamisch hinzufügen oder entfernen. Ich kann also mitten im Spiel die »Tastatur-Steuerungs«-Komponente aus der Spieler-Entität löschen, sodass er nicht mehr auf Tastatureingaben hört oder ich kann das »Soundeffekt«-System entfernen, damit kein Ton mehr abgespielt wird.

### Wiederverwendbarkeit von Quellcode

Genauso wie auf die Flexibilität wirkt sich die Modularisierung auch positiv auf die Wiederverwendbarkeit des Quellcodes aus. Denn wenn das Spiel in möglichst unabhängige Teile aufgespalten ist, kann ich ein Modul ohne größeren Aufwand in ein anderes Projekt einbetten.

Das ist beim hierarchischen Ansatz schwerer möglich, da man eine Entität nicht so einfach aus einer Hierarchie in eine andere kopieren kann. Da die Klasse »Spieler« etwa eine Unterklasse der Klasse »Auto« ist, benötigt sie auch im Modell eines anderen Spiels die Elternklasse »Auto«. Durch die Abhängigkeiten zu den übergeordneten Klassen ist es daher mit großem Aufwand verbunden, eine Entität mit in ein anderes Projekt zu übernehmen.

Beim Entity-Component-System stehen Komponenten ihre Systeme für sich unabhängig, was ihre Wiederverwertbarkeit erheblich erleichtert. Gehen wir zum Beispiel davon aus, dass ich als nächs-

tes Projekt ein »Jump 'n' Run« entwickeln möchte, in dem ich auch die Kollision zweier Entitäten prüfe. Ohne große Probleme kann ich dann die Kollisions-Komponente und das Kollisions-System mit in das neue Projekt übernehmen.

### *Kurzer Entwicklungszeitraum*

Bezüglich des Entwicklungszeitraums lassen sich beide Ansätze weniger pauschal bewerten, weil dieser Punkt ausschlaggebend vom Projektmanagement und weniger von der Architektur abhängt.

Wenn man das Konzept des Spiels von Beginn an vollständig abgegrenzt hat und während der Entwicklung keine weiteren Änderungen vornehmen möchte, kann der hierarchische Ansatz der schnellere sein. Das setzt voraus, dass man wie etwa im Sinne des Wasserfallmodells zuerst das komplette Modell entwirft und erst danach mit der Implementation beginnt.

Ich glaube aber, dass das Wasserfallmodell in der modernen Spieleentwicklung eine Utopie ist, weil nachträgliche Änderungen bei so vielen Interessensgruppen und Parteien – Grafikdesigner, Leveldesigner, Vermarktern, Konsumenten und weitere – kaum vermeidbar sind. Und in diesen Fällen kann der komponentenbasierte Ansatz klar überzeugen, denn auch dieses Kriterium wird maßgeblich von der Modularität des ECS gefördert. Dann nämlich können verschiedene Teams parallel an unterschiedlichen Modulen arbeiten. Ein Team könnte so nur am Kollisions-System arbeiten, ein anderes kann sich gleichzeitig dem Auto-Simulations-System widmen und beide Module überschneiden sich nur an wenigen Schnittstellen.

### *Datenverarbeitungsgeschwindigkeit*

Die Datenverarbeitungsgeschwindigkeit eines Spiels wird maßgeblich von ein paar wenigen Aspekten des Programms beeinflusst. In meinem Rennspiel sind beispielsweise das Zeichnen des Levels und die Berechnung der Kollision sehr rechenaufwendige Arbeitsschritte. Entscheidend ist an diesen Stellen besonders die Wahl des Algorithmus.

Aber auch die Softwarearchitektur ist eine Einflussgröße, da es sich negativ auf die Rechenleistung auswirkt, wenn das Spiel viele Objekte verwalten muss. Hier liegt der hierarchische Ansatz im Vorteil, weil es aus weniger Objekten besteht. Das Spiel verwaltet nur eine Liste aller Entity-Objekte und muss deren update-Methode in jedem Frame aufrufen. Bei 100 Entitäten müssen also 100 Schleifendurchläufe durchgegangen werden.

Beim Entity-Component-System dagegen ruft das Spiel zuerst die update-Methoden aller Systeme

auf. Diese iterieren dann jeweils durch alle Komponente, die ihnen zugeordnet sind. Wenn wir 100 Entitäten haben, dann muss also einmal das Kollisions-System durch fast 100 Kollisions-Komponenten iterieren und einmal das Zeichen-System durch fast 100 Zeichen-Komponenten iterieren.

Mit ansteigender Anzahl von Entitäten oder Systemen wächst die Zahl der Schleifendurchläufe im Entity-Component-System also um ein Vielfaches. Auf einem modernen Computer macht es aber praktisch kaum einen Unterschied, ob man durch 100 oder 500 Objekte iterieren muss.

## 6.2 Einordnung in Entwicklungsprozesse und Designansätze

Zu Beginn der Arbeit habe ich zwei Softwareentwicklungsprozesse, das Wasserfallmodell und Prototyping, sowie die beiden fundamentalen Designansätze Bottom-Up- bzw. Top-Down-Design vorgestellt. Nun möchte ich aufzeigen, welchen Ansatz ich jeweils für mein Projekt verwendet habe.

In vielerlei Hinsicht habe ich das Spiel nach dem *Top-Down-Design* modelliert. Zuerst habe ich grundlegend festgestellt, dass sich ein *Gameloop* eignet, um die Bewegungen eines Rennspiels darzustellen. Da es für große Projekte zu unübersichtlich wäre, den kompletten Quellcode in eine Schleife zu schreiben, habe im nächsten Schritt *Entitäten* eingeführt, die nur einen kleinen Teil des kompletten Spiels darstellen. Im Sinne des Top-Down-Design habe ich also ein großes Problem in kleinere Module unterteilt. Mithilfe des Entity-Component-System konnte ich die Entitäten dann auf die Summe ihrer einzelnen Funktionalitäten reduzieren, die in eigene Komponenten und Systeme unterteilt wurden. Die »Display«-Komponente und das »Render«-System widmen sich so etwa einzlig dem Zeichnen des Spiels auf den Bildschirm. Auch alle anderen Teilprobleme des Spiels wurden in verschiedene Module gekapselt, die möglichst in sich abgeschlossen sind.

Im letzten Schritt der Modellierung habe ich dann ein Teilproblem, wie etwa die Kollisionserkennung, noch konkreter modelliert, indem ich mich dazu entschieden habe, das *Separating Axis Theorem* anzuwenden. So habe ich ein schwieriges Projekt wie ein Autorennspiel Stück für Stück auf lösbare Probleme heruntergebrochen.

Bezüglich des Entwicklungsprozess bin ich vor allem dem *Prototyping*-Schema gefolgt. Einerseits habe ich neue Modellansätze schnell in Prototypen umgesetzt, um früh feststellen zu können, ob sich diese für die Autosimulation eignen. Nachdem ich das Entity-Component-System kennengelernt habe, war es so etwa einer meiner ersten Schritte, ein rudimentäres Spiel mit dieser Architektur

umzusetzen.

Andererseits habe ich auch im weiteren Verlauf der Entwicklung zunächst lauffähige Prototypen entwickelt, in denen ein Teilespekt der Autosimulation funktioniert. Von einem fahrenden Auto, über die Kollisionserkennung, bis hin zur Benutzeroberfläche bin ich dem fertigen Ergebnis daher evolutionär näher gekommen. Zwischen jedem Prototypen habe ich das Modell reflektiert und abgewandelt.

Nicht jeder Prototyp war dabei erfolgreich. Wie schon in Kapitel 5.1 erwähnt, habe ich beispielsweise zuerst mit einem physikalisch-aufwendigerem Modell experimentiert, um mich dann schließlich doch für den unrealistischeren, aber simplen Ansatz zu entscheiden.

Ich glaube, dass mir dieser Prototyping-Ansatz besonders zu gute kam, weil ich noch wenige Erfahrungen mit der Entwicklung eines Videospiels hatte. Ich konnte neue Ideen ausprobieren, ohne befürchten zu müssen, dass ich damit gleich mein ganzes Projekt in Gefahr bringe, sollten diese nicht umsetzbar sein. Bei größeren Projekten, an denen verschiedene Teams beteiligt sind, muss das Prinzip des »Trial and Error« nicht mehr das effizienteste sein. Hier sind eher erfahrene Projektleiter und Softwaredesigner gefragt, die die Risiken verschiedener Architekturen und Modellansätze im Voraus einschätzen und abwegen können.

### 6.3 Fazit

Zu Abschluss möchte ich nun wieder auf die Ausgangsfrage dieser Arbeit zurückkehren. Welches Modell ist also am Ende das Bessere, um ein Videospiel zu entwickeln?

Bevor ich mein persönliches Fazit darlege, muss ich auch hier nochmal auf den Beginn dieser Arbeit hinweisen. Da ein Modell immer zweckgebunden ist, lässt sich diese Frage nicht universell beantworten, sondern bezieht sich in meinem Fall auf das konkrete Beispiel eines Rennspiels. Dennoch lassen sich auch aus diesem Fallbeispiel Schlüsse ziehen, die sich auf viele Spiele anwenden lassen.

Ich bin zu der Meinung gekommen, dass das Entity-Component-System die effektivere Architektur für Spiele ist. Denn das traditionellere, hierarchische Modell wird den vielen Anforderungen der modernen Spieleentwicklung nicht mehr gerecht. Einerseits schränkt das statische Prinzip der Vererbung den Softwaredesigner dahingehend ein, die zahlreichen und vielfältigen Entitäten des Spiels zu modellieren. Durch Komposition kann er einer Entität beliebige Eigenschaften und Funktionen zuweisen, ohne sich mit Abhängigkeiten zwischen Eltern- und Kinderklassen auseinandersetzen zu

müssen.

Andererseits ermutigt die Aufteilung in Systeme und Komponenten dazu, das Spiel modularer zu entwickeln. Damit möchte ich nicht sagen, dass man ein hierarchisches Modell nicht ebenso modular gestalten kann – da das ECS die Entitäten aber aus seiner Definition heraus in verschiedene Funktionalitäten unterteilt, kapselt man die Teilprobleme intuitiv ab.

Gerade als Einsteiger in die Programmierung ist das Modell der objektorientierten Programmierung leicht verständlich. Denn nach dem Prinzip »Der Baum ist eine Pflanze ist ein Lebewesen ist ein Objekt« kategorisieren viele Menschen automatisch. Je komplexer Spielewelten allerdings werden, desto schwieriger lassen sich Entitäten einordnen. Ist eine »Fleischfressende Pflanze« nun eine Pflanze oder ein Gegner?

Beim komponentenbasierten Ansatz geht man diesem Dilemma aus dem Weg, indem man die Welt nicht als unübersichtlichen Stammbaum darstellt, sondern jede Entität als die Fülle ihrer Eigenschaften und Funktionalitäten betrachtet. Die Fleischfressende Pflanze ist also ein Objekt, das Photosynthese betreibt, das Fliegen isst und Blätter besitzt. Und genau wegen dieser Sicht auf die Welt wird die komponentenbasierte Architektur von vielen Softwaredesignern für Videospiele bevorzugt.

Wenn ich abseits der konkreten Architekturansätze eines aus diesem Projekt mitnehme, dann die Erkenntnis, dass jedes Modell seine Einschränkungen besitzt, weil ich die Realität auf feste Regeln und Gemeinsamkeiten herunterbreche. Es ist dabei unausweichlich, auf Grenzfälle zu stoßen, die den Rahmen des Modells sprengen.

Abschließend bin ich sehr zufrieden damit, mich für eine Besondere Lernleistung entschieden zu haben. Als Spieler konnte ich mir anfangs nicht vorstellen, welche theoretischen und praktischen Hindernisse überwunden werden müssen, damit ein Spiel auf dem Bildschirm erscheint. Mit einem Blick »hinter den Vorhang« wird einem bewusst, welcher Aufwand hinter der Entwicklung eines Videospiels steckt. Dazu gehört auch, dass die Wahl der Softwareentwicklungsprozesse sowie der Designvorgehensweisen (Top-Down bzw. Bottom-Up) entscheidend für den Erfolg und Misserfolg eines Projekts sind. Konkret bin ich zuvor beispielsweise daran gescheitert, angefangene Spielprojekte fertigzustellen. In der Zukunft möchte ich daher früh Prototypen umsetzen, damit ich Probleme vorzeitig umgehen kann und schon mit jedem kleinen Prototypen ein Erfolgserlebnis habe.

Mit dem Wissen, welche Architekturen geeignet sind und wie ich ein Softwareprojekt angehe, kann ich meine kommenden Projekte effizienter verwirklichen.

7 Anhang

# UML-Diagramme des kompletten Spiels

Um die verschiedenen Architekturansätze vergleichen zu können, habe ich zur besseren Erklärung eine vereinfachte Autosimulation umgesetzt. Hier sind nun die kompletten UML-Diagramme des vollständigen Spiels, wie es nach dem hierarchischen Ansatz und wie es nach dem komponentenbasierten Ansatz umgesetzt würde:

*Alle Grafiken sind auf der beigelegten CD in hoher Auflösung abgespeichert.*

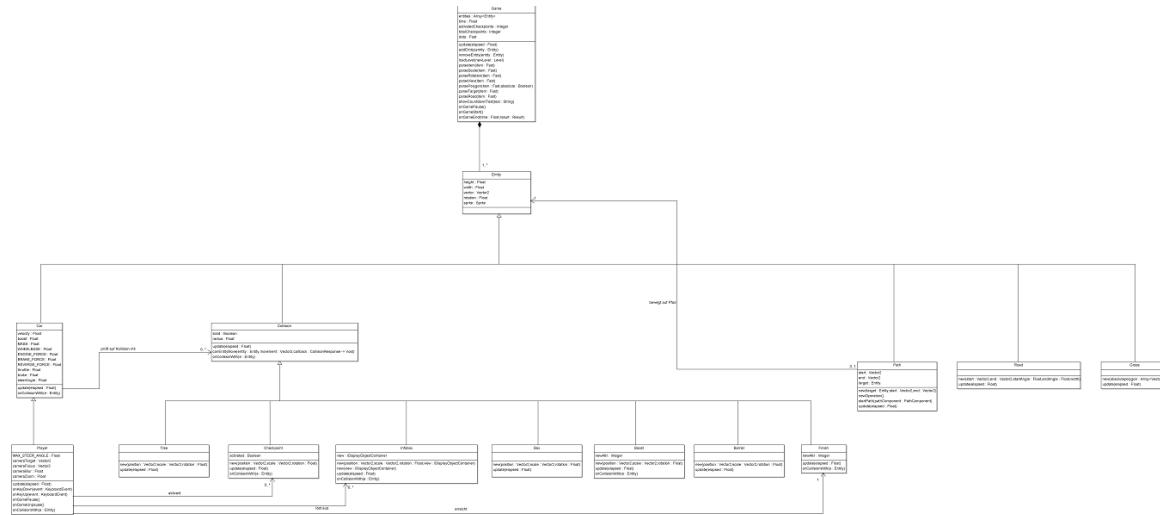
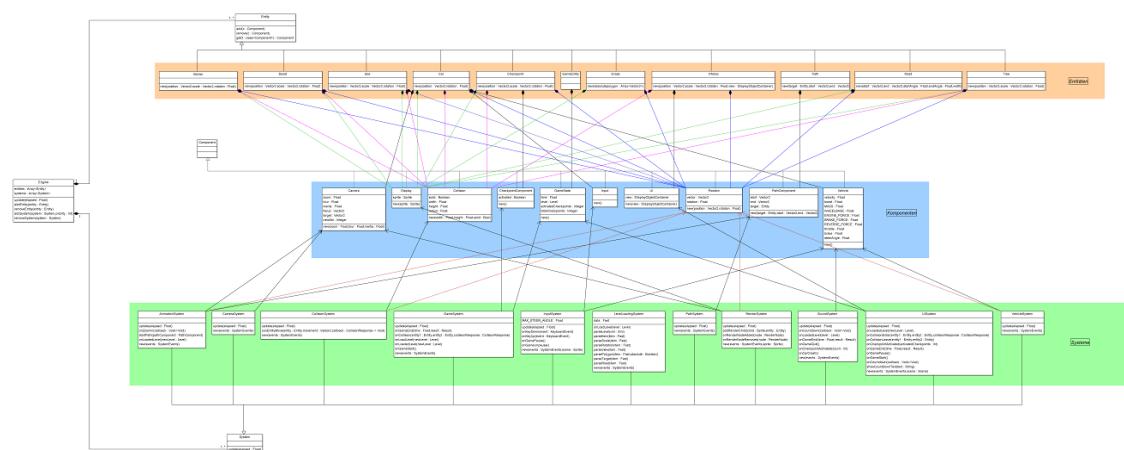


Abbildung 17: Hierarchischer Ansatz



*Abbildung 18: Komponentenbasierter Ansatz*

## Quelltext und Programm

Der Quelltext des Programms befindet sich im Ordner Quelltext/ auf der beigelegten CD oder online auf <https://github.com/jfbausch/Woehlerschule/>.

Das fertige, ausführbare Spiel ist für Windows im Ordner Spiel/ zu finden. Bei dieser Version des Programms werden aber keine Audioeffekte abgespielt. Das liegt daran, dass OpenFL die Audio-Bibliothek »OpenAL« benutzt, die aus lizenzerrechtlichen Gründen nicht in der exe-Datei mitgeliefert werden kann.

Möchte man das Spiel dennoch mit Toneffekten oder auf einem anderen Betriebssystem spielen, empfehle ich, den Quellcode selbst zu kompilieren.

## Installation von Haxe, OpenFL und weiterer Bibliotheken

Wenn man den Quellcode selbst ausführen möchte, benötigt man Haxe und weitere Bibliotheken. Zuerst installieren wir die Programmiersprache Haxe. Dazu auf folgender Website die Installationsdatei für das eigene Betriebssystem herunterladen und installieren:

<http://haxe.org/download/>

Mit Haxe wird auch gleich eine Bibliotheksverwaltung mitgeliefert. Um OpenFL, Actuate und Ash zu installieren, öffnet man die Eingabeaufforderung bzw. die Konsole und gibt folgende Befehle ein: (Die Installation kann einige Minuten dauern)

```
| haxelib install lime 2.7.0
| haxelib install openfl 3.4.0
| haxelib run openfl setup
| openfl setup windows
| haxelib install ash 1.5.4
| haxelib install haxeui 1.7.20
| haxelib install actuate 1.8.6
```

(Gegebenfalls muss die Installation der Version 2.7.0 der Lime-Bibliothek mit y bestätigt werden.)

Zuletzt führen wir das Spiel aus. Dazu wieder die Eingabeaufforderung öffnen, in den Quelltext-Ordner navigieren (indem sich die Datei `projext.xml` befindet) und dann folgenden Befehl ausführen:

ren:

```
| openfl test neko
```

## Dateien- und Ordnerstruktur des Quelltext

Der Ordner Quelltext/ setzt sich folgendermaßen zusammen:

- `assets/` Spieldaten und Ressourcen
  - `fonts/` Schriftarten
  - `levels/` Leveldateien
  - `sounds/` Soundeffekte
  - `textures/` Spieltexturen
  - `ui/` Layout-Dateien und Bilder für Benutzeroberfläche
- `src/app/` Quelltext
  - `components/` Komponenten des ECS
  - `entities/` Entitäten des ECS
  - `math/` Kleine Hilfsklassen zu mathematischen Problemen
  - `nodes/` Nodes des ECS
  - `scenes/` Verschiedene »Fenster« oder »Szenen« die angezeigt werden
  - `systems/` Systeme des ECS
    - `Configuration.hx` Klasse, die Optionen und Levelstände ausliest
    - `Game.hx` Klasse, die das ECS initiiert, wenn ein Level gestartet wird
    - `Main.hx` Hier wird das Programm gestartet
- `project.xml` Konfigurationsdatei für OpenFL

## Konstanten für Autosimulation

Der Übersichtlichkeit wegen führe ich hier alle Konstanten auf, die in der Physiksimulation relevant sind. Auf diese Werte bin ich einerseits durch Rechnen gekommen, größtenteils aber durch Ausprobieren und Testen. In einem physikalisch-realistischem Modell sind die Werte daher nicht unbedingt schlüssig.

Art	Größe	Variablenbezeichnung
Masse	1200 kg	Vehicle.MASS
Radstand	140 px	Vehicle.WHEELBASE
Motorkraft	10.000 N	Vehicle.ENGINE_FORCE
Bremskraft	20.000 N	Vehicle.BRAKE_FORCE
Rückfahrkraft	3.000 N	Vehicle.REVERSE_FORCE
Rollwiderstand	180	ROLLING_RESISTANCE
Luftwiderstand	5	AIR_RESISTANCE

## Screenshots



Abbildung 20: Hauptmenü



Abbildung 19: Level 3

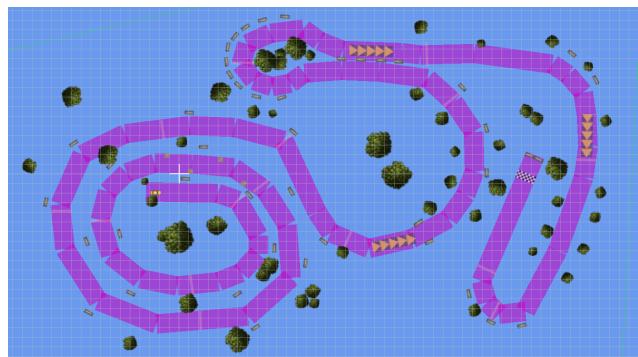


Abbildung 21: Level 8 im Leveleditor »GLEED2D«

## Abbildungen

Alle Abbildungen sind noch einmal in höherer Auflösung im Dokument `Abbildungen/Abbildungen.pdf` auf der beigelegten CD gespeichert. Sie befinden sich in dieser Datei jeweils auf den Seiten:

Abbildung 1: Wasserfallmodell.....	1
Abbildung 2: UML-Diagramm einer flachen Hierarchie.....	1
Abbildung 3: UML-Diagramm einer erweiterten Hierarchie.....	2
Abbildung 4: Quellcodedopplung.....	2
Abbildung 5: Höhere Hierarchie.....	3
Abbildung 6: Beziehungen zwischen Komponenten und Systemen.....	3
Abbildung 7: ECS in einer objektorientierten Programmiersprache.....	4
Abbildung 8: Kräfteausgleich bei höherer Geschwindigkeit.....	4
Abbildung 9: Bicycle Model.....	5
Abbildung 10: Verschiebung der Räder.....	5
Abbildung 11: Kollidieren Auto und Box?.....	5
Abbildung 12: Spielobjekte als rotierte Rechtecke darstellen.....	6
Abbildung 13: Grobe Kollisionsprüfung.....	6
Abbildung 14: Achsen finden.....	7
Abbildung 15: Projektion auf Achse y1.....	8
Abbildung 16: Tonmischung der Motorengeräusche.....	8
Abbildung 17: Hierarchischer Ansatz.....	9
Abbildung 18: Komponentenbasierter Ansatz.....	10

## Literatur

*Car Physics for Games*, Marco Monster, 2003

Auf der CD beigelegt

*Game Programming Patterns*, Robert Nystrom, 2014

<http://gameprogrammingpatterns.com/game-loop.html>

*Simple 2D car steering physics in games*, Engineering .NET, 2010

<http://engineeringdotnet.blogspot.de/2010/04/simple-2d-car-physics-in-games.html>

*Component-Based Entity Systems in Spielen*, Nick Prühs, veröffentlicht auf Heise.de, 2014

<http://www.heise.de/developer/artikel/Component-Based-Entity-Systems-in-Spielen-2262126.html>

*Tutorial Separating Axis Theorem*, wiki.delphigl.com, 2007 – 2013

[https://wiki.delphigl.com/index.php/Tutorial\\_Separating\\_Axis\\_Theorem](https://wiki.delphigl.com/index.php/Tutorial_Separating_Axis_Theorem)