



Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Design, Medien und Information
Department Medientechnik

Bachelor-Thesis zur Erlangung des akademischen Grades B.Sc.

**Konzeption und Realisierung des
Entity-Component-System am Beispiel der Unity
Game Engine und einer eigenen Anwendung in C++**

Florian Völkers
Matr.-Nr.: 2140739

Erstprüfer: Prof. Dr. Andreas Plaß
Zweitprüfer: Prof. Dr. Nils Martini

Hamburg, 14.10.2019

Inhaltsverzeichnis

1 Einleitung	4
1.1 Motivation	4
1.2 Ziel und Struktur der Arbeit	5
2 Einführung in das Entity Component System	6
2.1 Begriffserklärungen und Definitionen	6
2.1.1 Was ist eine Game Engine?	6
2.1.2 Was sind Architekturmuster?	9
2.1.3 Aggregation und Komposition	10
2.2 Gängige Architekturmuster der Videospielentwicklung	10
2.2.1 Verschiedene Ebenen der Vererbung	10
2.2.2 Entity Component Architecture	14
2.2.3 Entity Component System	17
2.3 Relevanz des ECS für die Spieleentwicklung	20
2.3.1 Vorteile	21
2.3.2 Nachteile	25
2.3.3 Einsatzbereiche	27
3 Konzeption	29
3.1 Grundlegendes Konzept aller Test-Applikationen	29
3.1.1 Ablauf der Applikationen	29
3.1.2 Komplexität der verwendeten Modelle	30
3.1.3 Ziele der Test-Applikationen	31
3.2 Konzeption der Unity Applikationen	31
3.2.1 Entity Component Architecture in Unity	31
3.2.2 Entity Component System in Unity	33
3.3 Konzeption der Test-Applikationen in C++	36
3.3.1 Verwendete Technologien und Bibliotheken	36
3.3.2 Grundlegende Funktionalität in C++	37
3.3.3 Entity Component Architecture in C++	41
3.3.4 Entity Component System in C++	44
4 Realisierung	48
4.1 Realisierung in Unity	48
4.1.1 Mit der Entity Component Architecture	48
4.1.2 Mit dem Entity Component System	50
4.2 Realisierung in C++	55
4.2.1 Mit der Entity Component Architecture	55
4.2.2 Mit dem Entity Component System	58

5 Auswertung	64
5.1 Ablauf der Tests	64
5.2 Realisierung in Unity	64
5.2.1 Evaluation der Entity Component Architecture	65
5.2.2 Evaluation des Entity Component Systems	66
5.2.3 Vergleich der Architekturmuster	66
5.3 Realisierung in C++	67
5.3.1 Evaluation der Entity Component Architecture	68
5.3.2 Evaluation des Entity Component Systems	69
5.3.3 Vergleich der Architekturmuster	70
5.4 Vergleich der Testergebnisse in Unity und C++	70
6 Resümee	72
6.1 Ausblick	72
6.2 Fazit	73
7 Glossar	74
8 Abbildungsverzeichnis	75
9 Tabellenverzeichnis	77
10 Literaturverzeichnis	78

Abstract

The aim of this bachelor thesis was to describe the architectural pattern called entity component system and find out if its usage improves the performance of video games. The tests were performed in the Unity Engine and in self-written applications in the programming language C++. In the course of this thesis other architectural patterns used in game development were described. The insights gained by these descriptions were used to successfully realize the applications and to distinguish the entity component system from other architectural patterns.

The results show that under most circumstances the entity component system improves the performance of an application. The implementation of this architectural pattern requires a lot of time and experience in software development. The usage of the entity component system in the Unity Engine was a good first step and is possible with little effort.

This thesis is aimed at experienced programmers, who would like to know more about the entity component system. It provides an opportunity to learn about its basics and to profit from the authors experiences.

Zusammenfassung

Das Ziel der Thesis war das Architekturmuster Entity Component System zu beschreiben und in verschiedenen Anwendungen, sowohl innerhalb der Unity Game Engine als auch in der Programmiersprache C++, zu prüfen, ob dessen Verwendung zu einer Verbesserung der Performance führt. Dabei wurden außerdem andere Architekturmuster, die bei der Entwicklung von Videospielen genutzt werden, beschrieben. Die bei der Beschreibung gewonnenen Erkenntnisse wurden für die erfolgreiche Realisierung der Anwendungen und für die Abgrenzung zu anderen Architekturmustern verwendet.

Das Ergebnis zeigt, dass das Entity Component System unter den meisten Umständen zu einer Verbesserung der Performance führt. Jedoch ist für die eigene Implementierung dieses Architekturmusters viel Zeit und Erfahrung im Bereich der Software-Entwicklung notwendig. Die Verwendung innerhalb der Unity Engine diente als guter Einstieg in die erforderliche Denkweise und ist mit geringem Aufwand möglich.

Die Thesis richtet sich an erfahrene Programmierer, die einen Einstieg in das Thema Entity Component System suchen. Sie bietet eine Möglichkeit, um das benötigte Grundwissen zu erlangen und von den Erfahrungen des Autors zu profitieren.

1 Einleitung

1.1 Motivation

Die Videospielindustrie ist in den letzten Jahren so stark gewachsen, dass die Konkurrenz unter den Spielentwicklern sehr groß geworden ist. Die Anzahl der veröffentlichten Videospiele pro Jahr auf Steam, der größten Verkaufsplattform für Computerspiele, ist von 537 im Jahr 2013 auf 8972 im Jahr 2018 gestiegen [steamspy, 2019]. Diese Entwicklung hängt nicht nur mit der gesteigerten Nachfrage, sondern vor allem mit den verbesserten Möglichkeiten zur Entwicklung von Videospiele zusammen. Game Engines, wie Unity oder die Unreal Engine, stehen den Entwicklern zur kommerziellen Produktion zur Verfügung. Eine Game Engine bietet den Entwicklern die Chance sich auf die Umsetzung ihrer Idee und die Produktion des Videospiele zu konzentrieren, da bereits viele grundlegende Funktionen, wie zum Beispiel Rendering, Physik oder Audio, implementiert sind. So können mehr Ressourcen für die Ausarbeitung der Idee, die visuelle Qualität und die Spielmechaniken verwendet werden.

Um sich aus der Masse der Videospiele, die auf dem Markt verfügbar sind, hervorzuheben, können Entwickler verschiedene Strategien verfolgen. Eine besondere, neuartige Idee, gut platziertes Marketing oder atemberaubende Grafiken ziehen das Interesse der Konsumenten an. Der wahrscheinlich wichtigste Aspekt ist jedoch die Performance. Einerseits wird das Spiel dank besserer Performance auch noch auf schwachen Endgeräten laufen und somit mehr Nutzer erreichen. Andererseits kann das Gameplay vielfältiger und intensiver gestaltet werden.

Eine Möglichkeit, um die Performance des Spiels zu verbessern, ist das Architekturmuster Entity Component System (ECS). Es beschreibt die Zusammensetzung der Spielwelt und folgt einem datenorientierten Design. In der Spielentwicklung wird das ECS schon seit geraumer Zeit von den meisten großen Spielentwicklern verwendet. In den öffentlich verfügbaren Engines, wie Unity oder der Unreal Engine, ist es aktuell jedoch nicht der verwendete Standard. Allerdings arbeitet Unity seit ein paar Jahren daran ECS als Standard in ihre eigene Engine zu implementieren und hat dazu weitere Tools entwickelt: das Job System und den Burst Compiler. Durch sie soll Unity ohne viel Aufwand der Entwickler ein höheres Performance-Niveau erreichen[Unity, 2019a]. Diese Tools sind in der aktuellen Version von Unity (2019.2.0f1) als Vorabveröffentlichungen vorhanden, decken jedoch noch nicht den gesamten Umfang der Engine ab.

Die Entwicklungen bei der Unity Engine, haben dazu geführt, dass ich mich näher mit dem Thema ECS beschäftigt habe. Während des Studiums habe ich in verschiedenen Projekten Spiele mit der Unity Engine entwickelt. Außerdem habe ich in einem vorherigen Projekt selbst eine Game Engine mit grundlegenden Funktionen entwickelt. Mit dem Architekturmuster ECS habe ich bisher keine eigenen Erfahrungen gesammelt. Viele erfahrene Entwickler sind der Meinung, dass datenorientiertes Design in naher Zukunft als Standard für die Videospielentwicklung gelten wird [Nikolov, 2018], weshalb auch ich mich nun damit auseinandersetzen möchte.

1.2 Ziel und Struktur der Arbeit

Diese Arbeit beschäftigt sich mit der Konzeption und Realisierung des ECS in der Unity Engine sowie der Implementierung in eigenen, in C++ geschriebenen, Anwendungen. Das Ziel der Arbeit ist es das ECS zu erklären und aufzuzeigen, dass dessen Verwendung zu einer deutlichen Verbesserung der Performance führt. Dazu werden zunächst einige Grundlagen und Begriffe erläutert, die zum Verständnis vom ECS beitragen. Anschließend wird das Architekturmuster ausführlich erklärt und dessen Relevanz für die Spieleentwicklung erörtert.

Der Kern der Arbeit liegt in der Konzeption und Realisierung von vier Applikationen, die an einem einfachen Beispiel aufzeigen, unter welchen Umständen durch das ECS eine Verbesserung der Performance erreicht werden kann. In der ersten Test-Applikation wird die Spielwelt mit der klassischen Herangehensweise an Spieleentwicklung in Unity erstellt. Dabei handelt es sich um eine objektorientierte Software-Architektur, bei der die Spielwelt aus Objekten besteht, deren Daten und Verhalten durch Komponenten bestimmt wird. Diese Architektur wird im weiteren Verlauf als Entity Component Architecture (EC) bezeichnet. In der zweiten Test-Applikation wird das von Unity bereitgestellte datenorientierte Design mit dem ECS verwendet. Analog zur Umsetzung in Unity werden zwei Applikationen in C++ entwickelt, bei denen das Hauptaugenmerk auf der Konzeption und Implementierung einer EC beziehungsweise eines ECS liegt.

Im weiteren Verlauf werden die Ergebnisse aus den vier Test-Applikationen ausgewertet. Es wird geprüft, inwiefern eine Verbesserung der Performance, sowohl in Unity als auch in den selbst entwickelten Anwendungen, erreicht werden konnte. Außerdem wird die Umsetzung der Unity Engine mit der eigenen in C++ verglichen. Zum Schluss wird ein Ausblick gewährt, welche Relevanz das ECS in Zukunft haben könnte und wie das datenorientierte Design weiterentwickelt werden kann.

2 Einführung in das Entity Component System

Dieses Kapitel befasst sich mit der Theorie hinter dem ECS. Zunächst werden einige wichtige Begriffe erklärt, die zum Verständnis des ECS beitragen. Anschließend werden die drei gängigen Architekturmuster für die Spieleentwicklung, zu denen das ECS gehört, erläutert und miteinander verglichen. Abschließend wird die Relevanz des ECS für die Spieleentwicklung aufgezeigt.

2.1 Begriffserklärungen und Definitionen

2.1.1 Was ist eine Game Engine?

Eine Game Engine ist eine Software, die Videospielentwicklern grundlegende Funktionen und Tools zur Verfügung stellt, um effizient Videospiele zu produzieren. Selbst große Entwicklerstudios, wie beispielsweise Electronic Arts, nutzen Game Engines, damit sie einen Großteil der erstellten Software wiederverwenden können. Game Engines sind meistens auf bestimmte Genres oder Plattformen spezialisiert. So gibt es Engines für die Entwicklung von 2D-Spielen, Rollenspielen oder Spielen für Smartphones. Nur wenige, wie zum Beispiel Unity, bieten dem Nutzer die Möglichkeit ein breites Spektrum an Genres und Plattformen abzudecken, denn jede Abstraktion führt dazu, dass die Engine weniger gut für ein bestimmtes Genre oder eine bestimmte Plattform optimiert ist [Gregory, 2018][12].

Funktionsweise

Eine Game Engine unterscheidet sich von einem Spiel durch eine datengesteuerte Architektur [TheChernoProject, 2018]. Während bei einem Spiel ein Großteil der Logik und Spielregeln fest kodiert ist, wird eine Game Engine so programmiert, dass sie ohne große Veränderungen als Grundlage für viele, unterschiedliche Spiele dient. Diese unterscheiden sich lediglich durch ihre Daten, die Idee oder das Aussehen.

Eine Game Engine besteht in der Regel aus einer großen Anzahl an Tools und Schichten für die verschiedenen Systeme. Oft hängen die höheren Schichten von den unteren ab. Um zum Beispiel eine Kugel zu rendern, werden eine Grafik- und eine Mathematikbibliothek benötigt.

2. Einführung in das Entity Component System

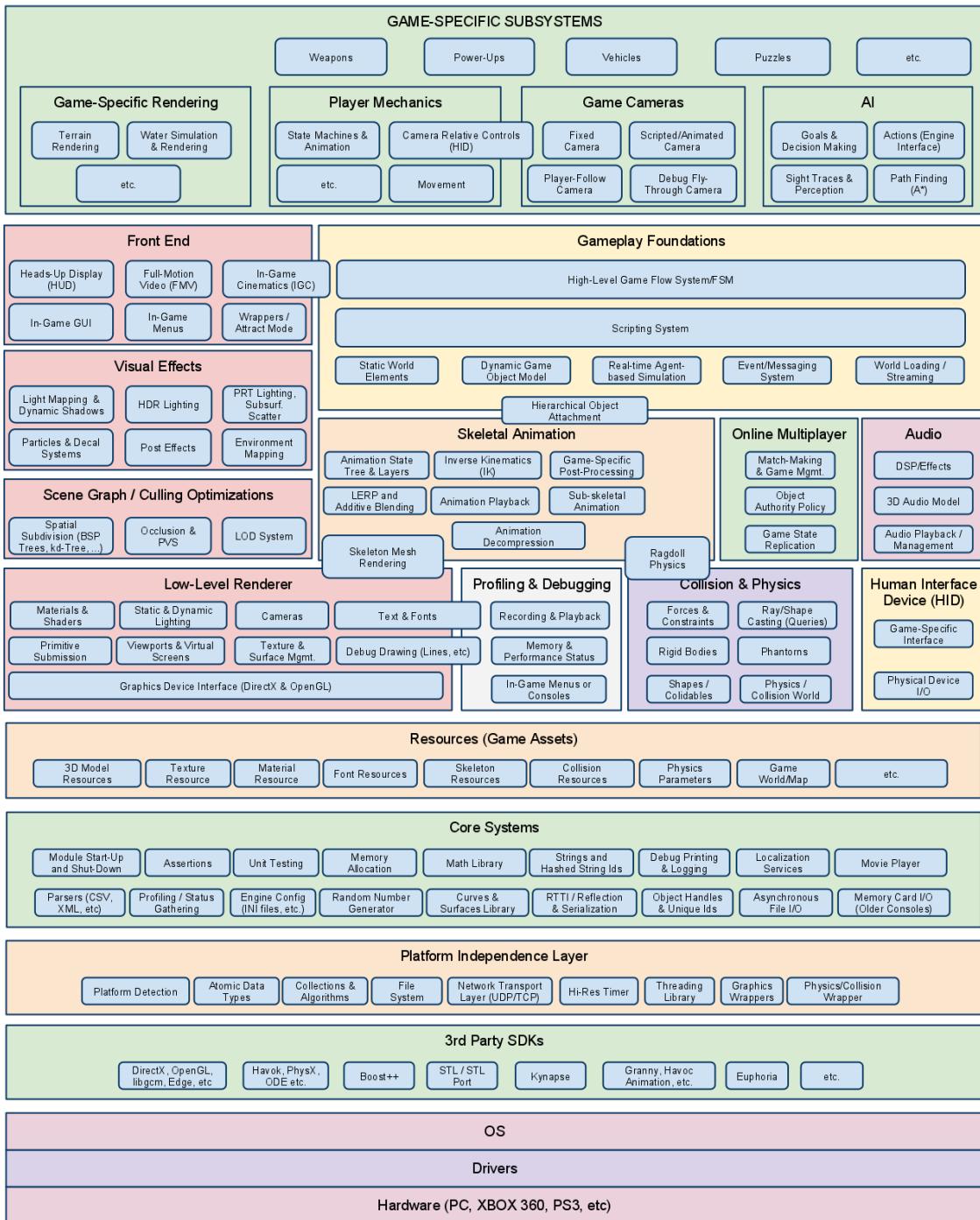


Abbildung 2.1: Überblick über die Systeme und Schichten einer Game Engine [Gregory, 2018, 39]

Geschichte

Der Begriff Game Engine wurde zum ersten Mal in der Mitte der 90er Jahre verwendet. Bis dahin wurden Videospiele und die grundlegenden Systeme, die benötigt werden, wie das Rendering, Physik oder Audio, immer wieder für jedes Spiel neu entwickelt.

Das 1993 veröffentlichte Spiel *Doom* von *id Software* war eines der ersten Spiele, bei dem die Programmierung des Gameplays deutlich von der Programmierung der grundlegenden Systeme getrennt wurde [Gregory, 2018][11]. Dadurch konnten große Teile der Entwicklung für weitere Projekte verwendet werden, ohne stark verändert werden zu müssen.

Das 1998 erschienene Videospiel *Unreal* von *Epic MegaGames* führte diesen Gedanken weiter und wurde mit der Absicht konzipiert nicht nur für weitere eigene Projekte wiederverwendet zu werden, sondern auch lizenziert zu werden [Busby et al., 2009]. So nutzten andere Entwickler die Unreal Engine, um ihre Ideen für Videospiele umzusetzen. Heutzutage gibt es zahlreiche lizenzierte Game Engines, die für die Entwicklung von Videospiele entwickelt werden.

Unity

Unity ist eine sehr umfangreiche Game Engine, mit der Applikationen für verschiedene Plattformen und Genres entwickelt werden können. Sie erschien im Jahr 2005, ist in C++ programmiert und nutzt als Skriptsprache C#. Unity ist darauf ausgelegt schnell und einfach Videospiele entwickeln zu können. Es bietet zahlreiche nützliche Tools für die Entwicklung, einen übersichtlichen Editor, in dem man Spielwelten erstellen, bearbeiten und testen kann, und eine ausführliche Dokumentation. Durch diese Tools und eine große Anzahl an Tutorials gelingt der Einstieg in Unity sehr leicht. Durch die Flexibilität, hinsichtlich des Genres und der Zielplattformen, ist Unity jedoch auch für fortgeschrittene Nutzer und für die kommerzielle Produktion von Videospiele sehr geeignet.

Um die Engine zukünftig noch konkurrenzfähiger zu machen, wurde 2018 der Data-Oriented Technology Stack (DOTS) von Unity veröffentlicht. Dieser kann in der aktuellen Version von Unity (2019.2.0f1) als Vorabversion genutzt werden und bietet verschiedene Tools zur Verbesserung der Performance. Das ECS zählt zu diesen Tools und soll zukünftig als Standard-Architekturmuster für Applikationen in Unity verwendet werden.

2.1.2 Was sind Architekturmuster?

Das ECS ist ein Architekturmuster aus dem Bereich der Videospielentwicklung. Daher ist es wichtig zunächst zu definieren was Architekturmuster sind, wie sie eingesetzt werden und was sie von Entwurfsmustern und Idiomen unterscheidet.

„An architectural pattern is a named collection of architectural design decisions that are applicable to a recurring design problem, parameterized to account for different software development contexts in which that problem appears.“ [Taylor et al., 2007]

Ein Architekturmuster wird also gezielt zur Lösung bestimmter Probleme in Software-Architekturen eingesetzt und erhöht dadurch direkt die Qualität dieser Architektur. Architekturmuster werden entsprechend ihrer Aufgabe in verschiedene Kategorien eingeteilt [ITWissen.info, 2018].

- **Strukturierungsmuster** sind für die Unterteilung und Zuordnung von Komponenten und Subsystemen im Bezug auf das Gesamtsystem zuständig. Ein Beispiel für die Anwendung eines Strukturierungsmusters ist das Schichten-System einer Game Engine.
- **Muster für verteilte Systeme** organisieren das Verteilen von Ressourcen und Diensten innerhalb eines Systems. Als Beispiel kann man das Client-Server-Modell nennen, das die Verteilung von Aufgaben innerhalb eines Netzwerks beschreibt.
- **Adaptive Systeme** dienen zur Modifizierung, Erweiterung und Anpassung von Software-Architekturen. Ein Beispiel für dieses Muster ist Reflexion, was bedeutet, dass ein Programm seine eigene Struktur kennen und modifizieren kann.
- **Interaktive Systeme** dienen zur Strukturierung von Interaktionen zwischen Mensch und Computer. Ein bekanntes Muster für die Lösung eines interaktiven Systems ist das Muster Model View Controller.

Architekturmuster sind dabei von Idiomen und Entwurfsmustern zu unterscheiden. Sie werden in die oberste Abstraktionsebene eingeteilt und beschreiben globale Systemeigenschaften und Lösungen zur allgemeinen Software-Architektur. Entwurfsmuster und Idiome bieten Lösungen für spezifische Probleme, die keine Auswirkung auf das Gesamtsystem haben. Idiome sind dabei spezifisch für eine bestimmte Programmiersprache.

2.1.3 Aggregation und Komposition

Aggregation und Komposition sind in der Informatik zwei Arten von Assoziationen. Assoziationen beschreiben eine Beziehung zwischen zwei unterschiedlichen Klassen, die miteinander auf eine bestimmte Art und Weise kommunizieren müssen [Paradigm, 2019]. Diese Assoziation wird als Aggregation bezeichnet, wenn eine Klasse eine andere benutzt, und als Komposition, wenn eine Klasse eine andere besitzt [Gregory, 2018, 111]. Außerdem ist bei der Komposition im Gegensatz zur Aggregation eine Komponente stark an die Lebensdauer des besitzenden Objektes gebunden. Das heißt, dass eine Komponente ohne das Objekt nicht existieren kann. Bei der Aggregation kann die selbe Komponente sogar Teil einer weiteren Klasse sein. Im Zusammenhang mit dem ECS werden beide Begriffe verwendet. In dieser Arbeit wird der Begriff der Komposition verwendet.

2.2 Gängige Architekturmuster der Videospielentwicklung

Da der Aufbau von Videospielen meist der realen Welt nachempfunden wird, befinden sich in der Spielwelt alle für das Spiel und seine Logik relevanten Objekte. Wie diese angeordnet sind, wie sie aktualisiert werden und wie sie erstellt sowie zerstört werden ist also von sehr großer Bedeutung. In diesem Abschnitt werden drei unterschiedliche Architekturmuster vorgestellt, die in der Videospielentwicklung verwendet werden.

2.2.1 Verschiedene Ebenen der Vererbung

Das erste Architekturmuster, das vorgestellt wird, besteht aus verschiedenen Ebenen der Vererbung.

„Vererbung ist eines der grundlegenden Prinzipien der objektorientierten Programmierung. Dort können von bestehenden Klassen ausgehend neue Klassen erstellt werden, die zunächst die gleichen Eigenschaften und Methoden besitzen wie die Ausgangsklasse. Die neue Klasse wird als abgeleitete Klasse oder UnterkLASSE bezeichnet, die Ausgangsklasse als Super- bzw. Oberklasse. Die abgeleitete Klasse kann die von ihrer Superklasse geerbten Eigenschaften und Methoden überschreiben oder durch zusätzliche ergänzen.“ [ITWissen.info, 2009]

Zur Architektur der Spielwelt wird dabei meistens zu Beginn eine Klasse *Game Object* erstellt, die als Basisklasse für alle Objekte in der Spielwelt dient. Je nach Engine wird diese Klasse auch *Entity* oder *Actor* genannt. Von ihr ausgehend werden den Objekten im Spiel weitere Funktionen und Eigenschaften hinzugefügt. Am besten lässt sich dies an einem Beispiel zeigen. Die Abbildung 2.2 zeigt eine Vererbungshierarchie, die für ein Videospiel verwendet werden kann.

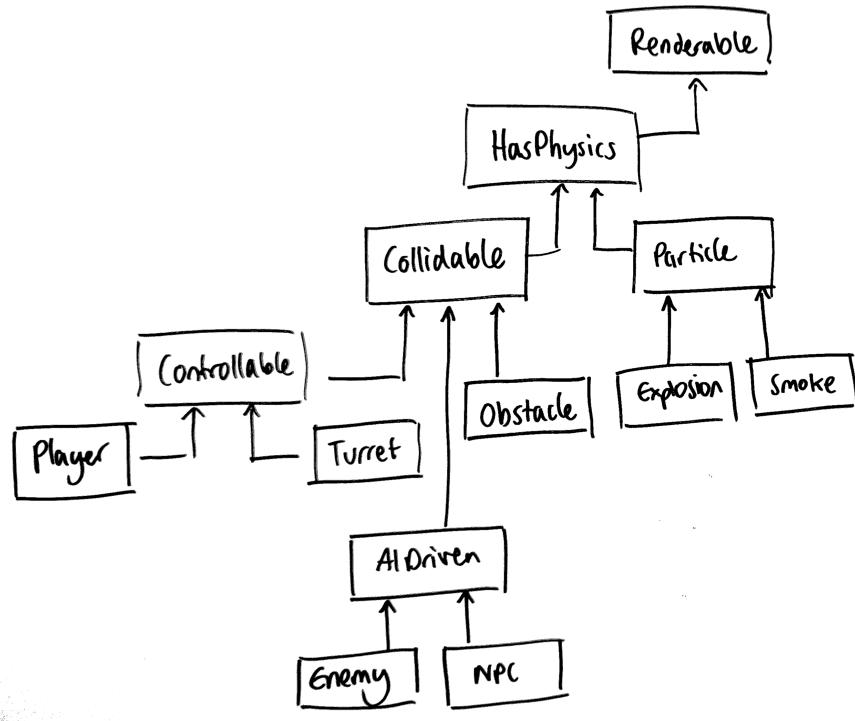


Abbildung 2.2: Beispiel einer Vererbungshierarchie in einem Videospiel [Jordan, 2018]

Ein Problem bei der Verwendung von Vererbung

In diesem Beispiel erben alle Objekte von der Klasse *Renderable*, die Objekten Funktionen und Eigenschaften gibt, die zur visuellen Darstellung auf dem Bildschirm nötig sind. Das Beispiel zeigt sehr gut, warum Vererbung aufgrund seiner Inflexibilität ungeeignet für die Architektur von Spielwelten ist. Möchte man, von dieser Vererbungshierarchie ausgehend, zum Beispiel einen Geist als Gegner zum Spiel hinzufügen, kommt es zu folgendem Problem: In der Regel kann man mit einem Geist nicht kollidieren, da dieser körperlos ist. Die Klasse *Enemy* erbt jedoch von der Klasse *Collidable*, wodurch der Geist Funktionen und Eigenschaften erhält, die zur Erzeugung und Abfrage von Kollisionen dienen.

Lösung durch Code-Duplikation

Zur Lösung könnte man eine weitere Klasse *EnemyWithNoCollidable* erstellen, die nicht von *Collidable*, sondern von *Renderable* erbt. Dies führt jedoch dazu, dass man nicht nur den Code aus der Klasse *Enemy* duplizieren muss, sondern auch den Code aus der Klasse *AIDriven*. Zukünftige Änderungen an diesen Klassen müssten also an mindestens zwei Stellen vorgenommen werden. Probleme sind dadurch vorprogrammiert.

Lösung durch Code-Verschiebung in Richtung der Basisklasse

Eine weitere Möglichkeit zur Lösung des Problems wäre es einen Großteil der Funktionen und Eigenschaften in die Basisklasse zu verschieben. In der Unreal Engine 3 wurde dies gemacht. Die Klasse *Actor* beinhaltete alle möglichen Funktionen zum Rendering, Physik oder zum Abspielen von Sounds, wodurch circa 90 Prozent aller Objekte im Spiel von dieser Klasse abgeleitet waren [Prühs, 2014]. Jedoch bringt diese Architektur schwerwiegende Probleme mit sich. Einerseits werden nicht alle Objekte im Spiel alle Funktionen und Eigenschaften, die in der Klasse *Actor* vorhanden sind, nutzen. Dadurch wird unnötig Platz im Speicher reserviert. Andererseits sind diese Art von Multifunktionsklassen sehr schwer zu pflegen, da sie sehr umfangreich und unübersichtlich sind. Außerdem sind dadurch viele Systeme aneinander gekoppelt.

Lösung durch Mehrfachvererbung

Eine dritte Möglichkeit zur Lösung des Problems wäre es die Vererbungshierarchie flacher zu gestalten und Mehrfachvererbung zu erlauben. Bei der Mehrfachvererbung erbt eine Klasse die Eigenschaften und Methoden von mehreren Klassen. Man könnte die Klassen *AIDriven* und *Collidable* auf die gleiche Ebene stellen wie die Klasse *Renderable*. Die Klasse *Ghost* könnte dann sowohl von *Enemy* als auch von *Renderable* erben.

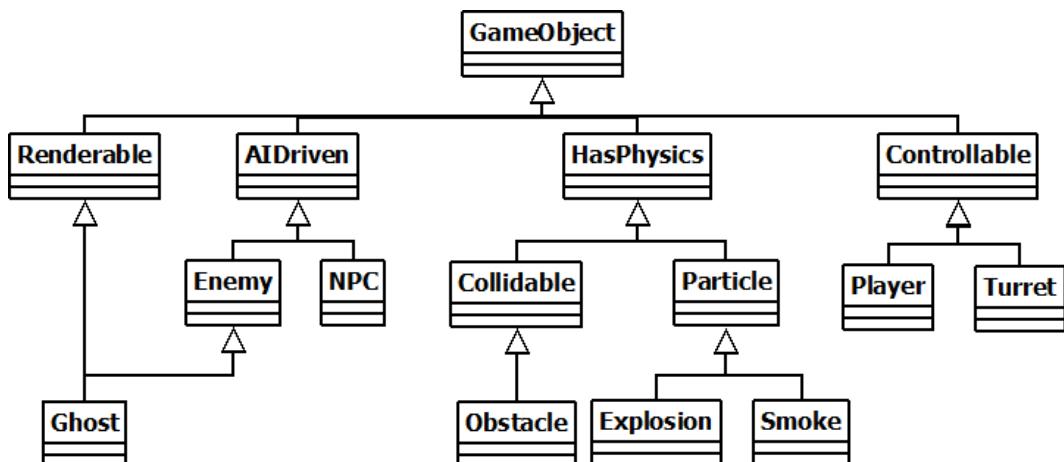


Abbildung 2.3: Veränderte Vererbungshierarchie mit Mehrfachvererbung

Doch was soll passieren, wenn das Spiel auch freundliche Geister als NPCs beinhalten soll? Ein weiterer Sonderfall tritt ein. Die Klasse *FriendlyGhost* muss erstellt werden, die von den Klassen *Renderable* und *NPC* erbt. Dies bleibt aber auch der einzige Unterschied zwischen *Ghost* und *FriendlyGhost*. Mit jedem Sonderfall wird Code dupliziert und die Vererbungshierarchie muss angepasst werden.

Deadly Diamond of Death

Außerdem ist Mehrfachvererbung nicht in allen Programmiersprachen möglich und wird aufgrund der mit sich bringenden Probleme von vielen Programmierern vermieden. Das wohl bekannteste Problem bei der Mehrfachvererbung ist der sogenannte *Deadly Diamond of Death*. Dieses tritt in folgendem Fall auf: Zwei Klassen, B und C, erben von einer Klasse A und überschreiben eine Funktion F der Klasse A. Erbt nun eine weitere Klasse D sowohl von Klasse B als auch von Klasse C und überschreibt nicht die Funktion F, so ist nicht eindeutig geklärt, welche Version von der Funktion F die Klasse D implementiert. Die von Klasse B oder die von Klasse C [Martin, 1997]? Ähnlich verhält es sich mit der Variable i, die in der Klasse A vorhanden ist. Soll in Klasse D nur eine Kopie der Variable i aus der Klasse A vorhanden sein oder soll es zwei Kopien der Variable i aus den Klassen B und C geben?

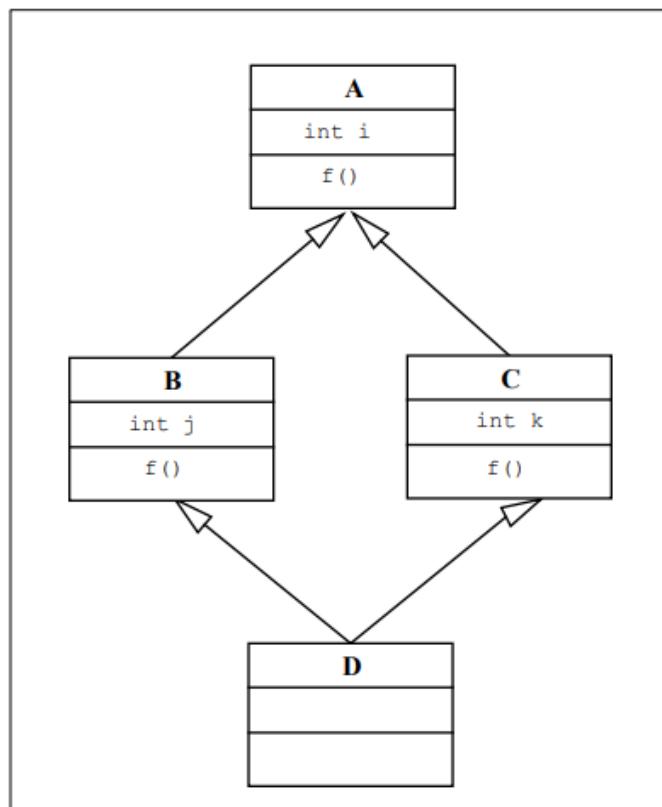


Abbildung 2.4: Deadly Diamond of Death [Martin, 1997]

Diese Fragen können nicht eindeutig geklärt werden und müssen für jeden Fall spezifisch gelöst werden. Dies führt dazu, dass die Verwaltung der Klassen viele Ressourcen in Anspruch nimmt, die für andere Aufgaben genutzt werden könnten. Daher wird Mehrfachvererbung im Allgemeinen vermieden.

2.2.2 Entity Component Architecture

„Favor object composition over class inheritance.“ [Gamma et al., 1994, 31]

Als Alternative zur Vererbung kann zur Architektur der Spielwelt Komposition verwendet werden. Robert Nystrom, der als Spiele-Entwickler unter anderem bei Electronic Arts gearbeitet hat, definiert komponentenbasiertes Design so:

„A single entity spans multiple domains. To keep the domains isolated, the code for each is placed in its own component class. The entity is reduced to a simple container of components.“ [Nystrom, 2014, 270]

Anstelle einer komplexen und tiefen Vererbungshierarchie, die die Eigenschaften und das Verhalten eines Objektes bestimmt, besteht jede Entität in der Spielwelt aus einer beliebigen Anzahl an Komponenten. Jede einzelne Komponente implementiert die benötigten Variablen und Funktionen, um ein bestimmtes Verhalten zu gewährleisten. Eine *TransformComponent* besteht zum Beispiel aus den Werten für Position, Rotation und Skalierung und implementiert Funktionen wie *Translate*, *Rotate* und viele mehr. Wenn eine Entität eine *HealthComponent* besitzt, kann sie sterben, wenn sie eine *AnimationComponent* besitzt, wird sie animiert. Dieses Muster wird von einigen Entwicklern als ECS bezeichnet, eine passendere Bezeichnung ist jedoch EC.

Vorteile der Entity Component Architecture

So kann, ohne am bestehendem Code etwas zu ändern, eine Vielzahl an Entitäten mit unterschiedlichen Eigenschaften und Verhalten generiert werden. Dazu muss einer Entität lediglich die passende Komponente hinzugefügt werden. Wenn dem Spiel eine neue Komponente hinzugefügt wird, ändert sich nichts an den Entitäten oder bestehenden Komponenten. Analog zum Beispiel aus dem Abschnitt zur Vererbungshierarchie könnte man die Entitäten *Player*, *Enemy*, *NPC* und *Invisible Wall* wie in Abbildung 2.5 dargestellt zusammensetzen.



Abbildung 2.5: Erstellung von Entitäten durch Komposition [Jordan, 2018]

2. Einführung in das Entity Component System

Die Verwendung von Komposition gibt dem Entwickler also eine große Flexibilität bei der Erstellung von Entitäten, ohne auf Redundanzen im Code oder Mehrfachvererbung angewiesen zu sein. Außerdem wird durch die Kapselung in die verschiedenen Komponenten jede einzelne Klasse kurz und einfach gehalten und bietet eine vollständige Entkopplung von anderen Komponenten. Dadurch entsteht der Vorteil, dass der Code durch die Programmierer leicht zu pflegen und zu erweitern ist.

Es wird natürlich aber nicht vollständig auf das Prinzip der Vererbung verzichtet. Es wird lediglich versucht tiefe und komplexe Vererbungshierarchien zu vermeiden und stattdessen flache, breite Hierarchien zu verwenden. So ist es üblich, dass jede Komponente von einer Basisklasse abgeleitet wird, die grundlegende Funktionen und Eigenschaften von Komponenten festlegt.

Ein weiterer Vorteil einer Lösung der Architektur der Spielwelt durch Komposition wird bei der Entwicklung des Videospiels deutlich. Sobald eine Komponente erfolgreich implementiert ist, kann sie von den Designern zu allen möglichen Objekten hinzugefügt, bearbeitet und auch wieder entfernt werden. Dadurch, dass die Komponenten erst während der Laufzeit hinzugefügt und entfernt werden, muss der Code nicht neu kompiliert werden, um Änderungen am Spiel vorzunehmen [Gregory, 2018, 159]. So kann effizient mit guter Aufgabenaufteilung an einem Projekt gearbeitet werden. Die einzelnen Komponenten können außerdem sehr einfach serialisiert werden. Das heißt, dass der Zustand eines Objekts als Datei, zum Beispiel im Format *XML* oder *JSON*, gespeichert wird. Diese Daten können unter anderem für die Rekonstruktion des Spielzustands verwendet werden. Während der Entwicklung können sie mit verschiedenen Tools editiert und unter Versionsverwaltung gestellt werden.

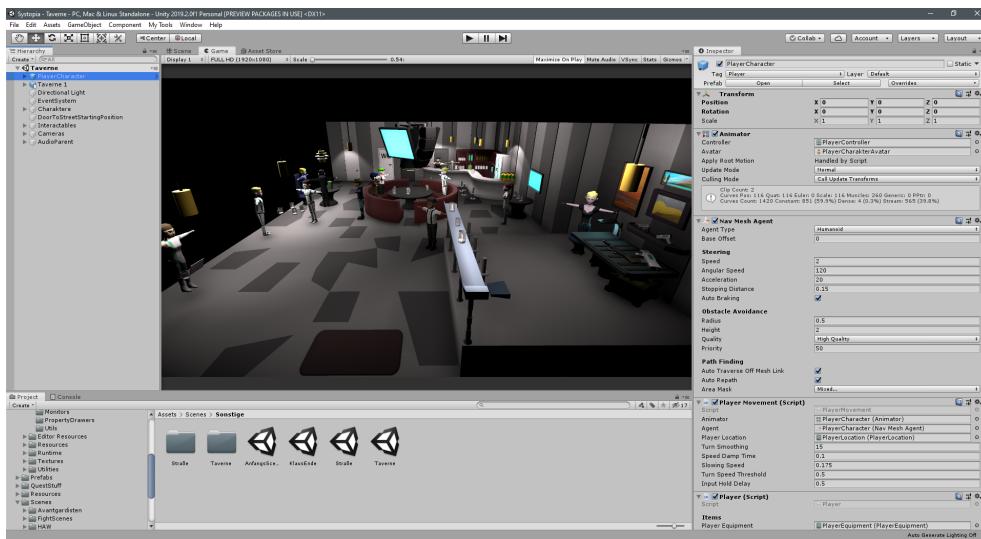


Abbildung 2.6: In der Unity Engine kann man über den *Inspector* (rechts) Komponenten von *GameObjects* editieren, entfernen und erstellen

Relevanz in der Unity Engine

In der *Unity* Engine wird die EC momentan als Standard verwendet. Die Objekte der Spielwelt sind *GameObjects*¹, die sich aus verschiedenen *Components*² zusammensetzen und vom Entwickler durch Skripte, die von der Klasse *MonoBehaviour*³ erben, weitere Funktionalität und Eigenschaften erhalten können.

Kommunikation zwischen den Komponenten

Obwohl eine komplette Entkopplung der Komponenten wünschenswert wäre, müssen die Komponenten in vielen Fällen miteinander kommunizieren. Eine *PhysicsComponent* muss zum Beispiel einer *TransformComponent* mitteilen, dass die Position verändert wird. Es gibt verschiedene Möglichkeiten, mit denen man dieses Problem lösen kann [Nystrom, 2014, 282].

- **Zugang über die Klasse GameObject:** Bei dieser Möglichkeit verlagert man Eigenschaften, die viele verschiedene Komponenten betreffen, in die Klasse *GameObject* und macht sie über Variablen oder Funktionen direkt verfügbar. So könnten zum Beispiel die Position, Rotation und Skalierung Teil der Klasse *GameObject* werden. Dadurch bleiben die Komponenten entkoppelt. Jedoch werden Eigenschaften und Verhalten des Objektes aus Komponenten in die Klasse *GameObject* verschoben, wodurch diese umfangreicher und schwieriger zu pflegen wird.
- **Direkte Kommunikation zwischen Komponenten:** Bei dieser Methode erhalten Komponenten, wenn nötig, direkte Referenzen zu anderen Komponenten. Zum Beispiel könnte die *AttackComponent* eines Gegners eine Referenz zu der *AudioComponent* erhalten, um einen Sound abzuspielen, wenn der Gegner den Spieler angreift. Diese Art von Kommunikation ist sehr einfach und schnell zu implementieren, jedoch koppelt es die Komponenten sehr eng miteinander.
- **Kommunikation über Events:** Bei der Kommunikation über Events, oft auch *Messaging* genannt, werden bei bestimmtem Verhalten Nachrichten an alle anderen Komponenten gesendet. Zum Beispiel löst die *CollisionComponent* eines Fahrzeugs in einem Rennspiel ein Event *Collided* aus. Wer oder was auf dieses Event reagiert, ist für die *CollisionComponent* irrelevant. Jede Komponente, die sich für dieses Event interessiert, registriert sich und reagiert, sobald das Event ausgelöst wird, auf ihre Art und Weise. So könnte auf das Event *Collided* zum Beispiel eine *AudioComponent* reagieren, indem sie einen Sound abspielt, und eine *GraphicsComponent*, indem sie das Modell des Fahrzeugs

¹Mehr Infos unter: <https://docs.unity3d.com/ScriptReference/GameObject.html>

²Mehr Infos unter: <https://docs.unity3d.com/Manual/Components.html>

³Mehr Infos unter: <https://docs.unity3d.com/ScriptReference/MonoBehaviour.html>

durch ein beschädigtes Modell austauscht. Der Vorteil dieser Art von Kommunikation ist die absolute Entkopplung aller Systeme. Außerdem können leicht neue Events hinzugefügt oder nicht mehr benötigte Events entfernt werden. Es entstehen auch keine Fehler, wenn es keine Reaktion auf ein Event gibt oder das Event nicht ausgelöst wird. Diese Methode ist jedoch auch die komplexeste Art und Weise der Kommunikation und kann bei falscher Implementierung mehr Probleme kreieren als lösen.

2.2.3 Entity Component System

Das Architekturmuster ECS ist eine Weiterentwicklung der EC. Es nutzt dabei die gleichen Vorteile, die durch die Verwendung von Komposition entstehen, aus und bringt neue Vorteile durch das datenorientierte Design mit sich. Es existieren viele, unterschiedliche Definitionen des ECS, jedoch werden die namensgebenden Begriffe in den meisten Fällen folgendermaßen definiert [Martin, 2007]:

- **Entity:** Die Entitäten stellen die Objekte in der Spielwelt dar. Sie verknüpfen die Komponenten mit den Systemen. Im Gegensatz zur im vorherigen Abschnitt vorgestellten EC bestehen die Entitäten lediglich aus einer einzigartigen *ID*. In vielen Implementierungen wird sogar auf eine Klasse *Entity* verzichtet. Das Erstellen und Löschen von Entitäten sowie die Zuordnung der gewünschten Komponenten wird oft von einem *EntityManager* übernommen.
- **Component:** Die Komponenten sind reine Datencontainer ohne jegliche Funktionalität. Sie werden möglichst klein gehalten und immer einem bestimmten Zweck zugeordnet. Es gibt außerdem Komponenten, die keinerlei Daten beinhalten und lediglich als Tags dienen. Tags werden verwendet, um eine Entität einer bestimmten Gruppe zuzuordnen, zum Beispiel zu der Gruppe von Gegnern.
- **System:** Systeme sind für das Verhalten von Entitäten zuständig. Jedes System wendet dieses Verhalten auf alle Entitäten an, die aus bestimmten Komponenten zusammengesetzt sind. Zum Beispiel sucht ein *MoveSystem* nach allen Entitäten, die die Komponenten *Position* und *MovementSpeed* haben, und verändert die Position der Entität entsprechend der angegebenen Bewegungsgeschwindigkeit. Alle Entitäten, die keine *Position* oder *MovementSpeed* haben, werden von diesem System nicht verändert.

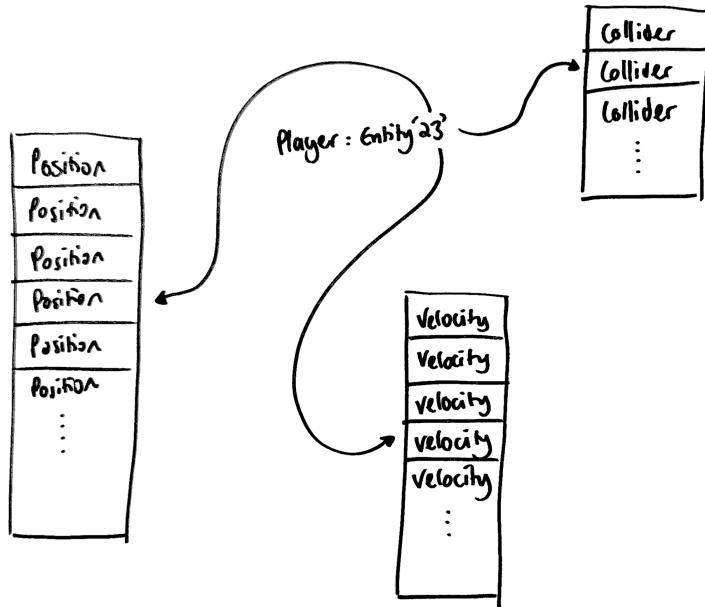


Abbildung 2.7: Skizze zum Zusammenhang von Entitäten und Komponenten [Jordan, 2018]

Da die einzige Aufgabe von Entitäten im ECS in der Verknüpfung von Komponenten besteht, die dann bestimmten Systemen zur Verfügung gestellt werden können, reicht es aus Entitäten als einzigartige *IDs* zu sehen. Anstatt also, dass die Entitäten wissen, welche Komponenten zu ihnen gehören, besitzen die Komponenten über die *EntityID* die Information, zu welcher Entität sie gehören [Nystrom, 2014, 352]. Während man bei der EC noch ein Design mit *Arrays of Structures* verwendet hat, nutzt man nun ein Design mit *Structures of Arrays*⁴.

Datenorientiertes Design

Das ECS folgt außerdem den Prinzipien des datenorientierten Designs. Dieses wird zur Optimierung von Applikationen, insbesondere in der Videospielentwicklung, angewandt. Einer der größten Befürworter von datenorientiertem Design, Mike Acton, arbeitet seit 2017 bei Unity, um die Implementierung von datenorientiertem Design in der Unity Engine voranzutreiben. In seinem Talk auf der CppCon 2014 zum Thema *Data-Oriented Design and C++* beschreibt er die Prinzipien von datenorientiertem Design.

„The purpose of all programs, and all parts of those programs, is to transform data from one form to another. If you don't understand the data you don't understand the problem. Conversely, understand the problem

⁴Mehr Infos unter: https://en.wikipedia.org/wiki/AoS_and_SoA

by understanding the data. [...] Everything is a data problem. Including usability, maintenance, debug-ability, etc. Everything. Solving problems you probably don't have creates more problems you definitely do.“ [Acton, 2014]

Nach Acton liegt die Aufgabe aller Programme in der Transformation von Daten von einer Form in eine andere. Daher ist er der Meinung, dass man Probleme am besten versteht, wenn man die zugrundeliegenden Daten und deren Probleme versteht. Als Grund für ein erforderliches Umdenken hin zu datenorientiertem Design sieht er drei *Lügen*, die durch objektorientiertes Denken in C++ erzeugt worden sind [Acton, 2014]:

- **Software is a platform**
- **Code should be designed around a model of the world**
- **Code is more important than data**

Diese *Lügen* führen seiner Meinung nach zu schlechter Performance, geringer Stabilität, weniger Möglichkeiten für Optimierungen, schlechter Testbarkeit und verhindern die Nutzung von Nebenläufigkeit.

Kommunikation zwischen den Komponenten

Das ECS bietet die Möglichkeit große Teile der nötigen Kommunikation zwischen Komponenten zu vereinfachen. Da in der EC Komponenten noch selbst das Verhalten von Entitäten implementiert haben, musste oft entschieden werden, welche Komponente für welches Verhalten zuständig war. Man möchte zum Beispiel zwei verschiedene Arten von Fischen zum Spiel hinzufügen. Fisch A ist klein und wehrt seine Fressfeinde ab, indem er ein tödliches Sekret absondert. Fisch B ist groß und intelligent. Er weiß, dass er nicht in die Nähe von Fisch A kommen darf. Als Programmierer muss man sich entscheiden, wie man dieses Verhalten der beiden Fische umsetzt. Zum einen könnte man es als Methode *AvoidPoisonousFish* von Fisch B implementieren, zum anderen als Methode *AvoidMe* von Fisch A. Egal für welche Implementierung man sich entscheidet, es muss eine direkte Kommunikation zwischen den beiden stattfinden. Im ECS wird dieses Problem durch die Erstellung eines *AvoidanceSystem* eindeutig gelöst. Fisch A erhält die Komponente *AvoidThis* und Fisch B erhält die Komponente *Avoid*. Das *AvoidanceSystem* regelt die Vermeidung der beiden. Es sucht nach allen Entitäten, die die Komponenten *Position* sowie *AvoidThis* besitzen und auch nach den Entitäten, die die Komponenten *Position*, *Heading* sowie *Avoid* besitzen. Sollte eine Entität mit der Komponente *Avoid* einer Entität mit der Komponente *AvoidThis* zu Nahe kommen, wird das *Heading* der Entität so verändert, dass es die Entität mit der Komponente *AvoidThis* vermeidet.

Ein Großteil der Kommunikation zwischen den Komponenten wird von den Systemen übernommen. Alles was an Kommunikation außerdem benötigt wird, muss über Events stattfinden, da die Möglichkeit zur Auslagerung von Daten in die Klasse *Entity* nicht mehr vorhanden ist.

Nähere Betrachtung von Systemen

Systeme sind im ECS für das Verhalten innerhalb des Spiels zuständig. Ein System wirkt dabei auf mindestens eine Art von Komponenten. Dafür müssen Systeme nicht unbedingt die Daten einer Komponente verändern. Sie können die Daten von Komponenten lesen, verändern oder lesen und verändern.

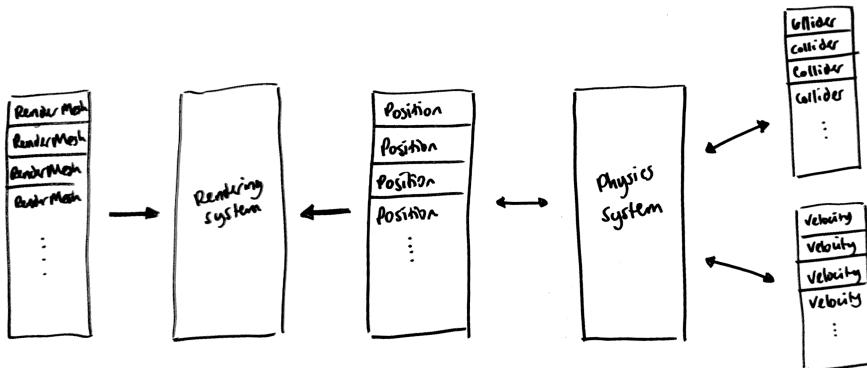


Abbildung 2.8: Skizze zum Zusammenhang von Komponenten und Systemen [Jordan, 2018]

Ein *RenderingSystem* liest zum Beispiel die Daten vom *RenderMesh* sowie der *Position*, um ein bestimmtes Modell an einer bestimmten Stelle auf dem Bildschirm darzustellen. Es nimmt keine Veränderungen an den Komponenten vor. Ein *PhysicsSystem* hingegen liest zum Beispiel die Daten aus den Komponenten *Position*, *Velocity* und *Collider* ein, prüft, ob es zu einer Kollision mit einem anderen Objekt kommt, und verändert dementsprechend die *Position* entweder aufgrund der Kollision oder der Geschwindigkeit der Entität.

2.3 Relevanz des ECS für die Spieleentwicklung

Um die Relevanz von ECS in der Spieleentwicklung einzurichten, werden zunächst die aus der Verwendung entstehenden Vor- und Nachteile genannt. Anschließend werden sowohl die bisherigen als auch die zukünftigen Einsatzbereiche des ECS beschrieben sowie die Spielgenres erläutert, die von dem ECS am meisten profitieren dürften.

2.3.1 Vorteile

Die Verwendung von ECS bringt weitere Vorteile mit sich ohne dabei auf die vorher genannten Vorteile der Nutzung von Komposition zu verzichten. Die größten Vorteile dieses Architekturmusters werden im folgenden Abschnitt erörtert.

Optimierung des Speicherzugriffs

Die Optimierung des Zugriffs auf die im Speicher vorhandenen Daten ist ein großer Vorteil des datenorientierten Designs innerhalb des ECS. Da sich die Performance im Speicherbereich in den letzten Jahrzehnten nicht so stark verbessert hat wie die der Prozessoren (siehe Abbildung 2.9), ist es besonders wichtig darauf zu achten, wie Daten im Speicher geschrieben und gelesen werden.

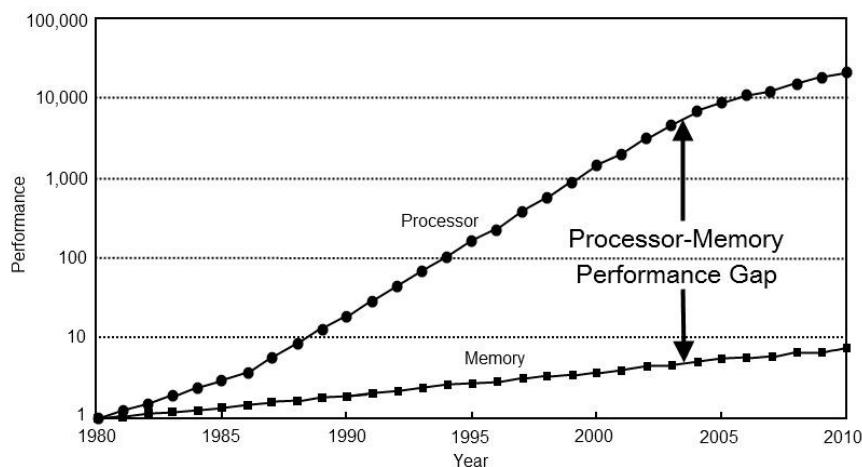


Abbildung 2.9: Steigerung der Performance von Prozessor und Speicher von 1980 bis 2010 [Cheney, 2019]

Da also der Zugriff auf Daten im Speicher verhältnismäßig lange dauert, gilt es diesen so gut wie möglich zu optimieren. Eine Optimierung wird durch moderne Central Processing Units (CPUs) vorgenommen, die kleine Bereiche zum Speichern, sogenannte *Caches*, besitzen. Wenn Daten aus dem Speicher angefordert werden, werden nicht nur die angeforderten Bytes, sondern auch die benachbarten Bytes in den Cache geladen. Sollten weitere Daten aus dem Cache benötigt werden, stehen diese dem Prozessor sofort zur Verfügung. Man spricht in diesem Fall von einem *cache hit*. Falls sich die Daten nicht im Cache befinden, muss der Prozessor so lange warten bis diese Daten aus dem Speicher geholt werden. In diesem Fall spricht man von einem *cache miss* [Nystrom, 2014, 333]. Jeder *cache miss* führt dazu, dass der Programmablauf sich verzögert.

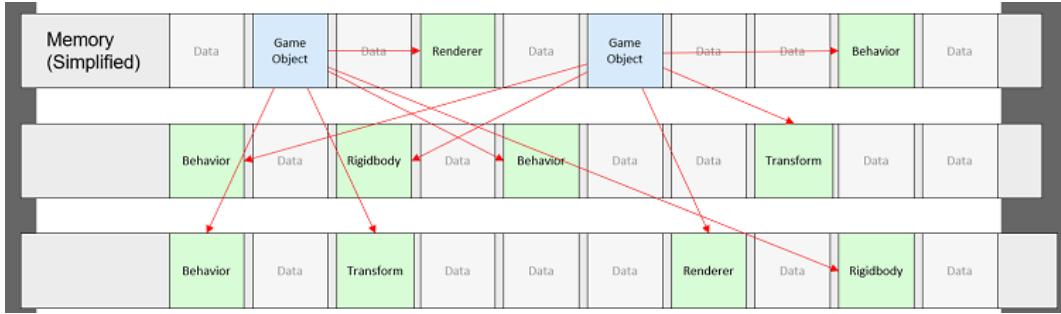


Abbildung 2.10: Vereinfachte Darstellung des Speichers von *GameObjects* und ihren Komponenten [Ferreira and Geig, 2018]

Idealerweise sollten die Daten im Speicher so angelegt werden, dass sie zusammenhängend ausgelesen werden, damit so wenig *cache misses* wie möglich entstehen. Wenn man sich jedoch ansieht, wie die Daten im Speicher bei der EC angelegt sind, kann man erkennen, dass es zu einigen *cache misses* führen wird. Dadurch, dass im ECS das Verhalten von den Daten entkoppelt wird, besteht die Möglichkeit die Daten der Komponenten zusammenhängend im Speicher anzulegen und folglich weniger *cache misses* zu erzielen.

Virtuelle Funktionen

Virtuelle Funktionen sind in der Programmierung eine Methode, die genutzt wird um Funktionsaufrufe dynamisch zu binden. Das heißt, dass erst zur Laufzeit feststeht, welche Version einer virtuellen Funktion aufgerufen wird. Virtuelle Funktionen sind streng mit der Vererbung oder dem Einsatz von Interfaces verbunden. Ein typischer Einsatz ist die Verwendung von virtuellen Funktionen in Basisklassen, die dann von Subklassen überschrieben werden. Um die virtuellen Funktionen dynamisch zu binden, verwenden die meisten Compiler virtuelle Tabellen, in denen Zeiger auf die Varianten einer virtuellen Funktion hinterlegt sind. Während der Laufzeit wird in der virtuellen Tabelle nachgeschaut, welche Variante einer Funktion in diesem Fall aufgerufen werden soll [Alex, 2018].

Das Nachschlagen der richtigen Funktion in der virtuellen Tabelle kann jedoch zu *cache misses* und damit zu Performanceeinbußen führen. Diese sind zwar in der Regel gering, können aber in Summe deutlich schwerer ins Gewicht fallen. Bei der EC wird in der Regel die Methode *Update* als virtuelle Funktion implementiert, die anschließend von allen Komponenten überschrieben wird. Durch die große Anzahl an unterschiedlichen Komponenten, die in einem Spiel normalerweise vorkommen, wird diese Funktion sehr oft überschrieben. Außerdem findet der Aufruf von *Update* bei jedem Frame, also in den meisten Fällen 60 mal pro Sekunde, statt und mit einem Aufruf wird nur eine einzige Komponente aktualisiert. Dadurch steigt die Zahl der Aufrufe der virtuellen Funktion *Update* sehr schnell und führt zu vielen *cache misses*.

Bei der Verwendung des ECS wird ebenfalls eine virtuelle Methode *Update* verwendet, um die Entitäten in der Welt zu aktualisieren. Diese Funktion wird jedoch von den Systemen und nicht den Komponenten implementiert. Dies führt im Vergleich zur EC zu deutlich weniger *cache misses*, weil es zum einen weniger Systeme als Komponenten gibt und zum anderen Systeme mit einem Aufruf der Methode *Update* alle Entitäten aktualisieren, die bestimmte Komponenten besitzen.

Nutzung von Multithreading

Dadurch, dass die Daten im ECS in zusammenhängenden Feldern im Speicher angelegt sind und die Systeme das Verhalten der Entitäten deutlich von den Daten trennen, ist die Verwendung von Multithreading naheliegend [Ferreira and Geig, 2018]. Moderne CPUs bestehen aus mehreren Kernen, die gleichzeitig genutzt werden können. Die Verwendung mehrerer Kerne ist allerdings kein Standard und kann bei unsauberer Implementierung Probleme mit sich bringen. Für die Nutzung von Multithreading sollte gewährleistet sein, dass der Code threadsicher ist, da es ansonsten zu sogenannten *Race Conditions* kommen kann. Der Zustand der Daten würde in diesen Fällen davon abhängen, welcher Thread schneller die Daten verändert kann.

Verfügbarkeit von Daten zur Rekonstruktion

Im ECS sind die Entitäten durch die Daten aus ihren Komponenten definiert. Was eine Entität machen kann, hängt davon ab, welche Komponenten zu ihr gehören. Die genaue Ausprägung, das Wie, wird durch die Daten in den Komponenten bestimmt. Beispielsweise stelle man sich eine Schnecke und einen Hasen vor. Beide Entitäten werden Komponenten zur Bewegung implementieren. Ein Bewegungssystem wird diese Komponenten nutzen, um beide Tiere zu bewegen. Wie sie sich bewegen, wie sie dabei aussehen und andere Eigenschaften werden nur durch die Daten bestimmt.

„Schließlich lässt sich der gesamte Spielzustand in Form aller Daten aller Komponenten einfach in einer relationalen Datenbank speichern. Jede Komponentenklasse wird auf eine Datenbanktabelle abgebildet. Die IDs der Entities, denen die Komponenten zugeordnet sind, dienen als Primärschlüssel.“ [Prühs, 2014, 31]

Dieser Vergleich trifft sehr gut den Kern der Idee, der hinter dem ECS steht. Dadurch, dass es möglich ist den gesamten Spielzustand in Form von Daten zu speichern, ergeben sich viele Vorteile, die bei der Verwendung von Vererbung nicht entstehen und bei der Verwendung einer EC nicht so ausgeprägt sind. Mit der Serialisierung der Daten lässt sich nicht nur das Laden und Speichern von Spielständen umsetzen, sondern es können auch Replays konstruiert und Logs ausgelesen werden. Für die Implementation eines Mehrspielermodus können die Systeme auf dem Server laufen und die

Klienten senden ihre Daten an den Server. Als Resultat befindet sich bei allen Klienten das Spiel im gleichen Zustand. Eine künstliche Intelligenz unterscheidet sich nicht großartig von einem Spieler. Sie erzeugt genauso Events, die von Systemen verarbeitet werden, wie ein Spieler. Um also einen Spieler mit einer K.I. auszutauschen, um zum Beispiel eine selbständig laufende Demo vom Spiel zu erstellen, müssen nur die Zustandsänderungen durch Events an die Systeme weitergegeben werden.

Keine überflüssigen Daten und Aktualisierungen

Beim ECS werden die Komponenten und Systeme nach dem *Single-Responsibility-Prinzip* in möglichst kleine Teile aufgeteilt. Jede Komponente und jedes System wird so entworfen, dass sie einen einzigen Zweck erfüllt, der von keiner anderen Komponente beziehungsweise keinem anderem System erfüllt wird. Dieses Prinzip bringt den Vorteil mit sich, dass keine überflüssigen Daten den Speicher füllen, keine unnötige Methoden aufgerufen werden und der Code übersichtlich bleibt.

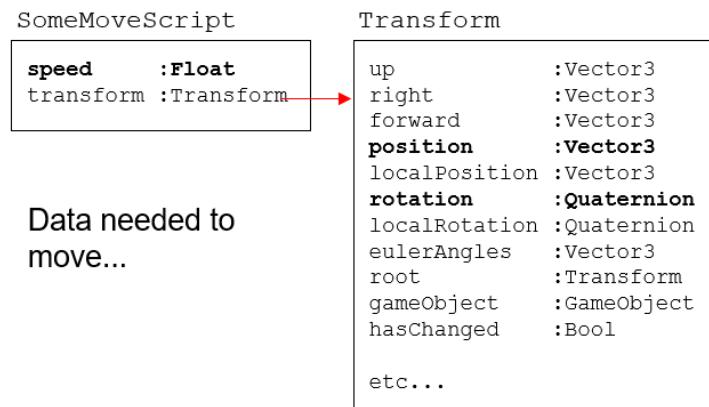


Abbildung 2.11: Daten der Komponente *Transform* und eines Skripts zur Bewegung von Entitäten in der Unity Engine [Ferreira and Geig, 2018]

Abbildung 2.11 zeigt die Daten, die in der Komponente *Transform* in der Unity Engine vorhanden sind und die Daten eines Skripts, das zur Bewegung von Entitäten dient. Die fett gedruckten Variablen, werden für die Bewegung einer Entität benötigt. Alle anderen Daten sind für die Bewegung eines Objektes irrelevant, reservieren jedoch Platz im Speicher. In einem ECS würde man die relevanten Daten in eigene Komponenten verschieben. So würde es wahrscheinlich folgende Komponenten geben: *MovementSpeedComponent*, *PositionComponent* und *RotationComponent*. Das *MovementSystem* würde nach allen Entitäten suchen, die alle drei Komponenten besitzen und die Position der Entität entsprechend ihrer Rotation um einen von der Bewegungsgeschwindigkeit abhängigen Wert verändern. Dies führt nicht nur zu einer Verbesserung der Performance, sondern gibt auch einen guten Überblick über den Zuständigkeitsbereich der Komponenten und Systeme.

2.3.2 Nachteile

Nachdem die Vorteile des ECS beschrieben worden sind, werden im folgenden Abschnitt die Nachteile erörtert, die dessen Verwendung mit sich bringt.

Eine ungewohnte Denkweise

Der wohl meist genannte Nachteil vom ECS und datenorientiertem Designs ist die ungewohnte Denkweise. Die meisten Entwickler sind durch ihre Ausbildung eine objektorientierte Denkweise gewohnt, die die reale Welt als Analogie für die Spielwelt nutzt. Diese Umstellung erfordert eine gewisse Eingewöhnungszeit, sollte jedoch keinen allzu großen Nachteil darstellen. Ein größeres Problem ist jedoch das Finden von Schnittstellen zu klassischen, objektorientierten Code[LLopis, 2009].

Komplexere Implementierung

Die Implementierung eines ECS kann aufgrund der ungewohnten Denkweise durchaus komplex sein, sollte aber mit zunehmender Erfahrung und Popularität zur Gewohnheit werden. Es gilt jedoch zu bedenken, dass ein Großteil der Performanceverbesserung durch die zusammenhängenden Felder im Speicher generiert wird. Dies bedeutet, dass eine benutzerdefinierte Speicherzuweisung implementiert werden sollte, um den vollen Nutzen aus dem ECS zu erhalten. Ein gut funktionierendes Event-System ist ebenfalls wichtig, damit die Systeme miteinander kommunizieren können. Dies ist zwar schwieriger zu implementieren als Komponenten direkt miteinander zu koppeln, ist jedoch langfristig besser zu pflegen und zu erweitern.

Lösung nicht vorhandener Probleme

Das ECS sollte nicht als Allzweckwaffe zur Lösung aller Probleme in der Entwicklung eines Videospiels gesehen werden. Es bietet viele Vorteile und hat einen eindeutigen Nutzen. In einigen Bereichen eines Spiels sind aber die Performanceverbesserungen, die dieses Architekturmuster mit sich bringt, nicht so entscheidend wie in anderen. Dort kann man auf objektorientierte Lösungen zurückgreifen oder auf Lösungen, die am besten zu diesem Bereich der Game Engine gehören. Randy Gaul, ein Software-Entwickler, der momentan für Microsoft arbeitet, nennt das Physik-System als einen dieser Bereiche und empfiehlt das ECS nur dort anzuwenden, wo es die beste Lösung ist.

„There's a problem. Blindly shoving the idea of an ECS implementation into every nook and cranny of an engine during development is just silly (or any complex system, not just game engines or libraries). Often times a particular system is not best implemented with a component or aggregate paradigm in mind.“ [Gaul, 2014]

Ein weiterer Grund, warum man das ECS nicht als Allzweckwaffe ansehen sollte ist, dass man damit eventuell Probleme löst, die im eigenen Spiel noch gar nicht vorhanden sind. Wie viele Muster, die sich großer Beliebtheit erfreuen, kann das ECS bei der Entwicklung von Videospielen mehr schaden als helfen. Videospiele sind auch ohne ein ECS oft performant genug um erfolgreich zu sein. Außerdem wird die Verbesserung der Performance meist erst bei einer großen Anzahl an Entitäten im Spiel signifikant. Wenn es also keine Performanceprobleme in Bereichen gibt, in denen das ECS helfen kann, gibt es auch keinen Grund Ressourcen in die Implementierung dessen zu stecken.

Debugging

Ein in der Entwicklung sehr relevanter Bereich, das Debugging, könnte durch die Verwendung eines ECS deutlich umständlicher zu implementieren sein als in anderen Architekturen. Der Nutzer *forsakenforgotten* beschreibt in einem reddit Thread zum Thema ECS seine Erfahrungen.

„One thing that I noticed from the beginning is that debug becomes usually marginally more difficult, specially when you just want to know what is going on with a specific entity on an array-per-component. In fact, many implementation proposals that I see has a lot of extra code and data that is mentioned to be used just to facilitate debugging (an entity/component manager returning all components of an entity, for example). If I've implemented it with an Entity class with an id attribute and a bag of different components (something that I usually see called an ECS as well), debugging would become quite easier.“ [forsakenforgotten, 2018]

Die Unity Engine bietet zum Debugging des ECS eine umfangreiche Benutzeroberfläche, den *Entity Debugger*. Mit diesem können alle wichtigen Informationen über die aktiven Entitäten, Systeme und Komponenten in Erfahrung gebracht werden.

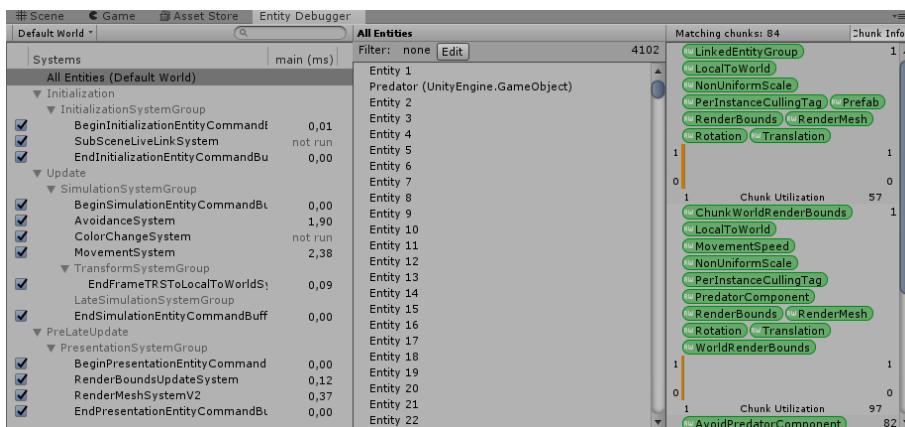


Abbildung 2.12: Der Entity Debugger der Unity Engine

2.3.3 Einsatzbereiche

Nachdem die Vor- und Nachteile des ECS erläutert worden sind, soll nun daraus gefolgert werden, für welche Anwendungen die Verwendung dieses Architekturmusters am besten geeignet ist. Aufgrund der komplexeren Implementierung sollte das ECS keineswegs in jeder Applikation und für alle Bereiche eingesetzt werden. Da die Performance-Verbesserungen sich erst bei einer großen Anzahl Entitäten in der Welt bemerkbar machen, sollten vor allem Anwendungen davon profitieren, die eine große, lebendige, virtuelle Welt erschaffen. Dazu gehören natürlich vor allem Videospiele, aber auch Applikationen aus anderen Bereichen, beispielsweise der Virtual Reality, können einen Nutzen aus der Verwendung eines ECS ziehen. Die Megacity Demo⁵, die von Unity 2018 zur Veröffentlichung des *Data-Oriented Technology Stacks* entwickelt wurde, zeigt eine solche große und lebendige Welt.

Im Bereich der Videospiele gibt es insbesondere zwei Genres, die von der Verwendung eines ECS profitieren können: Massive Multiplayer Online Roleplaying Games (MMORPGs) und Real Time Strategy Games (RTSs). MMORPGs spielen oft in einer großen, lebendigen Welt, in der viele Spieler gleichzeitig über das Internet miteinander verbunden sind. Die große Anzahl an Spielern und anderen Entitäten spricht für die Verwendung eines ECS. Neben diesem Aspekt bestehen MMORPGs ohnehin schon aus einer großen Menge an Daten, die für das Game Design nötig sind. Diese befinden sich im stetigen Wandel, um das Erlebnis für die Spieler abwechslungsreich zu gestalten. Durch die Verwendung eines ECS können Veränderungen in der Spielwelt innerhalb von kürzester Zeit ohne Veränderungen am Code von Designern vorgenommen werden. Auch RTSs besitzen eine große Menge an Daten und finden in einer belebten Welt mit einer großen Anzahl an Gebäuden, Zivilisten sowie militärischen Einheiten statt. Da in diesen Spielen oft Spieler gegeneinander antreten, sind Replays der Spiele für das Training sowie zur Unterhaltung eine wichtige Funktion. Die Architektur des ECS kann also besonders in diesen beiden Genres, aber auch in einigen anderen, hilfreich sein. Jedoch sollte man als Entwickler darauf achten, ob man von den Vorteilen des ECS profitiert und ob sich eine Implementierung lohnt.

Es gibt bereits einige Frameworks und Game Engines, die ein ECS implementieren beziehungsweise nutzen [Mertens, 2019]. Außerdem implementieren viele, nicht frei verfügbare, Game Engines ein ECS. Trotzdem gibt es noch viele andere Game Engines, die das ECS, nicht defaultmäßig nutzen und dies auch nicht planen, da es nicht notwendig ist, um erfolgreich Videospiele zu produzieren. Wie in vielen Bereichen der Softwareentwicklung muss man entscheiden, ob die Ressourcen, die für die Implementierung dieser Architektur erforderlich sind, in Relation zum Nutzen stehen.

⁵Mehr Infos unter: <https://unity.com/de/megacity>

3 Konzeption

Dieses Kapitel beschreibt die Konzeptionierung der Test-Applikationen innerhalb der Unity Engine und in eigenen Anwendungen, die in C++ programmiert sind. Zunächst wird auf das grundlegende Konzept eingegangen, das alle Applikationen gemeinsam haben. Anschließend wird im Detail die Konzeptionierung der beiden Applikationen in der Unity Engine sowie der beiden Applikationen in C++ besprochen.

3.1 Grundlegendes Konzept aller Test-Applikationen

Da das Ziel dieser Bachelorarbeit darin liegt zu prüfen, inwiefern das ECS zur Verbesserung der Performance in Videospielen führt, beruhen alle Applikation auf dem gleichen Konzept. Dies soll einen vergleichbaren Rahmen schaffen. Die Applikationen werden mit Absicht sehr simpel gehalten, um zum einen zu zeigen, dass schon bei simplen Anwendungen die Performance verbessert werden kann, und zum anderen, um die Performance-Tests in den Vordergrund zu stellen.

3.1.1 Ablauf der Applikationen

Alle Applikationen sind so aufgebaut, dass eine große Anzahl an Entitäten instanziert wird. Dabei ist das Ziel der Anwendung zu prüfen, wie viele Entitäten instanziert werden können bis die Framerate unter 30 Frames pro Sekunde (FPS) sinkt. 30 FPS gelten heutzutage als Mindest-Framerate in Videospielen. Um verschiedene Bereiche der Videospielentwicklung bei den Tests abzudecken, sollen die Entitäten drei verschiedene Verhaltensweisen implementieren können: die Bewegung mit einer konstanten Geschwindigkeit, die Veränderung der Farbe des Modells und die Vermeidung von anderen Objekten.

Die Bewegung mit einer konstanten Geschwindigkeit ist eine Verhaltensweise, die sehr einfach zu implementieren ist, da sie von keinen äußeren Umständen abhängt. Sie bezieht sich nur auf eine Entität und führt lediglich zu einer Veränderung der Position dieser Entität. Die Veränderung der Farbe des Modells ist eine ähnliche Verhaltensweise, die jedoch Einfluss auf einen der wichtigsten Bereiche einer Game Engine, das Rendering, hat. Bei der Vermeidung von Objekten wird die Position der Entität in Abhängigkeit zur Distanz zu einem anderen Objekt verändert. Diese Verhaltensweise ist also von äußeren Umständen abhängig und betrifft nicht alle Entitäten.

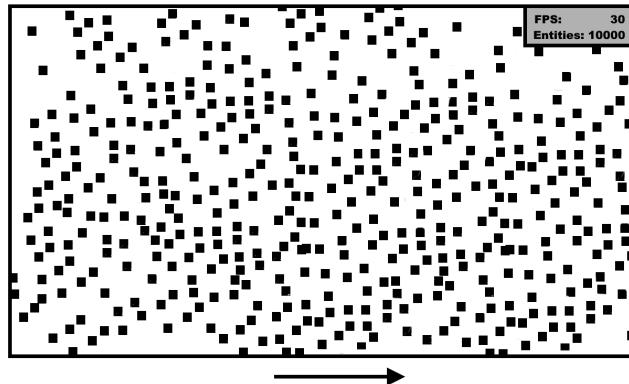


Abbildung 3.1: Skizziertes Design und Ablauf aller Test-Applikationen

Bei den Tests sollen die drei beschriebenen Verhaltensweisen als Parameter dienen. Das bedeutet, dass Varianten der Applikation erstellt werden, bei denen die Verhaltensweisen unterschiedlich kombiniert werden. Als Grundlage für alle Tests wird die Applikation wie folgt ablaufen: Zu Beginn wird eine gewünschte Anzahl an Entitäten am linken Rand des sichtbaren Bereichs mit einer zufälligen y-Position instantiiert. Die Entitäten bewegen sich mit einer zufällig festgelegten, aber konstanten, Geschwindigkeit nach rechts. Sobald die Entitäten den rechten Rand des sichtbaren Bereichs erreichen, werden sie mit einer neuen, zufälligen y-Position an den linken Rand zurückgesetzt. Es wird außerdem zu jeder Zeit angezeigt, mit wie vielen FPS die Applikation läuft und wie viele Entitäten in der Applikation aktiv sind. Die Veränderung der Farbe der Modelle sowie die Vermeidung von anderen Objekten kann sowohl einzeln als auch gemeinsam aktiv sein. Das zu vermeidende Objekt wird im weiteren Verlauf *Predator* genannt. Es bewegt sich entgegen aller anderen Objekte von rechts nach links.

3.1.2 Komplexität der verwendeten Modelle

Neben den unterschiedlichen Verhaltensweisen wird in den Tests ein weiterer Parameter verändert. Die Komplexität der verwendeten Modelle ist ein wichtiger Faktor bei der Entwicklung eines Videospiels. Daher werden in den Tests drei unterschiedlich komplexe Modelle verwendet und es wird geprüft, wie groß der Einfluss der Komplexität der Modelle auf die Performance ist. Da das Rendering auf der Graphics Processing Unit (GPU) stattfindet und das ECS die Nutzung der CPU optimiert, sollte die Veränderung dieses Parameters vor allem zeigen, was das ECS nicht leisten kann. Für die Modelle werden ein Würfel aus 12 Dreiecken, ein Fisch⁶ aus 672 Dreiecken und ein Affenkopf⁷ aus 15744 Dreiecken verwendet. Als Maßstab zur Kom-

⁶Modell erhalten von <https://free3d.com/3d-model/3d-fish-model-low-poly-63627.html>

⁷Modell erhalten von <https://github.com/BennyQBD/3DGameProgrammingTutorial/blob/master/res/models/monkey3.obj>

plexität dient die Anzahl der Dreiecke im 3D-Modell. Für gewöhnlich werden in der Computergrafik alle Modelle aus einer beliebigen Anzahl Dreiecke zusammengestellt.

3.1.3 Ziele der Test-Applikationen

Zusammenfassend können für das grundlegende Konzept aller Test-Applikationen folgende Ziele festgelegt werden:

1. In den Applikationen wird getestet, wie viele Entitäten gleichzeitig aktiv sein können bis die Framerate auf 30 FPS sinkt. Die Entitäten werden schrittweise durch die Betätigung der Leertaste instanziert.
2. Die Applikationen müssen untereinander vergleichbar sein. Das bedeutet, dass der Umfang und die Funktionalität aller Applikationen möglichst gleich sein soll.
3. In jeder Applikation bewegen sich alle Entitäten, mit Ausnahme des *Predators*, mit einer konstanten Geschwindigkeit, die zur Instanziierung zufällig gewählt wird, von links nach rechts. Der *Predator* bewegt sich von rechts nach links. Sobald eine Entität einen Seitenrand erreicht wird sie mit einer zufälligen y-Position an den anderen Seitenrand verschoben.
4. Für die Tests werden zwei veränderbare Parameter definiert. Der erste Parameter beschreibt die in der Applikation aktiven Systeme. Der zweite ist die Komplexität des zu rendernden 3D-Modells. Daraus ergeben sich für jede der insgesamt vier Umsetzungen zwölf Variationen.

3.2 Konzeption der Unity Applikationen

Der folgende Abschnitt stellt das Konzept der beiden Test-Applikationen in der Unity Engine vor. Dieses ist stark von den Standards der Unity Engine bezüglich der EC- und des ECS-Implementierungen bestimmt.

3.2.1 Entity Component Architecture in Unity

Die EC ist der aktuelle Standard in der Unity Engine. In diesem Abschnitt soll die Konzeption mit diesem Architekturmuster beschrieben werden.

Funktionsweise

Der übliche Workflow in Unity besteht daraus *GameObjects*, die Entitäten, zu Szenen hinzuzufügen und ihnen durch Komponenten Eigenschaften und Funktionalität zu geben. Diese Komponenten können vorgefertigte, wie zum Beispiel der *MeshRenderer*, oder eigene Komponenten sein. Die selbst kreierten Komponenten sind Skripte,

die von der Klasse *MonoBehaviour* erben. Die Klasse *MonoBehaviour* erbt, genauso wie alle vorgefertigten Komponenten der Unity Engine, von der Klasse *Component*. Dadurch, dass alle Komponenten von der Klasse *Component* erben und bestimmte Funktionen implementieren, wird die automatische Initialisierung und Aktualisierung aller Komponenten geregelt. Zu diesen Funktionen gehören *Start*, *Update*, *FixedUpdate*, *LateUpdate*, *OnGui*, *OnDisable* und *OnEnable*. Jedes *GameObject* besitzt außerdem die Komponente *Transform*, durch welche die Position in der Szene sowie die Rotation und Skalierung des *GameObjects* festgelegt werden.

Die Entitäten

Die Entität *Predator* wird als *GameObject* direkt in der Szene platziert. Es erhält die Komponenten *Transform*, *MeshFilter*, *MeshRenderer* und *MoveSideways*. Für die anderen Entitäten der Applikation wird ein sogenanntes *Prefab* verwendet. Ein *Prefab* ist eine Vorlage für *GameObjects*, das beliebig oft wiederverwendet werden kann, um *GameObjects* dieser Art zu instanziieren. Das Prefab für die angesprochene Test-Applikation beinhaltet folgende Komponenten:

- **Transform:** Diese Komponente muss bei jedem *GameObject* vorhanden sein.
- **MeshFilter:** Diese von Unity vorgefertigte Komponente hält den Verweis zum verwendeten 3D-Modell.
- **MeshRenderer:** Diese von Unity vorgefertigte Komponente ist für das Rendern des im *MeshFilter* angegebenen Modells zuständig. Außerdem besitzt es Informationen, die für das Rendering relevant sind. Dies beinhaltet zum Beispiel das Material oder das Lichtverhalten des Objektes.
- **MoveSideways:** Diese Komponente wird selbst erstellt. In ihr wird das Verhalten für die Bewegung entlang der x-Achse und das Zurücksetzen an den linken beziehungsweise rechten Rand implementiert. Außerdem wird die Bewegungsgeschwindigkeit des *GameObjects* gespeichert.
- **ChangeColor:** Diese Komponente wird ebenfalls selbst erstellt und implementiert die Farbänderung des *GameObjects*, indem sie die Farbe des Materials, welches im *MeshRenderer* gespeichert wird, verändert.
- **AvoidPredator:** Diese selbst erstellte Komponente implementiert das Verhalten zur Vermeidung eines bestimmten *GameObjects*, dem *Predator*, welches als Referenz zur Komponente hinzugefügt wird.

Die Klasse *DemoManager*

Für die Instanzierung der Prefabs wird die Klasse *DemoManager* genutzt. Auch diese Klasse erbt von *MonoBehaviour* und wird als Komponente einem leeren *GameObject*

hinzugefügt. Der *DemoManager* erhält eine Referenz zu dem Prefab, welches mit den oben genannten Komponenten instanziert werden soll. Wenn die Leertaste betätigt wird, soll eine bestimmte Anzahl an *GameObjects* instanziert werden, die das Prefab als Vorlage nutzen. Außerdem werden die x-Koordinaten der Seitenränder der sichtbaren Fläche als statische Variablen im *DemoManager* gespeichert. Diese werden bei der Instanziierung und in der Komponente *MoveSideways* benötigt.

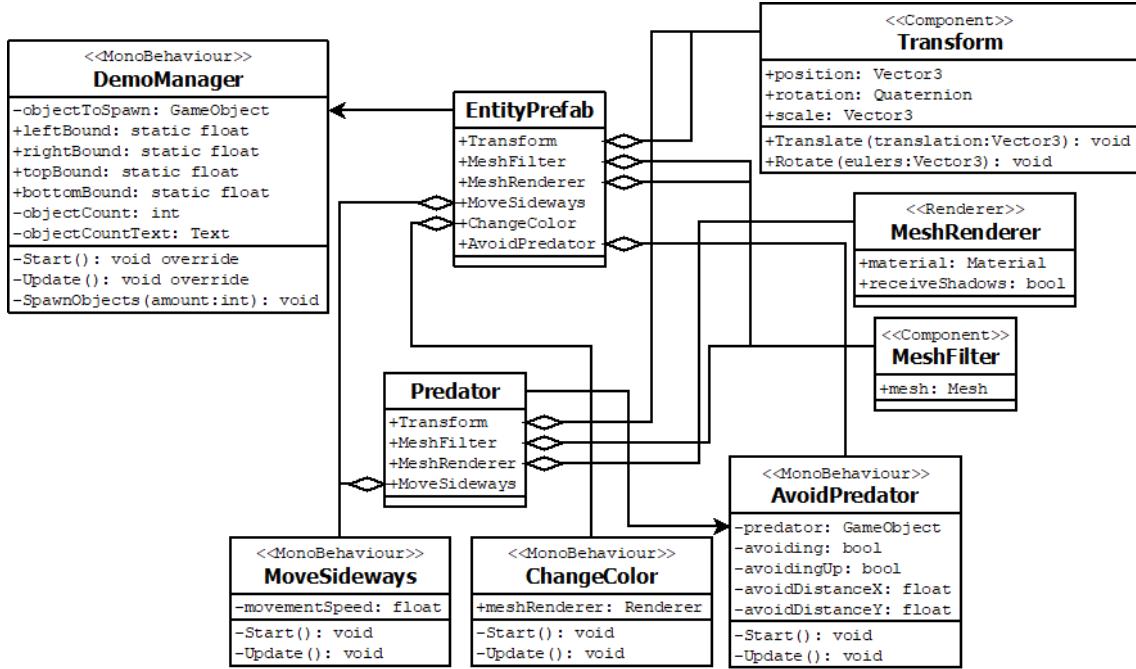


Abbildung 3.2: UML-Diagramm, reduziert auf die relevanten Informationen, zur Konzeption der Applikation mit der EC in der Unity Engine

Eine weitere Aufgabe der Klasse *DemoManager* ist die Aktualisierung der Anzahl der Objekte im User Interface. User Interfaces werden in Unity mit der Komponente *Canvas* implementiert. Alle Elemente des User Interfaces müssen als Kinder des *GameObjects*, das die Komponente *Canvas* besitzt, hinzugefügt werden. Unity bietet eine Vielzahl an üblichen Elementen für die Gestaltung eines User Interfaces an, die ebenfalls als Komponenten hinzugefügt werden. Für die geplante Applikation reichen Textelemente zur Anzeige der FPS und der Anzahl der Entitäten aus. Diese werden über die Skripte *ShowFPS* beziehungsweise *DemoManager* aktualisiert.

3.2.2 Entity Component System in Unity

Das ECS der Unity Engine befindet sich momentan noch in der Entwicklungsphase und ist in der aktuellen Version (2019.2.0f1) als Preview-Package verfügbar. Jedoch ist geplant, dass die Verwendung dieses Architekturmusters in Zukunft zum Standard für die Entwicklung in Unity wird.

Setup

Um das ECS nutzen zu können, müssen über den Package Manager die Pakete *Entities* und *Hybrid Renderer* hinzugefügt werden. Das Konzept für die Applikation mit dem ECS ist dem zur EC in vielen Aspekten ähnlich. Da Unity momentan die EC als Standard nutzt, wird das ECS als Hybrid verwendet. Dies bedeutet, dass in einigen Bereichen die herkömmliche Architektur mit *GameObjects* und Komponenten zum Einsatz kommt.

Gemeinsamkeit mit der Entity Component Architecture

Einer dieser Bereiche ist das User Interface, das so wie im vorherigen Abschnitt beschrieben, implementiert wird. Wie auch in der Applikation mit der EC wird zur Instanziierung der Entitäten ein *DemoManager* verwendet, der von der Klasse *MonoBehaviour* erbt und als Komponente zu einem leeren *GameObject* hinzugefügt wird. Es wird eine bestimmte Anzahl von Entitäten instanziert, wenn die Leertaste gedrückt wird. Außerdem ist der *DemoManager* für die Aktualisierung der Anzahl der Entitäten zuständig und hält Referenzen zu der Entität, die vermieden werden soll.

Erstellung von Entitäten

Der bedeutendste Unterschied zur Konzeption der Applikation mit der EC ist die Verwendung von Entitäten, Komponenten und Systemen anstelle der *GameObjects* mit Komponenten. Um Entitäten zu erstellen gibt es verschiedene Möglichkeiten. Eine Möglichkeit ist die *GameObjects* in Entitäten umzuwandeln. Dies kann über die Skript-Komponente *ConvertToEntity* oder über die Funktion *CovertGameObjectHierarchy* aus dem Paket *Entities* erreicht werden. Eine weitere Möglichkeit zur Erstellung von Entitäten ist die Verwendung der Funktion *CreateEntity* der Klasse *EntityManager*. Diese nimmt als Parameter entweder ein Array von *ComponentTypes* oder einen *EntityArchetype* an. Das Array von *ComponentTypes* setzt sich aus dem *ComponentType* der gewünschten Komponenten für diese Entität zusammen. Der *EntityArchetype* beschreibt eine einzigartige Kombination von Komponenten. Dieser wird ebenfalls dafür genutzt Entitäten in Gruppen einzuordnen und sie fortlaufend im Speicher anzulegen. Komponenten können jederzeit zu Entitäten hinzugefügt oder entfernt werden. Dies führt dazu, dass sie einem anderen *EntityArchetype* entsprechen. Außerdem bietet der *EntityManager* über die Funktion *Instantiate* die Möglichkeit neue Entitäten als Kopie einer bereits vorhandenen Entität zu erstellen.

Als Vorlage für die Instanziierung von Entitäten wird ein *Prefab* genutzt, das auf einem *GameObject* mit den Komponenten *Transform*, *MeshFilter* und *MeshRenderer* basiert. Im *DemoManager* wird dieses *GameObject* in eine Entität umgewandelt. Dabei wird zum Beispiel die Komponente *Transform* unter anderem in die Komponenten *Translation* und *Rotation* umgewandelt, da die Komponenten in einem ECS anders aufgebaut sein müssen als in der EC. Der Entität werden außerdem weitere,

3. Konzeption

eigene Komponenten hinzugefügt: *MovementSpeed*, *AvoidPredator* und *ChangeColor*. Diese Entität wird als Vorlage für die Instanziierung mit der Funktion *Instantiate* vom *EntityManager* genutzt.

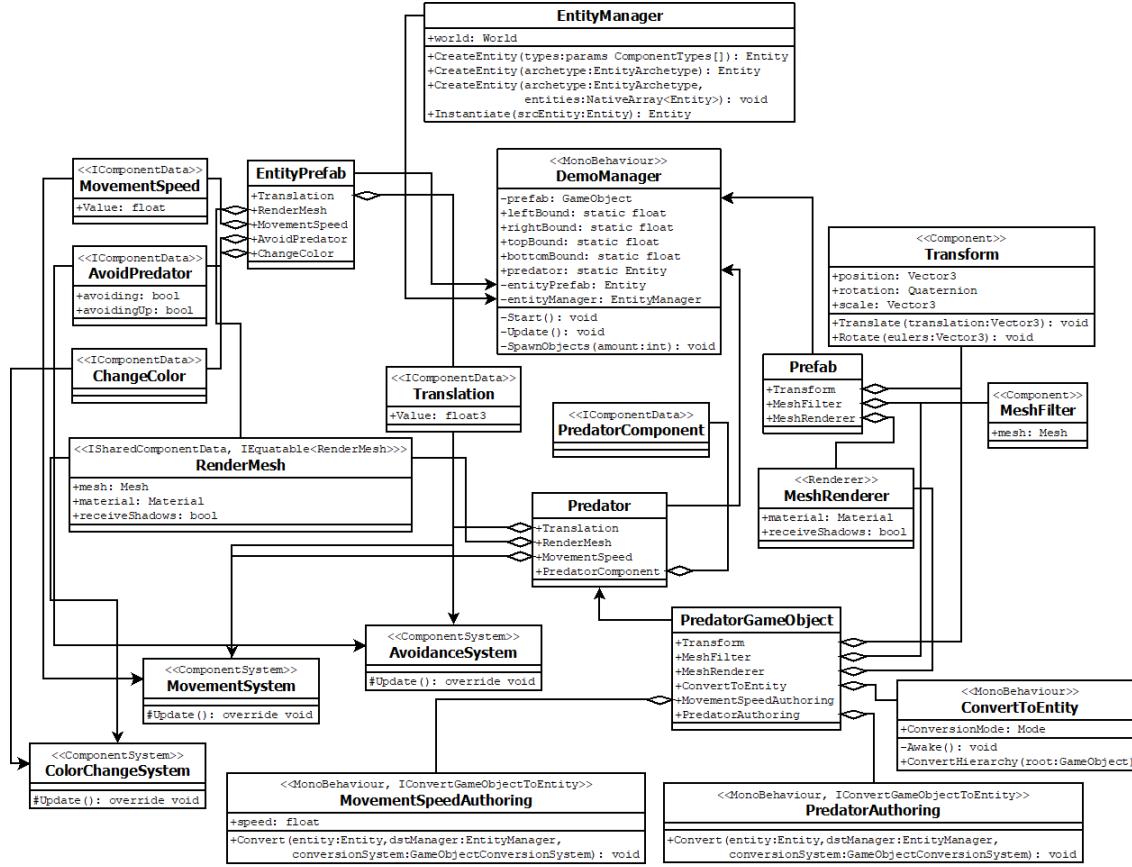


Abbildung 3.3: UML-Diagramm, reduziert auf die relevanten Informationen, zur Konzeption der Applikation mit dem ECS in der Unity Engine

Eine weitere Entität, die in der Applikation benötigt wird, ist die Entität *Predator*. Sie wird zunächst als *GameObject* in der Szene positioniert und besitzt die Komponenten *Transform*, *MeshFilter*, *MeshRenderer*, *ConvertToEntity*, *MovementSpeedAuthoring* und *PredatorAuthoring*. Die Komponente *ConvertToEntity* ist ein von Unity bereitgestelltes *MonoBehaviour*, welches das *GameObject* in eine Entität umwandelt. Die beiden Komponenten *MovementSpeedAuthoring* und *PredatorAuthoring* sind eigene Komponenten, die von *MonoBehaviour* erben und das Interface *IConvertGameObjectToEntity* implementieren. Diese beiden *MonoBehaviours* fügen die Komponenten *MovementSpeed* und *Predator* zur Entität hinzu. Die Komponente *Predator* dient als Tag-Komponente. Sie wird nur dazu verwendet die Entität eindeutig identifizieren zu können und besitzt keinerlei Daten.

Die Systeme

Für das Verhalten der Entitäten sind die Systeme zuständig. Unity stellt einige Systeme, zum Beispiel das *RenderMeshSystemV2* zum Rendern von *Meshes* bereit. Für die Applikation werden drei eigene Systeme entwickelt:

- Das **MovementSystem** ist für die Bewegung von Entitäten zuständig, die die Komponenten *Translation* und *MovementSpeed* besitzen. Es verändert die x-Position in Abhängigkeit von der Bewegungsgeschwindigkeit und der vergangenen Zeit. Außerdem setzt es die Entität an eine zufällige Startposition zurück, sobald einer der Seitenränder erreicht wird.
- Das **AvoidanceSystem** ist dafür zuständig, dass die Entitäten, die die Komponenten *Translation* und *AvoidPredator* besitzen, der Entität mit der Komponente *Predator* ausweichen.
- Das **ColorChangeSystem** ist für die Veränderung der Farbe der Entitäten zuständig, die die Komponente *ChangeColor* besitzen.

Die Systeme werden automatisch zu jedem Frame über die Funktion *Update* aktualisiert, wenn mindestens eine Entität vorhanden ist, die die gesuchten Komponenten besitzt.

3.3 Konzeption der Test-Applikationen in C++

In diesem Abschnitt wird die Konzeption der Test-Applikationen in der Programmiersprache C++ beschrieben. Dazu wird zunächst auf die Technologien und Bibliotheken eingegangen, die bei der Entwicklung verwendet werden. Anschließend werden die grundlegenden Funktionen konzeptioniert und im Anschluss wird auf die Konzeption der EC beziehungsweise des ECS eingegangen.

3.3.1 Verwendete Technologien und Bibliotheken

Entwicklungsumgebung und Compiler

Als Entwicklungsumgebung für die Programmierung in C++ wird *Visual Studio 2019* verwendet. Damit einhergehend wird als Compiler der Standard-Compiler von Visual Studio, der Microsoft Visual C++ Compiler verwendet. Visual Studio 2019 bietet neben einer übersichtlichen Entwicklungsumgebung hilfreiche Debugging-Tools, wie zum Beispiel den Profiler, an. Damit kann analysiert werden, an welcher Stelle im Code wie viel CPU-Zeit verbraucht wird. Außerdem ist die Verwendung von Visual Studio als Entwicklungsumgebung naheliegend, da aufgrund eines vorherigen Projekts und der Entwicklung mit der Unity Engine einige Erfahrungen vorhanden sind.

Die wichtigsten Bibliotheken

Für die Entwicklung der Test-Applikationen werden einige Bibliotheken verwendet:

- **OpenGL**⁸ ist eine Grafikbibliothek der *Khronos Group* und gehört neben Microsofts *Direct3D* und *Vulkan* zu den Standard-Grafikbibliotheken, die in der Industrie verwendet werden. Eine Grafikbibliothek wird als Schnittstelle zur Entwicklung von 2D- und 3D-Computergrafikanwendungen verwendet.
- **GLFW**⁹ ist eine Bibliothek, mit der man, unter anderem für *OpenGL*, Fenster erstellen sowie Eingaben und Events abfragen kann.
- **glad**¹⁰ ist eine Bibliothek, die OpenGL-Funktionen zur Laufzeit lädt. Dies ist auf den meisten Plattformen seit OpenGL 1.1 nötig und daher unumgänglich.
- **OpenGL Mathematics**¹¹ ist eine Mathematikbibliothek, die darauf ausgelegt ist für Grafiksoftware verwendet zu werden.
- **Dear ImGui**¹² ist eine Bibliothek zur Erstellung von Benutzeroberflächen. Es bietet zahlreiche Funktionen und bereits vorgefertigte Elemente zur Gestaltung von Benutzeroberflächen. Außerdem ist es leicht einzubinden, da es keine weiteren Bibliotheken nutzt und unabhängig von der gewählten Grafikbibliothek ist.
- **Assimp**¹³ bietet Funktionen zum Importieren und Exportieren von 3D-Modellen.

3.3.2 Grundlegende Funktionalität in C++

Die grundlegende Funktionalität, die zum reibungslosen Ablauf der Applikation nötig ist, ist in der Unity Engine bereits implementiert. In den Anwendungen in C++ muss diese Basis jedoch selbst gelegt werden. Zu der grundlegenden Funktionalität gehören die Erstellung eines Fensters, in dem die Applikation läuft, das Rendering sowie ein User Interface zur Anzeige der FPS und der Anzahl der Objekte. Da die Anwendungen relativ simpel sind, wird auf einen Großteil der Abstraktion, die bei einem größeren Projekt notwendig ist, verzichtet. Trotzdem soll darauf geachtet werden, dass das *Single Responsibility Prinzip* eingehalten wird.

⁸Weitere Infos zu OpenGL: <https://www.opengl.org>

⁹Weitere Infos zu GLFW: <https://www.glfw.org>

¹⁰Weitere Infos zu glad: <https://github.com/Dav1dde/glad>

¹¹Weitere Infos zu glm: <https://glm.g-truc.net/0.9.9/index.html>

¹²Weitere Infos zu Dear ImGui: <https://github.com/ocornut/imgui>

¹³Weitere Infos zu assimp: <https://github.com/assimp/assimp>

Die Klasse Application

Die Klasse *Application* dient als Herzstück der Anwendungen. Über sie werden unter anderem das Fenster, das Rendering und das User-Interface initialisiert. Da die Anwendungen relativ simpel gehalten werden und wenig Abstraktion beinhalten müssen, werden in der Klasse *Application* außerdem die benötigten Ressourcen, Modelle und *Shader* geladen sowie Entitäten mit den passenden Komponenten beziehungsweise Komponenten und Systemen erstellt. In umfangreicheren Projekten würde man diese Aufgaben zum Beispiel in die Klassen *ResourceManager* und *World* auslagern. Weiterhin enthält diese Klasse die Funktionalitäten die normalerweise vom sogenannten *Game-Loop* übernommen werden. Der *Game-Loop* ist eine Schleife, die erst beim Schließen der Applikation beendet wird. Innerhalb dieser Schleife findet die ständige Aktualisierung der Anwendung statt. Neben den bisherigen Funktionen implementiert *Application* außerdem die Funktion *OnEvent*, welche die Events, die von *GLFW* registriert werden, an alle Bereiche der Applikation weiter. Im UML-Diagramm lässt sich erkennen, wie die Klasse *Application* grundsätzlich aufgebaut ist.

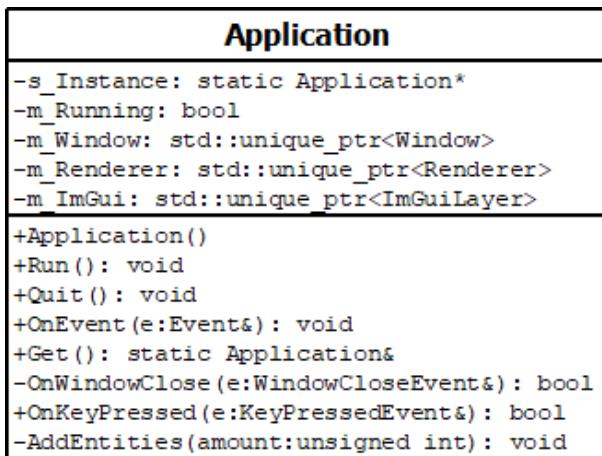


Abbildung 3.4: Vereinfachtes UML-Diagramm der Klasse Application

Das Fenster

Das Fenster, in dem die Applikation läuft, wird mithilfe der Bibliothek *GLFW* erstellt. *GLFW* bietet eine Schnittstelle, um Fenster mit OpenGL zu erstellen und Eingaben als Events über Callbacks abzufragen. Als Schnittstelle zu *GLFW* dient die Klasse *Window*, die ein *GLFWWindow* initialisiert und erstellt. Die von *GLFW* hervorgerufenen Events werden in applikationsspezifische Events umgewandelt. Wenn eine Klasse bei einem Event benachrichtigt werden soll, implementiert sie die Klasse *EventDispatcher* und stellt eine Callback-Funktion bereit, die im Fall, dass das Event ausgelöst wird, aufgerufen wird. Im folgenden UML-Diagramm lässt sich erkennen, wie die Klasse *Window*, die Events und *GLFW* miteinander verknüpft sind.

3. Konzeption

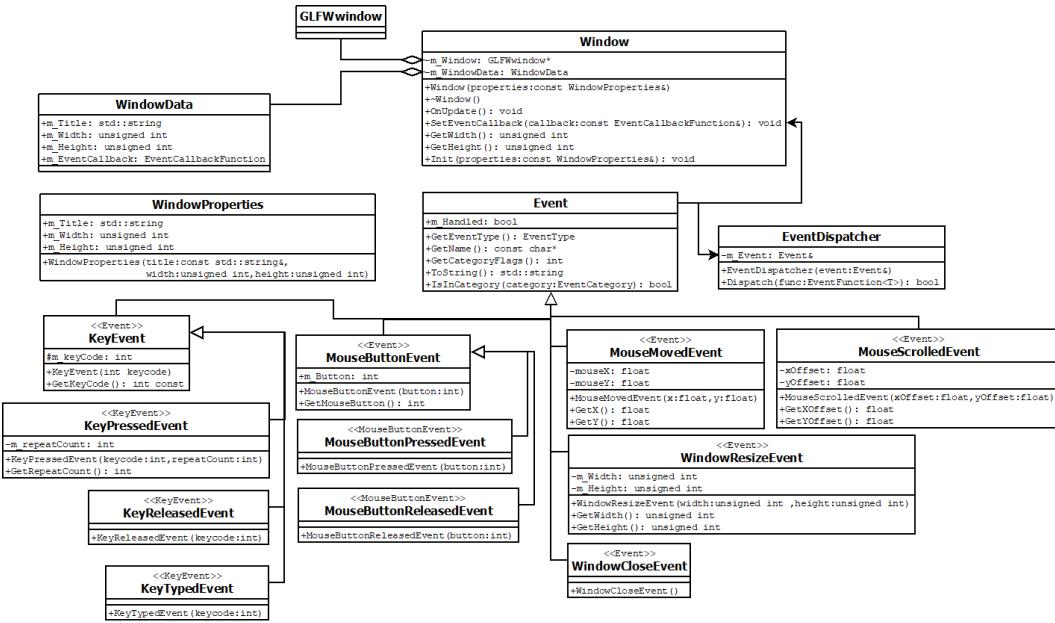


Abbildung 3.5: UML-Diagramm zur Funktionsweise von Window, Events und GLFW

Rendering

Das Rendering gehört zu den wichtigsten Bereichen einer 2D- beziehungsweise 3D-Applikation. Für die Test-Applikationen wird die Grafikbibliothek *OpenGL* in Verbindung mit *glad* verwendet. Da in dieser Arbeit das Hauptaugenmerk jedoch nicht auf dem Rendering liegt, wird dieser Bereich nur oberflächlich erklärt¹⁴. Einen Großteil der benötigten Funktionalität für das Rendering übernimmt *OpenGL*, dessen Prozesse hauptsächlich auf der GPU ablaufen. Der Ablauf des Renderings einer dreidimensionalen Welt lässt sich vereinfacht wie folgt beschreiben: Es wird eine virtuelle, dreidimensionale Welt erstellt, in der Objekten bestimmte Koordinaten zugeordnet werden. Diese Koordinaten sowie Informationen zur Perspektive und Blickrichtung der Kamera werden an sogenannte *Shader* weitergegeben. Die *Shader* berechnen aufgrund dieser Informationen, wie das zweidimensionale Bild, welches auf dem Bildschirm zu sehen ist, aussehen muss. Sie berechnen für jeden Pixel des Bildes, welche Farbe dieser Pixel hat. Für das Rendering in beiden Test-Applikationen werden folgende Klassen erstellt:

- **Renderer:** Diese Klasse ist für die Initialisierung von *OpenGL* zuständig. Außerdem implementiert sie die Funktion *Clear*, durch die zu Beginn jedes Frames das Bild aus dem vorherigen Frame einfarbig übermalt wird. Da die Kamera in beiden Test-Applikationen statisch ist, dient diese Klasse außerdem dazu die für das Rendering im *Shader* benötigten Matrizen für die *projection* und den *view*

¹⁴Wer mehr über das Rendering mit OpenGL erfahren möchte: <https://learnopengl.com>

zu berechnen und speichern. Die Matrix *projection* beschreibt die Perspektive der Kamera, während die Matrix *view* die Blickrichtung der Kamera beschreibt.

- **Mesh:** Diese Klasse beschreibt die Form eines Objekts und ist für die Initialisierung dessen sowie die Übergabe der Daten an die *Shader* zuständig. Die Informationen über die Form des Objektes wird in zwei Arrays gespeichert, den *Vertices* und *Indices*.
- **Model:** Diese Klasse dient als Container für *Meshes*. Modelle können aus vielen *Meshes* bestehen. In den beiden Test-Applikationen wird diese Klasse auch als Schnittstelle zur Bibliothek *assimp* verwendet, mit der vorgefertigte Modelle geladen werden können. In der Klasse *Model* erfolgt dabei die Übersetzung der Informationen aus der Modell-Datei zur Repräsentation als eine Ansammlung von *Meshes*.
- **Shader:** Diese Klasse dient als Schnittstelle von den *Shader-Dateien* zu den von OpenGL verwendeten *Shadern*. Des Weiteren bietet sie Funktionen zum Setzen von Variablen, die in *Shadern* benötigt werden.

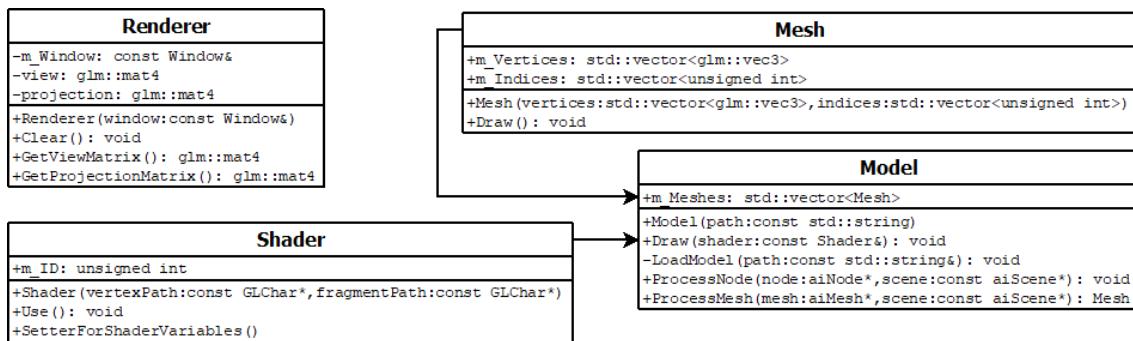


Abbildung 3.6: UML-Diagramm zur Funktionsweise von Renderer, Mesh, Model und Shader

Außerdem kommen in der EC und im ECS die Komponente *Renderable* sowie das System *RenderSystem* in der ECS hinzu. Die Konzeption dieser Klassen wird in Abschnitt 3.3.3 beziehungsweise 3.3.4 näher beleuchtet.

User Interface

Das User-Interface wird zur Anzeige der FPS und Anzahl der in der Spielwelt aktiven Entitäten verwendet. Für die Test-Applikationen wird die Bibliothek *Dear ImGui* genutzt. *Dear ImGui* bietet eine Vielzahl vorgefertigter UI-Elemente, die einfach zu verwenden sind. Für die Anwendungen wird eine transparente Box am oberen rechten Rand platziert, die als Text die FPS und die Anzahl der Entitäten anzeigt. Als

Schnittstelle zu *Dear ImGui* wird die Klasse *ImGuiLayer* genutzt, die *Dear ImGui* initialisiert und in der Funktion *OnUpdate* die angezeigten Informationen aktualisiert.

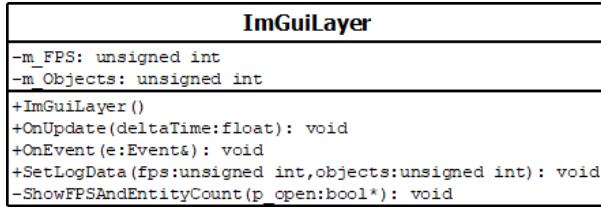


Abbildung 3.7: UML-Diagramm der Klasse *ImGuiLayer*

3.3.3 Entity Component Architecture in C++

Die eigene Implementierung der EC in der Programmiersprache C++ orientiert sich an der Architektur, die in der Unity Engine verwendet wird. Vor allem die Syntax und die grundlegende Funktionalität von Komponenten und Entitäten sind Vorbild für die Umsetzung.

Die Klasse Entity

Die Entitäten werden durch die Klasse *Entity* repräsentiert. Eine Entität besitzt eine beliebige Anzahl Komponenten. Außerdem sind Entitäten hierarchisch aufgebaut. Das heißt, dass Entitäten eine übergeordnete Entität, den sogenannten *Parent*, und eine beliebige Anzahl untergeordneter Entitäten, sogenannter *Children*, haben können. Zur Identifizierung einer Entität erhält diese einen Namen und eine einzigartige *ID*. Entitäten können aktiv oder inaktiv sein. Zum Beispiel ist es sinnvoll eine Entität zu deaktivieren, wenn diese momentan nicht, aber später wieder benötigt werden könnte. Ob eine Entität aktiv ist oder nicht wird in einem booleschen Ausdruck gespeichert. Nur aktive Entitäten werden aktualisiert oder erhalten Events. Die Klasse *Entity* stellt außerdem Funktionen zum Hinzufügen, Entfernen und Erhalten von Komponenten und untergeordneten Entitäten bereit. Eine Entität wird über die Funktion *OnUpdate* aktualisiert. In dieser Funktion wird zunächst die Funktion *OnUpdate* aller Komponenten und anschließend aller untergeordneten Entitäten dieser Entität aufgerufen. So werden alle Entitäten sukzessive aktualisiert. Die Klasse *Entity* implementiert ebenfalls eine Schnittstelle zum Erhalten von Events. Die Funktion *OnEvent* wird aufgerufen und leitet die Events an alle Komponenten und untergeordneten Entitäten weiter. Eine Besonderheit ist die Funktion *Draw*, die spezifisch für das Rendering von Entitäten eingesetzt wird. Diese Funktion wird erst am Ende des *Game-Loops* aufgerufen, sobald alle Entitäten und ihre Komponenten aktualisiert worden sind.

Die Klasse Component

Komponenten sind in der EC Klassen, die sowohl Eigenschaften als auch Verhalten einer Entität implementieren. Die Klasse *Component* ist die Superklasse, von der alle Komponenten erben. Um das Spektrum der Funktionalität von möglichen Komponenten abzudecken, werden virtuelle Funktionen in der Klasse definiert. Diese können von den abgeleiteten Klassen implementiert werden. Die Funktion *OnUpdate* wird bei jedem Frame aufgerufen, die Funktion *OnEvent* wird aufgerufen, wenn ein Event ausgelöst wird, und die Funktion *Draw* wird zum Rendering am Ende des *Game-Loops* aufgerufen. Des Weiteren erhält die Klasse *Component* einen Verweis zu der Entität, die diese Komponente besitzt, um Interaktionen zwischen anderen Komponenten oder direkt mit der Entität zu ermöglichen. Außerdem kann eine Komponente aktiviert oder deaktiviert werden. Dieser Zustand wird in dem booleschen Ausdruck *enabled* gespeichert. So kann zum Beispiel bei einer Taschenlampe das Licht ausgeschaltet werden, ohne die Lampe unsichtbar zu machen. Dafür wird die Komponente *SpotLight* deaktiviert, aber die Entität mit allen anderen Komponenten bleibt aktiv. Nur aktive Komponenten werden aktualisiert oder erhalten Events.

Die einzelnen Komponenten

Für die Test-Applikationen werden, ähnlich wie bei den Anwendungen in der Unity Engine, nur wenige Komponenten benötigt. Diese Komponenten erben von der Klasse *Component* und implementieren die Eigenschaften und das Verhalten der Entitäten. Im Folgenden werden die benötigten Komponenten beschrieben:

- **Transform:** Diese Komponente ist der gleichnamigen Komponente aus der Unity Engine nachempfunden und beinhaltet Informationen über die Position, Rotation und Skalierung einer Entität. Des Weiteren bietet sie Funktionen zur Veränderung dieser Variablen und zur Berechnung der Transformationsmatrix, die für das Rendering an die *Shader* weitergegeben werden muss.
- **Renderable:** Diese Komponente implementiert als einzige der verwendeten Komponenten die Funktion *Draw*. Sie besitzt Verweise zu dem verwendeten *Model* und *Shader* der Entität und ruft die Methode *Draw* der Klasse *Model* auf. Außerdem gibt sie die für das Rendering benötigten Matrizen an den *Shader* weiter.
- **Movement:** Diese Komponente ist für die Bewegung einer Entität zuständig. Sie besitzt die Information über die Bewegungsgeschwindigkeit der Entität und implementiert in der Funktion *OnUpdate* die Veränderung der Position anhand der Bewegungsgeschwindigkeit und der Zeit, die seit dem letzten Update vergangen ist.
- **ColorComponent:** Diese Komponente bestimmt, in welcher Farbe das Modell der Entität dargestellt wird. Außerdem implementiert sie in der Funktion

3. Konzeption

OnUpdate die Farbänderung der Entität.

- **Predator:** Diese Komponente dient zur Identifikation der Entität *Predator* und aktualisiert in der Funktion *OnUpdate* die Referenz zur aktuellen Position des *Predators*.
- **AvoidPredator:** Diese Komponente implementiert in der Funktion *OnUpdate* das Verhalten der Entität *Predator* zu vermeiden und nach erfolgreicher Vermeidung zur ursprünglichen y-Position zurückzukehren. Außerdem wird darin die Information gespeichert, ob eine Entität aktuell dem *Predator* ausweicht oder nicht.

Übersicht der Entity Component Architecture

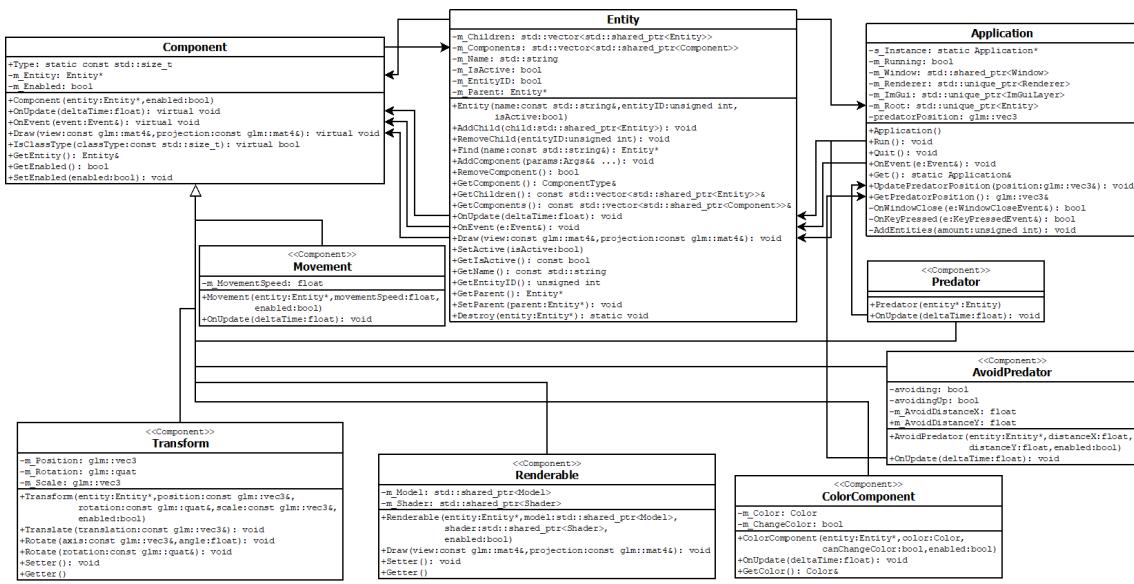


Abbildung 3.8: UML-Diagramm zur EC in C++

Die Klasse *Application* besitzt eine einzigartige Entität, die *Root* genannt wird. Alle weiteren Entitäten sind untergeordnete Entitäten von *Root*. Dadurch können alle Entitäten im *Game-Loop* in der Funktion *Run* mit dem Aufruf der Funktionen *OnUpdate* beziehungsweise *Draw* erreicht werden. Über die Funktion *OnEvent* der Entität *Root* werden außerdem Events an alle anderen Entitäten weitergeleitet.

3.3.4 Entity Component System in C++

Im ECS werden Daten und Verhalten aufgeteilt. Es gibt Komponenten, die nur aus Daten bestehen, und Systeme, die für das Verhalten der Entitäten verantwortlich sind. Die Implementierung eines ECS lässt sich auf verschiedene Arten lösen. Bei der für diese Arbeit verwendeten Lösung handelt es sich um eine Adaption der Implementierung des Github-Nutzers *BennyQBD*¹⁵.

Die Komponenten

Für die Architektur von Komponenten werden drei Ebenen der Vererbung verwendet. Als Superklasse fungiert die Struktur *BaseComponent*. Darin werden einige statische Funktionen und Variablen definiert, die dafür genutzt werden können Komponenten anhand ihres Typs einzuordnen. Der Komponententyp wird als Kombination der *ComponentCreateFunction*, *ComponentFreeFunction* und *size* im Array *componentTypes* gespeichert. Die *ComponentCreateFunction* ist eine Funktion zur Erstellung einer bestimmten Komponente, die *ComponentFreeFunction* ist eine Funktion zum Löschen einer bestimmten Komponente und die Variable *size* gibt die Größe der Komponente im Speicher an. Jeder neue Komponententyp wird über die Funktion *RegisterComponentType*, die diesen Komponententyp zum Array *componentTypes* hinzufügt, registriert. Außerdem besitzt jede *BaseComponent* eine Referenz zu der Entität, zu der diese Komponente zugeordnet werden kann.

Die Struktur *Component* erbt von *BaseComponent*. *Component* ist ein Struktur-Template, was den Vorteil hat, dass die Erstellung von Komponenten generisch stattfinden kann. Eine *Component* besteht aus den statischen Variablen *ID*, die zur Identifikation des Komponententyps dient, sowie der *ComponentCreateFunction*, der *ComponentFreeFunction* und der Größe dieses Komponententyps. Alle unterschiedlichen Komponententypen erben von der Klasse *Component*.

Die Komponenten, die für die Applikation erstellt werden, sind denen aus der EC sehr ähnlich. Der wichtigste Unterschied ist, dass diese Komponenten reine Datencontainer sind. Folgende Komponenten werden erstellt:

- **TransformComponent:** Diese Komponente beinhaltet die Daten zur Position, Rotation und Skalierung einer Entität.
- **RenderableComponent:** Diese Komponente beinhaltet die Daten zum verwendeten *Model* und *Shader* einer Entität.
- **ColorComponent:** Diese Komponente beinhaltet die für das Rendering verwendete Farbe einer Entität.

¹⁵Seine Implementierung findet man hier: <https://github.com/BennyQBD/3DGameProgrammingTutorial>

- **ColorChangeComponent:** Diese Komponente wird als Tag-Komponente verwendet. Das heißt, dass sie keinerlei Daten besitzt. Sie dient zur Kennzeichnung aller Entitäten, die die Farbe wechseln können.
- **MovementComponent:** Diese Komponente beinhaltet die für die Bewegung relevante Bewegungsgeschwindigkeit.
- **PredatorComponent:** Diese Komponente ist ebenfalls eine Tag-Komponente und kennzeichnet alle Entitäten, denen ausgewichen wird.
- **AvoidPredatorComponent:** Diese Komponente beinhaltet die Daten, die zur Vermeidung der Entität *Predator* verwendet werden.

Die Systeme

Die Systeme im ECS sind für das Verhalten der Entitäten zuständig. Als Superklasse wird die Klasse *System* erstellt. Diese stellt die virtuelle Funktion *Update* bereit, über die die Systeme aktualisiert werden. Systeme werden durch die Kombination der Komponenten, die einer Entität zugeordnet werden können, definiert. Die gesuchten Komponententypen werden im Array *componentTypes* über ihre *ID* gespeichert. Nur, wenn mindestens eine Entität die gesuchte Kombination von Komponenten besitzt, wird das dazugehörige System aktualisiert. Die folgenden Systeme werden für die Test-Applikation entwickelt:

- Das **RenderSystem** ist für das Rendering aller Entitäten zuständig, die die Komponenten *TransformComponent*, *RenderableComponent* und *ColorComponent* besitzen. In der Funktion *Update* werden die Daten dieser Komponenten sowie die fürs Rendering benötigten Matrizen an den verwendeten *Shader* weitergegeben, um das Modell zu zeichnen.
- Das **MovementSystem** ist für die Bewegung aller Entitäten zuständig, die die Komponenten *TransformComponent* und *MovementComponent* besitzen. Die Veränderung der Position wird innerhalb der Update-Funktion anhand der Bewegungsgeschwindigkeit und dem Zeit-Delta des letzten Frames vorgenommen. Außerdem ist dieses System für das Zurücksetzen an einen der Seitenränder mit einer zufälligen y-Position bei Erreichen des anderen Seitenrandes verantwortlich.
- Das **ColorChangeSystem** ist für die Veränderung der Farbe aller Entitäten zuständig, die die Komponenten *ColorComponent* und *ColorChangerComponent* besitzen.
- Das **CopyPredatorTransformsSystem** ist für die Aktualisierung der Position der Entität *Predator* zuständig. Dieses System sucht nach der Entität, die die Komponenten *TransformComponent* und *PredatorComponent* besitzt.

- Das **AvoidanceSystem** ist dafür zuständig, dass alle Entitäten, die die Komponenten *TransformComponent* und *AvoidPredatorComponent* besitzen, der Entität *Predator* ausweichen.

Um die unterschiedlichen Systeme besser verwalten und gruppieren zu können, wird die Klasse *SystemList* eingeführt. Diese Klasse besitzt eine Liste von Systemen. So können die Systeme zu verschiedenen Listen hinzugefügt werden, um in einer bestimmten Reihenfolge aktualisiert zu werden. Zum Beispiel ist es sinnvoll, dass das *RenderSystem* erst aktualisiert wird, nachdem die anderen Systeme aktualisiert worden sind. Daher wird es in der Klasse *Application* zur Liste *m_RenderingPipeline* hinzugefügt, während die anderen Systeme zur Liste *m_MainSystems* gehören.

Die Klasse ECS

Die Klasse *ECS* kann mit der Klasse *EntityManager* aus der Unity Engine verglichen werden. Sie liefert die Funktionalität, um Entitäten zu erstellen und zu zerstören, Komponenten hinzuzufügen und zu entfernen sowie Systeme zu aktualisieren. Sie verbindet die drei Bereiche des ECS miteinander. Diese Klasse besitzt außerdem Referenzen zu allen vorhandenen Komponenten über deren *ID* und Speicherort sowie zu allen Entitäten. Entitäten werden zwar nicht als eigener Datentyp angegeben, es muss jedoch bekannt sein, welche Komponenten zu welcher Entität gehören. Dies geschieht über das Array *entities*, in dem eine Entität als eine eindeutige *ID* mit einem Array aus den dazugehörigen Komponenten dargestellt wird.

Übersicht über das Entity Component System

Die Klasse *ECS* wird in der Klasse *Application* implementiert. Darin werden mit der Funktion *MakeEntity* Entitäten erstellt. Die Komponenten und Systeme werden ebenfalls direkt in der Klasse *Application* erstellt. Die benötigten Komponenten werden als Parameter beim Funktionsaufruf von *MakeEntity* mitgegeben. Die Systeme werden zu zwei Listen von Systemen, der Liste *m_MainSystems* und der Liste *m_RenderingPipeline*, hinzugefügt. Die Aktualisierung der Systeme findet innerhalb des *Game-Loops* über die Funktion *UpdateSystems*, die als Parameter eine *SystemList* erhält, statt. Bei einem umfangreicheren Projekt sollten diese Vorgänge weiter abstrahiert werden. Die Abbildung 3.9 bietet einen Überblick über das ECS in Form eines UML-Diagramms.

3. Konzeption

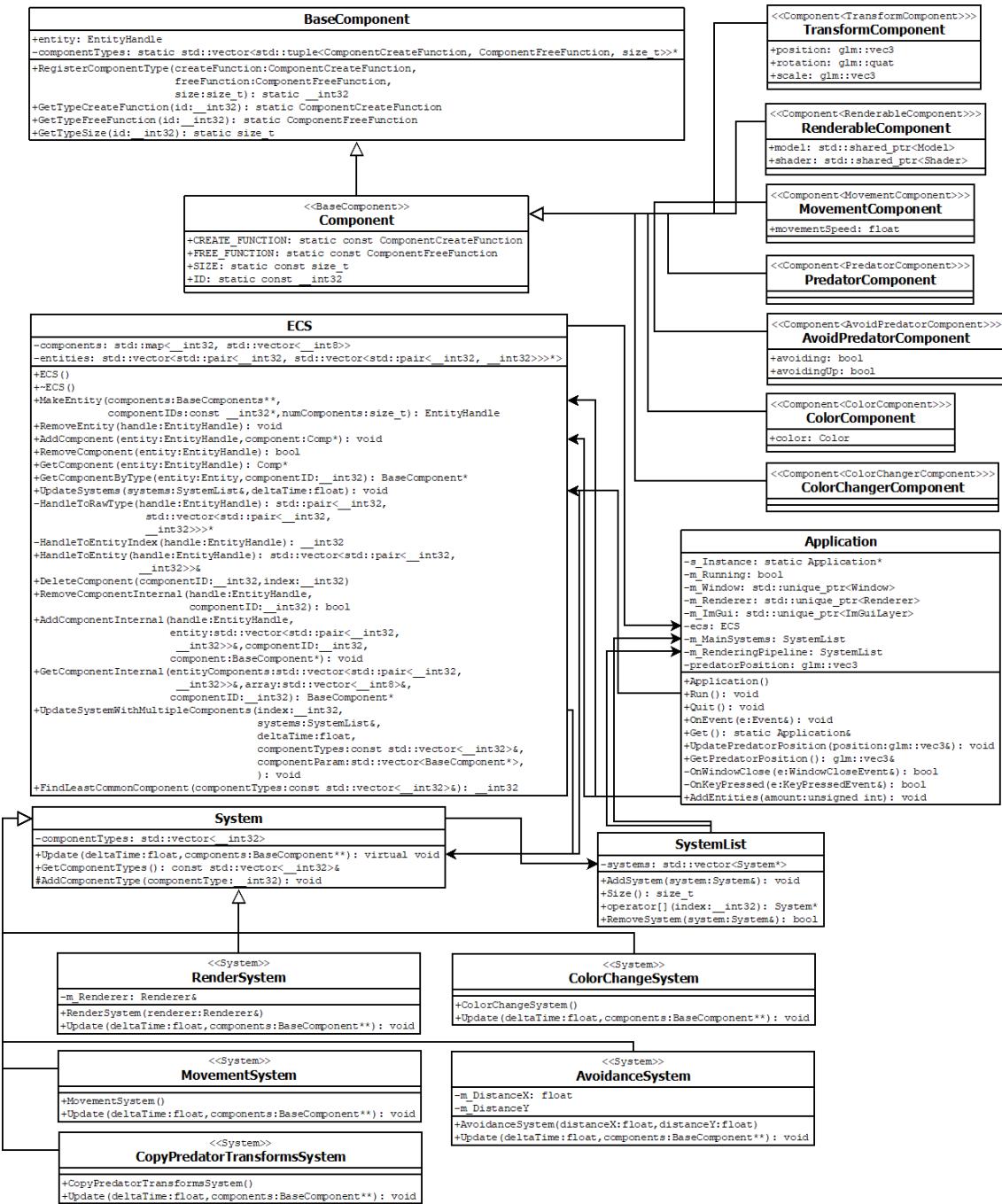


Abbildung 3.9: UML-Diagramm zum ECS in C++

4 Realisierung

Dieses Kapitel beschreibt die Realisierung der Test-Applikationen¹⁶ in der Unity Engine und in C++. Das Hauptaugenmerk liegt dabei auf der detaillierten Beschreibung der Besonderheiten bei der Realisierung einer EC und eines ECS. Außerdem werden die Bereiche der Entwicklung angesprochen, die Unterschiede zur Konzeption vorweisen.

4.1 Realisierung in Unity

Die Realisierung der beiden Test-Applikation in der Unity Engine weist keine großen Unterschiede zur Konzeption auf, da viele Aspekte von der Engine vorgegeben sind. Trotzdem gibt es einige interessante Bereiche der Realisierung, die in diesem Abschnitt thematisiert werden.

4.1.1 Mit der Entity Component Architecture

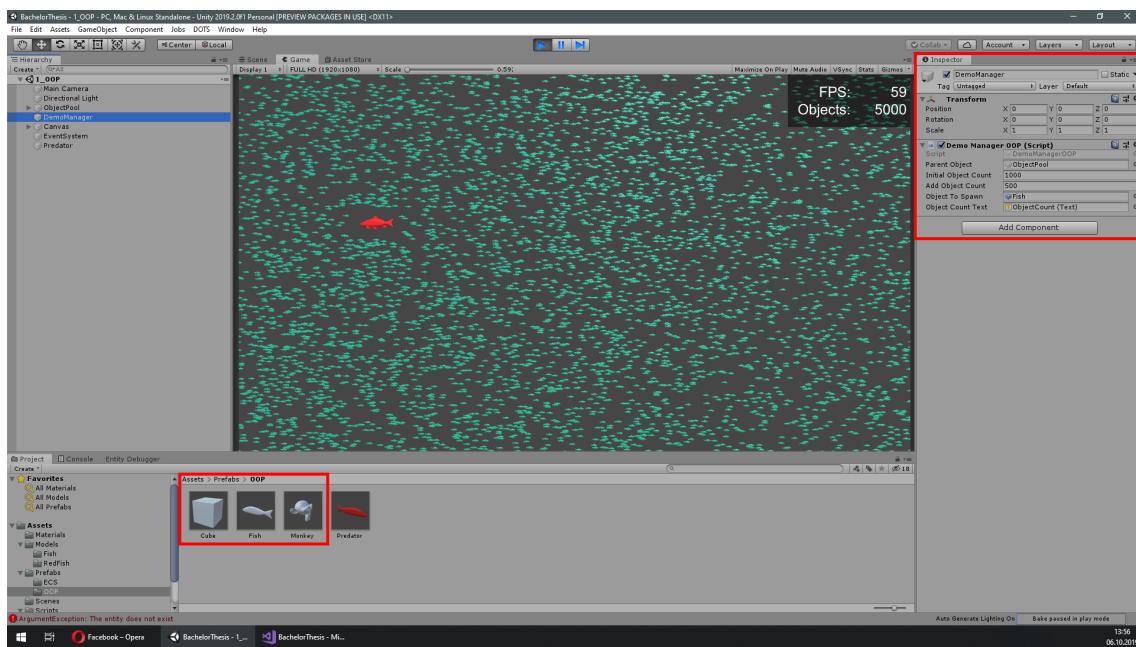


Abbildung 4.1: Die Test-Applikation mit der EC im Unity Editor

¹⁶Das Github-Repository mit allen Anwendungen: https://github.com/florianvoelkers/bachelor_thesis_applications

Der Ausgangspunkt der Anwendung ist das *GameObject DemoManager*, das als Komponente das Skript *DemoManagerOOP* besitzt. Bei Betätigung der Leertaste werden *GameObjects* in der Methode *SpawnObjects* instanziert. Im Inspektor, auf der rechten Seite der Abbildung 4.1, lassen sich einige Variablen einstellen. Die Variable *ObjectToSpawn* bestimmt, welches *Prefab* instanziert wird. Zur Auswahl stehen drei verschiedene *Prefabs*, die in der Abbildung 4.1 im *ProjectView* in der Mitte unten zu erkennen sind. Diese *Prefabs* unterscheiden sich lediglich durch das *Mesh*, das in der Komponente *MeshFilter* angegeben ist. Ansonsten besitzen sie die Komponenten, die in der Konzeption im Abschnitt 3.2.1 genannt werden.

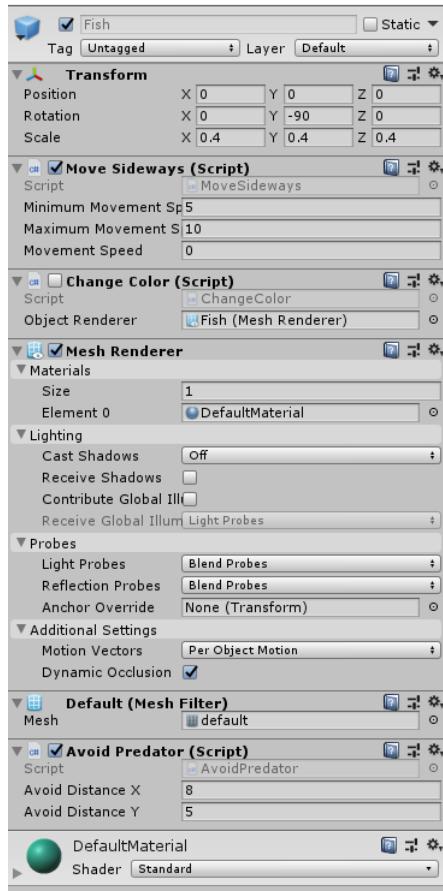


Abbildung 4.2: Die Komponenten des *Prefabs Fish* im Unity Inspector

Die Komponenten können im *Inspector* editiert werden, wenn das *Prefab* ausgewählt wird. So kann zum Beispiel eingestellt werden, welche Komponenten aktiv sein sollen. In Abbildung 4.2 ist die Komponente *ChangeColor* deaktiviert, weswegen die Entitäten nicht die Farbe wechseln. Die Realisierung der Komponenten *MoveSideways*, *AvoidPredator* und *ChangeColor* weist keine Unterschiede zur Konzeption auf. Eine genauere Betrachtung der Komponenten ist daher nicht notwendig.

4.1.2 Mit dem Entity Component System

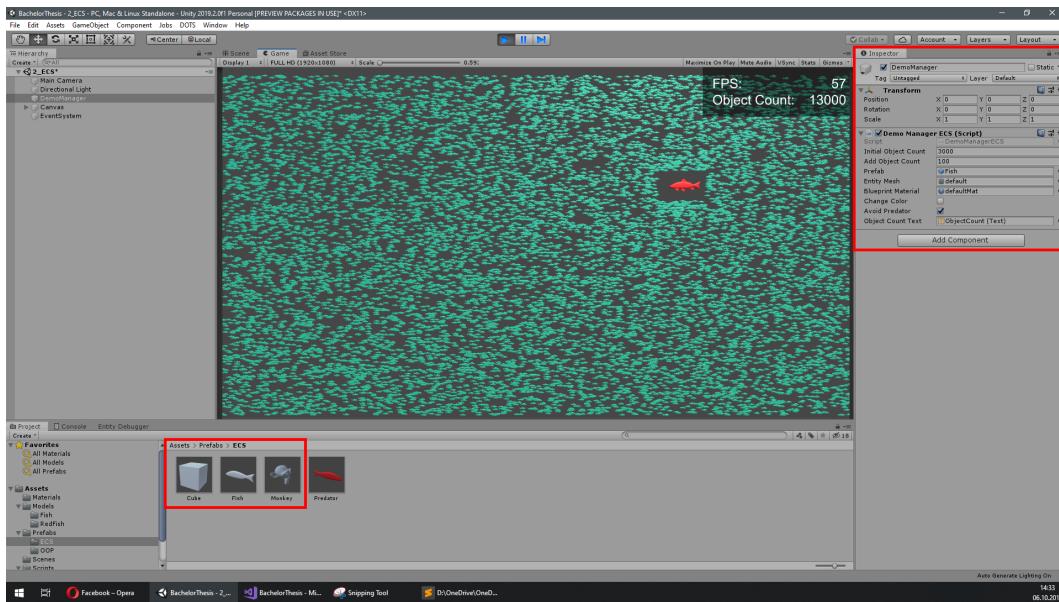


Abbildung 4.3: Die Test-Applikation mit dem ECS im Unity Editor

Der DemoManager

Im ECS ist ebenfalls das *GameObject DemoManager* Ausgangspunkt der Anwendung. Über die Klasse *DemoManagerECS* wird das *Prefab* in eine Entität umgewandelt. Sobald die Leertaste gedrückt wird, werden eine bestimmte Anzahl an Entitäten erstellt. Ihnen werden außerdem weitere Komponenten hinzugefügt: *MovementSpeed*, *AvoidPredatorComponent* und *ChangeColorComponent*. Über den Inspektor lässt sich einstellen, ob die Komponenten *AvoidPredatorComponent* und *ChangeColorComponent* hinzugefügt werden sollen. Wenn die Komponenten nicht vorhanden sind, werden die Systeme, die diese Komponenten verwenden, nicht aktiv sein. Die Klasse *DemoManagerECS* besitzt eine Referenz zur Entität *Predator*.

Instanzierung der Entitäten

Das Erstellen der Entitäten findet in drei Schritten statt. Zu Beginn wird über den Editor ein *GameObject* als *Prefab* erstellt. Dieses *GameObject* besitzt die von Unity bereitgestellten Komponenten *Transform*, *MeshFilter* und *MeshRenderer*. Zur Auswahl stehen drei unterschiedliche *Prefabs*, die sich, wie in der Anwendung mit der EC, lediglich durch das verwendete *Mesh* unterscheiden.

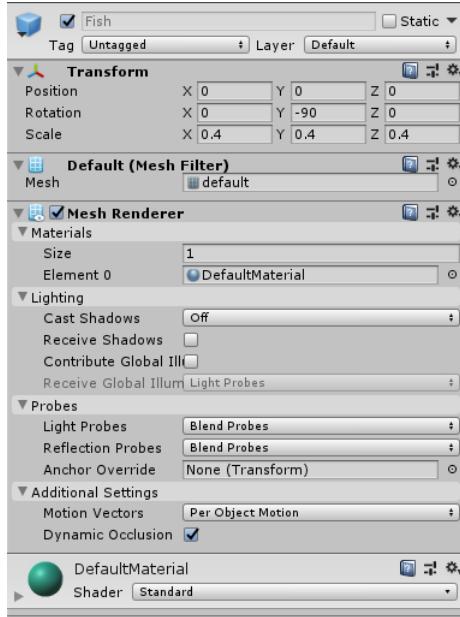


Abbildung 4.4: Die Komponenten des *Prefabs Fish* im Unity Inspector

Der zweite Schritt ist die Umwandlung dieses *GameObjects* in eine Entität, die *prefab* genannt wird. Dies findet im *DemoManager* mit der Funktion *ConvertGameObjectHierarchy* statt. Der dritte Schritt erfolgt, sobald die Leertaste betätigt wird. Aus der Vorlage *prefab* wird eine bestimmte Anzahl an Entitäten in der Funktion *SpawnObjects*, die in Abbildung 4.5 zu sehen ist, erstellt. Dafür wird die Funktion *Instantiate* des *EntityManagers* sowie ein *NativeArray* genutzt. Der Speicherplatz, der für die Instanziierung temporär genutzt wird, wird am Ende der Funktion über *Dispose* wieder freigegeben. Innerhalb der Schleife werden die fehlenden Komponenten zur Entität hinzugefügt und deren Werte gesetzt.

Eine Besonderheit bietet dabei die Komponente *RenderMesh*, die vom Typ *IShaderComponentData* ist. Komponenten dieses Typs können von mehreren Entitäten gleichzeitig verwendet werden. Dies ist ein riesiger Vorteil bei Entitäten, die das gleiche *Mesh*, *Material* und Schattenverhalten aufweisen. Für das Setzen einer zufälligen Startfarbe jeder einzelnen Entität, ist es jedoch von Nachteil, da sich alle Entitäten dieses Typs die Daten teilen. Verändert man also das Material einer Entität, verändert man es für alle Entitäten. Daher muss bei Aktivierung der Farbveränderung für jede Entität ein neues *RenderMesh* mit einem neuen *Material* erstellt werden. Als Alternative müsste ein eigenes *RenderSystem* implementiert werden, das nicht auf der Komponente *RenderMesh*, sondern auf eigenen Komponenten basiert.

4. Realisierung

```
64  private void SpawnObjects(int amount)
65  {
66      NativeArray<Entity> entities = new NativeArray<Entity>(amount, Allocator.Temp);
67      entityManager.Instantiate(prefab, entities);
68
69      for (int i = 0; i < amount; i++)
70      {
71          if (avoidPredator)
72          {
73              entityManager.AddComponent<AvoidPredatorComponent>(entities[i]);
74              entityManager.SetComponentData(entities[i], new AvoidPredatorComponent { avoiding = false, avoidingUp = false });
75          }
76
77          entityManager.AddComponent<MovementSpeed>(entities[i]);
78          entityManager.SetComponentData(entities[i], new MovementSpeed { Value = UnityEngine.Random.Range(2.0f, 10.0f) });
79
80          float spawnPositionY = UnityEngine.Random.Range(bottomBound, topBound + 1);
81          Vector3 position = transform.TransformPoint(new Vector3(leftBound, spawnPositionY, 0));
82          entityManager.SetComponentData(entities[i], new Translation { Value = position });
83
84          if (changeColor)
85          {
86              entityManager.AddComponent<ColorChangeComponent>(entities[i]);
87              Material randomMaterial = new Material(blueprintMaterial);
88              randomMaterialSetColor("_Color", UnityEngine.Random.ColorHSV());
89
90              entityManager.SetSharedComponentData(entities[i], new RenderMesh
91              {
92                  mesh = entityMesh,
93                  material = randomMaterial
94              });
95          }
96
97          objectCount++;
98      }
99
100     entities.Dispose();
101 }
102 }
```

Abbildung 4.5: Die Funktion *SpawnObjects* der Klasse *DemoManagerECS*

Der Predator

Eine weitere Entität, die für die Applikation verwendet wird, ist die Entität *Predator*. Diese wird mit einer anderen Methode als die übrigen Entitäten erstellt, damit die Daten im Inspektor editierbar sind. Es wird ein *GameObject* in der Szene platziert, welches über die von Unity bereitgestellte Komponente *ConvertToEntity* zu Beginn der Anwendung in eine Entität umgewandelt wird. Über die *MonoBehaviours* *MovementSpeedAuthoring* und *PredatorAuthoring* werden der Entität bei der Umwandlung außerdem die Komponenten *MovementSpeed* und *Predator* hinzugefügt. Über die Komponente *Predator*, die eine Tag-Komponente ist und keinerlei Daten besitzt, wird diese Entität im *DemoManager* identifiziert.

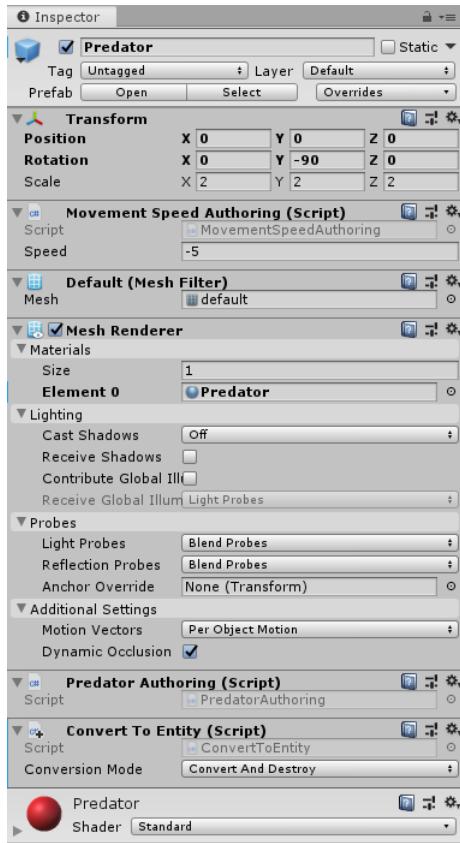


Abbildung 4.6: Die Komponenten des *GameObjects Predator* im Unity Inspector

Die Systeme

Neben dem von Unity bereitgestellten *RenderMeshSystemV2*, welches für das Rendering von *Meshes* zuständig ist, sind bis zu drei eigene Systeme in der Anwendung aktiv: das *MovementSystem*, das *AvoidanceSystem* und das *ColorChangeSystem*. Jedes System ist im ECS nur aktiv, wenn die Kombination von Komponenten, nach denen das System sucht, bei mindestens einer Entität vorhanden ist. Aufgrund der Einstellungen im *DemoManager* können das *AvoidanceSystem* und das *ColorChangeSystem* inaktiv sein. Um zu verstehen, wie die Systeme im ECS in Unity funktionieren, werden zwei Systeme vorgestellt. Diese verwenden unterschiedliche Funktionen, um die passenden Entitäten zu finden.

Das *MovementSystem*, dessen Realisierung in Abbildung 4.7 zu sehen ist, ist in der Anwendung immer aktiv, da alle Entitäten die Komponenten *Translation* und *MovementSpeed* besitzen. Es erbt von der Klasse *ComponentSystem* und überschreibt die Funktion *OnUpdate*, die zu jedem neuen Frame aufgerufen wird. In der Funktion wird zunächst ein sogenannter *EntityQueryBuilder* verwendet, der als Variable *Entities* in der Klasse *ComponentSystem* vorhanden ist. Dieser bietet viele verschiedene Möglich-

keit zum Finden von Entitäten, die eine bestimmte Kombination von Komponenten vorweisen. In diesem Fall wird die Funktion *ForEach* verwendet und als Parameter werden die beiden Komponenten mitgegeben, die bei einer Entität vorhanden sein müssen. Die Implementierung findet über einen Lambda-Ausdruck statt. Die Codezeilen 12 bis 39 aus der Abbildung 4.7 werden für jede Entität ausgeführt, die sowohl die Komponente *Translation* als auch die Komponente *MovementSpeed* besitzt. Darin wird der Wert der *Translation* in Abhängigkeit der Bewegungsgeschwindigkeit und der verstrichenen Zeit verändert. Außerdem wird geprüft, ob eine Entität den Rand des sichtbaren Bereichs erreicht hat. Sollte dies der Fall sein, wird sie an den anderen Rand zurückgesetzt.

```

6  public class MovementSystem : ComponentSystem
7  {
8      protected override void OnUpdate()
9      {
10         Entities.ForEach((ref Translation translation, ref MovementSpeed movementSpeed) =>
11         {
12             if (movementSpeed.Value > 0.0f)
13             {
14                 if (translation.Value.x < DemoManagerECS.rightBound)
15                 {
16                     float spawnPositionY = UnityEngine.Random.Range(DemoManagerECS.bottomBound, DemoManagerECS.topBound + 1);
17                     translation.Value.x += movementSpeed.Value * Time.deltaTime;
18                 }
19                 else
20                 {
21                     float spawnPositionY = UnityEngine.Random.Range(DemoManagerECS.bottomBound, DemoManagerECS.topBound + 1);
22                     translation.Value.x = DemoManagerECS.leftBound;
23                     translation.Value.y = spawnPositionY;
24                 }
25             }
26             else
27             {
28                 if (translation.Value.x > DemoManagerECS.leftBound)
29                 {
30                     translation.Value.x -= movementSpeed.Value * Time.deltaTime;
31                 }
32                 else
33                 {
34                     float spawnPositionY = UnityEngine.Random.Range(DemoManagerECS.bottomBound, DemoManagerECS.topBound + 1);
35                     translation.Value.x = DemoManagerECS.rightBound;
36                     translation.Value.y = spawnPositionY;
37                 }
38             }
39         });
40     }
41 }
42 
```

Abbildung 4.7: Die Klasse *MovementSystem* implementiert die Bewegung von Entitäten

Das *ColorChangeSystem*, dessen Realisierung in Abbildung 4.8 zu sehen ist, nutzt eine andere Funktion des *EntityQueryBuilder*. Die Funktion *ForEach* konnte in diesem Fall nicht verwendet werden, da die Komponente *RenderMesh* als Parameter unzulässig ist. Mit der Funktion *WithAll* kann jedoch erfolgreich nach den Entitäten gefiltert werden, die sowohl die Komponente *RenderMesh* als auch die Komponente *ColorChangeComponent* besitzen. Anschließend kann die Farbe des Materials über die Referenz der Komponente *RenderMesh* im *EntityManager* verändert werden.

```

7  public class ColorChangeSystem : ComponentSystem
8  {
9      protected override void OnUpdate()
10     {
11         float deltaTime = Time.deltaTime;
12
13         Entities.WithAll<RenderMesh, ColorChangeComponent>().ForEach((Entity e) =>
14         {
15             RenderMesh renderMesh = EntityManager.GetSharedComponentData<RenderMesh>(e);
16
17             Color oldColor = renderMesh.material.GetColor("_Color");
18             float newR = oldColor.r + deltaTime;
19             float newG = oldColor.g + deltaTime;
20             float newB = oldColor.b + deltaTime;
21             if (newR > 1.0f)
22                 newR = 0.0f;
23             if (newG > 1.0f)
24                 newG = 0.0f;
25             if (newB > 1.0f)
26                 newB = 0.0f;
27
28             renderMesh.material.SetColor("_Color", new Color(newR, newG, newB, oldColor.a));
29         });
30     }
31 }
```

Abbildung 4.8: Die Klasse *ColorChangeSystem* dient zur Farbänderung des Materials

4.2 Realisierung in C++

In diesem Abschnitt wird die Realisierung einer EC und eines ECS in C++ beschrieben. Besonders interessant ist dabei, dass im Gegensatz zur Unity Engine kein Framework zur Verfügung steht und somit die darin gegebene Funktionalität selbst in C++ erstellt werden muss. Aus diesem Grund werden die eigens für die EC und das ECS kreierten Frameworks im nachfolgenden Abschnitt detailliert beschrieben. Neben der verwendeten Lösung gibt es jedoch weitere Möglichkeiten eine EC oder ein ECS umzusetzen. Die Realisierung der grundlegenden Funktionalität, die im Abschnitt 3.3.2 beschrieben wird, weist aufgrund von Erfahrungen aus einem vorherigen Projekt keine Unterschiede zur Konzeption auf und wird daher nicht näher beschrieben. Außerdem ist sie für die Thematik der Thesis irrelevant.

4.2.1 Mit der Entity Component Architecture

Für die Realisierung der EC wird eine ähnliche Vorgehensweise wie in der Unity Engine verwendet. Als Grundlage wird die Lösung des Nutzers *Tom_*¹⁷ aus einer Frage bei *Stackoverflow* genutzt¹⁷. Darin wird beschrieben, wie man Komponenten entwirft, deren Typ zur Laufzeit bestimmt werden kann. Diese Eigenschaft wird Runtime Type Information (RTTI) genannt. Für die Implementierung werden Makros und Templates genutzt. Als Grundlage dienen die Klasse *Component*, von der alle Komponenten erben, und die Klasse *Entity*, die eine Entität im Spiel repräsentiert.

¹⁷Zu finden unter: <https://stackoverflow.com/questions/44105058/how-does-unitys-getcomponent-work>

Komponenten

Die Klasse *Component* dient als Superklasse für alle Komponenten. Darin werden die virtuellen Funktionen *OnUpdate*, *OnEvent* und *Draw* definiert. Außerdem erhält sie eine Referenz zu der Entität, die sie besitzt, und einen booleschen Ausdruck, der anzeigt, ob sie aktiv oder inaktiv ist. Die Identifizierung des Komponententyps findet über die statische Variable *Type* statt. Die Besonderheit der Klasse *Component* sind die beiden Makros, die verwendet werden, damit die Komponenten per RTTI identifiziert werden können. Makros sind Anweisungen, die mit einem einfachen Aufruf ausgeführt werden können. Sie werden genutzt, um Code abzukürzen und damit leserlicher zu gestalten. Das Makro *CLASS_DECLARATION* muss in der Klassendeklaration jeder Komponente vorhanden sein, um die Variable *Type* und die virtuelle Funktion *IsClassType* zu überschreiben. Dafür wird das Makro mit dem Klassennamen als Parameter aufgerufen.

```
28     #define CLASS_DECLARATION( classname ) \
29     public: \
30         static const std::size_t Type; \
31         virtual bool IsClassType( const std::size_t classType ) const override; \
32 
```

Abbildung 4.9: Das Makro CLASS_DECLARATION

In der Definition der Klasse der Komponenten muss außerdem das Makro *CLASS_DEFINITION* vorhanden sein, welches den Namen der Superklasse und den Namen der Klasse als Parameter angibt. Mit diesem Makro werden die Variablen zur Überprüfung des Komponententyps initialisiert. Die Variable *Type* ist ein Hash des Klassennamens. Die Funktion *IsClassType* gibt den booleschen Wert *true* zurück, wenn der mitgegebene Parameter mit dem Typ der Klasse übereinstimmt. Wenn dies nicht der Fall ist, wird geprüft, ob der Typ der Superklasse mit dem gesuchten Typ übereinstimmt.

```
38     #define CLASS_DEFINITION( parentclass, childclass ) \
39     const std::size_t childclass::Type = std::hash< std::string >()( TO_STRING( childclass ) ); \
40     bool childclass::IsClassType( const std::size_t classType ) const { \
41         if ( classType == childclass::Type ) \
42             return true; \
43         return parentclass::IsClassType( classType ); \
44     } 
```

Abbildung 4.10: Das Makro CLASS_DEFINITION

Die Definition der Klassen der Komponenten findet in der Datei *ComponentClasses.cpp* statt. Die Klassen könnten auch in eigenen Dateien, passend zu den Komponenten, oder in anderen *.cpp* Dateien definiert werden, jedoch hat die Implementierung über die Datei *ComponentClasses.cpp* den Vorteil alle Definitionen an einem Ort zu bündeln.

```

1  #include "Component.hpp"
2
3  // include all components here
4  #include "Renderable.hpp"
5  #include "Transform.hpp"
6  #include "Movement.hpp"
7  #include "ColorComponent.hpp"
8  #include "Predator.hpp"
9  #include "AvoidPredator.hpp"
10
11 const std::size_t Component::Type = std::hash<std::string>()(TO_STRING(Component));
12
13 // All class definitions for components
14 // CLASS_DEFINITION(parent class, sub class)
15 CLASS_DEFINITION(Component, Transform)
16 CLASS_DEFINITION(Component, Renderable)
17 CLASS_DEFINITION(Component, Movement)
18 CLASS_DEFINITION(Component, ColorComponent)
19 CLASS_DEFINITION(Component, Predator)
20 CLASS_DEFINITION(Component, AvoidPredator)

```

Abbildung 4.11: Die Datei *ComponentClasses.cpp*

Die Realisierung der einzelnen Komponenten weist keinen Unterschied zu der Konzeption aus Abschnitt 3.3.3 auf. Die Komponenten erben von der Klasse *Component*, verwenden das Makro *CLASS_DECLARATION* und überschreiben eine oder mehrere virtuelle Methoden der Klasse *Component*.

Entitäten

Die Entitäten werden in der Klasse *Entity* definiert. Das Besondere an der Realisierung ist die Nutzung von Funktions-Templates für die Funktionen *AddComponent*, *RemoveComponent* und *GetComponent*. Dies ermöglicht die in Abbildung 4.12 dargestellte Syntax zum Hinzufügen, Entfernen oder Erhalten bestimmter Komponenten.

```

std::shared_ptr<Entity> entity = std::make_shared<Entity>("Entity", 0);
entity->AddComponent<Transform>(entity.get(), glm::vec3(0.0f), glm::quat(0.0f, 0.0f, 0.0f, 1.0f), glm::vec3(1.0f));
Transform& transform = entity->GetComponent<Transform>();
entity->RemoveComponent<Transform>();

```

Abbildung 4.12: So können Komponenten hinzugefügt, entfernt und erhalten werden

Die Funktionen werden in der Datei *Entity.hpp* definiert. Die Klasse der Komponenten wird in den spitzen Klammern angegeben und innerhalb der Funktionen als *ComponentType* referenziert. Bei der Funktion *AddComponent* gibt es außerdem die Möglichkeit über den Parameter *Args* eine beliebige Anzahl an Parametern beim Funktionsaufruf hinzuzufügen. Diese werden anschließend mit der Funktion *forward* aus der Bibliothek *std* an den Konstruktor der Komponente weitergeleitet.

```

86     template<class ComponentType, typename ...Args>
87     inline void Entity::AddComponent(Args&& ...params)
88     {
89         m_Components.emplace_back(new ComponentType(std::forward<Args>(params)...));
90     }
91
92     template< class ComponentType >
93     bool Entity::RemoveComponent() {
94         if (m_Components.empty())
95             return false;
96
97         auto& index = std::find_if(m_Components.begin(),
98                                     m_Components.end(),
99                                     [classType = ComponentType::Type](auto& component) {
100                                         return component->IsClassType(classType);
101                                     });
102
103         bool success = index != m_Components.end();
104
105         if (success)
106             m_Components.erase(index);
107
108         return success;
109     }
110
111     template<class ComponentType>
112     inline ComponentType& Entity::GetComponent()
113     {
114         for (auto&& component : m_Components)
115         {
116             if (component->IsClassType(ComponentType::Type))
117                 return *static_cast<ComponentType*>(component.get());
118         }
119
120         return *std::unique_ptr< ComponentType >(nullptr);
121     }

```

Abbildung 4.13: Realisierung der Funktionen *AddComponent*, *RemoveComponent* und *GetComponent*

Des Weiteren sind die Entitäten hierarchisch aufgebaut. Die Klasse *Application* besitzt eine Referenz zur Entität *root*, der alle weiteren Entitäten untergeordnet sind. Die Realisierung der anderen Bereiche der Applikation unterscheidet sich nicht von der Konzeption und ist für diese Arbeit von geringer Bedeutung.

4.2.2 Mit dem Entity Component System

Für die Realisierung eines ECS in C++ existieren verschiedene Lösungswege. Die Lösung, die in dieser Arbeit verwendet wird, ist eine Adaption der Lösung des Github-Nutzers *BennyQBD*¹⁸. Für diese Lösung werden an vielen Stellen Funktions- und Strukturen-Templates verwendet. Da eine Beschreibung aller Funktionen, die für die Realisierung des ECS wichtig sind, zu umfangreich wäre, beschränkt sich der folgende

¹⁸Link zum Github-Repository: <https://github.com/BennyQBD/3DGameProgrammingTutorial>

Abschnitt auf die wichtigsten Aspekte der Realisierung der Komponenten und Systeme sowie die interessantesten Funktionen der Klasse *ECS*, die die Komponenten und Systeme miteinander verknüpft.

Die Komponenten

In der Struktur *BaseComponent* werden die statischen Funktionen und Variablen deklariert, die für alle Komponenten eines bestimmten Typs gleich sind. Es handelt sich dabei um typspezifische Funktionen zum Erstellen und Löschen von Komponenten sowie um die Größe der Komponente. Diese Informationen definieren einen Komponententyp. Alle unterschiedlichen Komponententypen werden in dem statischen Array *componentTypes* der Struktur *BaseComponent* gespeichert. Die *ComponentCreateFunction* ist eine Funktion zum Erstellen von Komponenten. Sie erhält als Parameter die Speicheradresse des Komponententyps, die Entität sowie eine Referenz zur *BaseComponent* und gibt den Index der erstellten Komponente im Array dieses Komponententyps zurück. Die *ComponentFreeFunction* nimmt als Parameter eine Referenz zu der *BaseComponent*, die gelöscht werden soll, entgegen und ruft explizit den Dekonstruktor der Komponente auf, um den Speicherplatz wieder freizugeben. Die Funktionen werden in der Datei *Component.hpp* als Funktions-Templates definiert. Sie werden automatisch für jeden Komponententyp erstellt. Welche Funktion letztendlich aufgerufen wird hängt vom Typ der Komponente ab, die erstellt oder gelöscht werden soll. Diese wird im Funktions-Template als *typename Comp* referenziert.

```

14     template<typename Comp>
15     int32 ComponentCreate(std::vector<__int8>& memory, EntityHandle entity, BaseComponent* baseComponent)
16     {
17         __int32 index = memory.size();
18         memory.resize(index + Comp::SIZE);
19         Comp* component = new(&memory[index])Comp(*((Comp*)baseComponent));
20         component->entity = entity;
21         return index;
22     }
23
24     template<typename Comp>
25     void ComponentFree(BaseComponent* baseComponent)
26     {
27         Comp* component = (Comp*)baseComponent;
28         component->~Comp();
29     }

```

Abbildung 4.14: Definition der Funktionen *ComponentCreate* und *ComponentFree*

Die Initialisierung der statischen Variablen eines Komponententyps findet automatisch beim Aufruf des Konstruktors statt. Über die Funktions-Templates in der Datei *Component.hpp*, werden die Variablen für *ID*, *SIZE*, *CREATE_FUNCTION* und *FREE_FUNCTION* initialisiert.

```

31     template<typename T>
32     const __int32 Component<T>::ID(BaseComponent::RegisterComponentType(ComponentCreate<T>, ComponentFree<T>, sizeof(T)));
33
34     template<typename T>
35     const size_t Component<T>::SIZE(sizeof(T));
36
37     template<typename T>
38     const ComponentCreateFunction Component<T>::CREATE_FUNCTION(ComponentCreate<T>);
39
40     template<typename T>
41     const ComponentFreeFunction Component<T>::FREE_FUNCTION(ComponentFree<T>);
42

```

Abbildung 4.15: Initialisierung der Variablen der Komponenten

Bei der Definition der einzelnen Komponenten, die von der Struktur *Component* erben, muss zur korrekten Initialisierung lediglich der Komponententyp als Template-Parameter mitgegeben. Die Realisierung der einzelnen Komponenten weist keine Unterschiede zur Konzeption aus Abschnitt 3.3.4 vor. Als Beispiel für die Realisierung einer Komponente ist in Abbildung 4.16 die *TransformComponent* zu sehen.

```

7     struct TransformComponent : public Component<TransformComponent>
8     {
9         glm::vec3 position;
10        glm::quat rotation;
11        glm::vec3 scale;
12    };

```

Abbildung 4.16: Realisierung der *TransformComponent*

Die Systeme

Die Realisierung der Systeme entspricht der Konzeption in Abschnitt 3.3.4. Als Superklasse fungiert die Klasse *System*, die die virtuelle Funktion *Update* definiert. Außerdem besitzt jedes System das Array *componentTypes*, das aus den *IDs* der Komponenten besteht, die das System benötigt. Dieses Array wird mit dem Aufruf des Konstruktors über die Funktion *AddComponentType* gefüllt. Nur wenn eine Entität alle Komponententypen besitzt, die in diesem Array vermerkt sind, wirkt das System auf diese Entität. Die Logik jedes Systems wird in der Funktion *Update* implementiert, die als Parameter die seit dem letzten *Update* vergangene Zeit *deltaTime* und ein Array von *Pointern* zu den Komponenten erhält. Dieses Array wird von der Klasse *ECS* bereitgestellt und enthält nur die Komponenten, die vom System benötigt werden. Die Reihenfolge der Komponenten in diesem Array entspricht der Reihenfolge der Komponenten im Array *componentTypes* wodurch sie über ihre Indizes eindeutig referenzierbar sind. Als Beispiel für die Realisierung eines Systems wird in Abbildung 4.17 das *RenderSystem* vorgestellt, das auf alle Entitäten wirkt, denen die Komponenten *TransformComponent*, *RenderableComponent* und *ColorComponent* zugeordnet werden können.

```

18  class RenderSystem : public System
19  {
20  public:
21      RenderSystem(Renderer& renderer) : System(), m_Renderer(renderer)
22      {
23          AddComponentType(TransformComponent::ID);
24          AddComponentType(RenderableComponent::ID);
25          AddComponentType(ColorComponent::ID);
26      }
27
28      virtual void Update(float deltaTime, BaseComponent** components) override
29      {
30          TransformComponent* transform = (TransformComponent*)components[0];
31          RenderableComponent* renderable = (RenderableComponent*)components[1];
32          ColorComponent* color = (ColorComponent*)components[2];
33
34          glm::mat4 translationMatrix = glm::mat4();
35          translationMatrix = glm::translate(translationMatrix, transform->position);
36          glm::mat4 rotationMatrix = glm::mat4();
37          rotationMatrix = glm::toMat4(transform->rotation);
38          glm::mat4 scaleMatrix = glm::mat4();
39          scaleMatrix = glm::scale(scaleMatrix, transform->scale);
40          glm::mat4 transformationMatrix = translationMatrix * rotationMatrix * scaleMatrix;
41
42          renderable->shader->Use();
43          renderable->shader->SetMat4("model", transformationMatrix);
44          renderable->shader->SetMat4("view", m_Renderer.GetViewMatrix());
45          renderable->shader->SetMat4("projection", m_Renderer.GetProjectionMatrix());
46          renderable->shader->SetVec4("color", color->color.ToVector4f());
47          renderable->model->Draw(*renderable->shader);
48      }
49
50  private:
51      Renderer& m_Renderer;
52 };

```

Abbildung 4.17: Realisierung des *RenderSystems*

Die Systeme werden in der Klasse *Application* erstellt und einer von zwei System-Listen zugeordnet. Bis auf das *RenderSystem* werden alle Systeme zur Liste *m_MainSystems* hinzugefügt. Da das *RenderSystem* jedoch erst als letztes aktualisiert werden darf, wird es zu einer separaten Liste *m_Rendering-Pipeline* hinzugefügt, welche nach der Liste *m_MainSystems* aktualisiert wird.

Die Klasse ECS

Die Aktualisierung der System-Listen findet über die Funktion *UpdateSystems* der Klasse *ECS* statt. Diese Klasse fungiert als Manager zwischen den Entitäten, Komponenten und Systemen, indem sie Funktionalität zum Erstellen von Entitäten sowie zum Hinzufügen, Entfernen und Erhalten von Komponenten bietet. An dieser Stelle werden ebenfalls Funktions-Templates zur einfacheren Verwendung eingesetzt. Für die Funktion *MakeEntity* zum Erstellen von Entitäten werden verschiedene Funktions-Templates zur Verfügung gestellt, die eine bis sieben Komponenten als Template-Parameter erhalten können. Außerdem ist es jederzeit möglich über das Funktions-Template *AddComponent* weitere Komponenten zu einer Entität hinzuge-

fügt oder über das Funktions-Template *RemoveComponent* entfernt werden.

Die Klasse *ECS* verwaltet außerdem eine Map mit allen Komponenten. Als Schlüssel wird der Komponententyp, repräsentiert durch die *ID*, verwendet. Der Wert ist der Speicherbereich dieses Komponententyps. Diese Map wird durchsucht, wenn die Systeme aktualisiert werden sollen. So kann eine bestimmte Komponente schnell gefunden und zum Array der Komponenten, die ein System zur Aktualisierung benötigt, hinzugefügt werden. Die Aktualisierung der Systeme findet in der Funktion *UpdateSystems* der Klasse *ECS* statt. Darin wird für alle Systeme zunächst geprüft, ob mehr als eine Komponente benötigt wird. Wenn nur eine Komponente benötigt wird, ist dies ein Sonderfall, der optimiert wird, indem die Liste der Komponenten direkt durchsucht wird. Bei mehreren Komponenten, was in der Applikation auf alle Systeme zutrifft, wird die Funktion *UpdateSystemWithMultipleComponents* aufgerufen, die in Abbildung 4.18 dargestellt wird.

```

141 void ECS::UpdateSystemWithMultipleComponents(__int32 index, SystemList& systems, float deltaTime,
142     const std::vector<__int32>& componentTypes, std::vector<BaseComponent*>& componentParam, std::vector<std::vector<__int8*>>& componentArrays)
143 {
144     componentParam.resize(std::max(componentParam.size(), componentTypes.size()));
145     componentArrays.resize(std::max(componentArrays.size(), componentTypes.size()));
146     for (__int32 i = 0; i < componentTypes.size(); i++)
147         componentArrays[i] = &components[componentTypes[i]];
148
149     __int32 minSizeIndex = FindLeastCommonComponent(componentTypes);
150
151     size_t typeSize = BaseComponent::GetTypeSize(componentTypes[minSizeIndex]);
152     std::vector<__int8*> array = *componentArrays[minSizeIndex];
153     for (__int32 i = 0; i < array.size(); i += typeSize)
154     {
155         componentParam[minSizeIndex] = (BaseComponent*) & array[i];
156         std::vector<std::pair<__int32, __int32>>& entityComponents = HandleToEntity(componentParam[minSizeIndex]->entity);
157
158         bool isValid = true;
159         for (__int32 j = 0; j < componentTypes.size(); j++)
160         {
161             if (j == minSizeIndex)
162                 continue;
163
164             componentParam[j] = GetComponentInternal(entityComponents, *componentArrays[j], componentTypes[j]);
165             if (componentParam[j] == nullptr)
166             {
167                 isValid = false;
168                 break;
169             }
170         }
171
172         if (isValid)
173             systems[index]->Update(deltaTime, &componentParam[0]);
174     }
175 }
```

Abbildung 4.18: Aktualisierung der Systeme

Darin werden zunächst das Array *componentParam* mit allen Komponenten, die das System verlangt, und das Array *componentArrays* mit allen unterschiedlichen Komponententypen, auf die benötigte Größe gebracht. Anschließend wird über die Funktion *FindLeastCommonComponent* die am seltensten vorkommende Komponente gesucht. Dadurch entstehen weniger *cache misses*, weil zuerst diese Komponente geprüft wird. Wenn eine vom System verlangte Komponente nicht existiert, wird das System für diese Entität nicht aktualisiert und es werden die Komponenten der nächsten Entität überprüft. Sind alle gesuchten Komponenten vorhanden, werden die Komponenten an das System weitergeleitet und das System wird aktualisiert. Die Überprüfung der

Komponenten findet in der Funktion *GetComponentInternal* statt, die in der Abbildung 4.19 dargestellt wird.

```

132     BaseComponent* ECS::GetComponentInternal(std::vector<std::pair<__int32, __int32>>& entityComponents, std::vector<__int8>& array, __int32 componentID)
133     {
134         for (__int32 i = 0; i < entityComponents.size(); i++)
135             if (componentID == entityComponents[i].first)
136                 return (BaseComponent*) & array[entityComponents[i].second];
137
138         return nullptr;
139     }

```

Abbildung 4.19: Mit der Funktion *GetComponentInternal* wird überprüft, ob eine Entität eine bestimmte Komponente besitzt

Die Funktion *GetComponentInternal* erhält als Parameter die Referenz zum Array der Komponenten einer bestimmten Entität, eine Referenz zu dem Speicherbereich des Komponententyps und die *ID* der Komponente. Sollte die *ID* der Komponente mit einer der *IDs* der Komponenten einer Entität übereinstimmen, wird ein Pointer zu dieser Komponente zurückgegeben.

Die Entitäten werden im ECS nicht durch eine Klasse repräsentiert, sondern existieren lediglich als Array innerhalb der Klasse *ECS*. Das Array besteht aus Paaren. Der erste Wert ist die einzigartige *ID* der Entität. Der zweite Wert ist ein Array mit allen Komponenten dieser Entität. Komponenten werden wiederum als Paare dargestellt, bei denen der erste Wert den Komponententyp über die *ID* angibt und der zweite Wert den Index der Komponente im Array des Komponententyps. Um eine Entität zu erstellen, wird die Funktion *MakeEntity* genutzt, die in Abbildung 4.20 zu sehen ist. Darin wird eine neue Entität kreiert und ihr werden über die Funktion *AddComponentInternal* nacheinander die gewünschten Komponenten hinzugefügt.

```

21     EntityHandle ECS::MakeEntity(BaseComponent** entityComponents, const __int32* componentIDs, size_t numComponents)
22     {
23         std::pair<__int32, std::vector<std::pair<__int32, __int32>>>* newEntity = new std::pair<__int32, std::vector<std::pair<__int32, __int32>>>();
24         EntityHandle handle = (EntityHandle) newEntity;
25         for (__int32 i = 0; i < numComponents; i++)
26             AddComponentInternal(handle, newEntity->second, componentIDs[i], entityComponents[i]);
27
28         newEntity->first = entities.size();
29         entities.push_back(newEntity);
30
31         return handle;
32     }

```

Abbildung 4.20: Mit der Funktion *MakeEntity* werden neue Entitäten und ihre Komponenten erstellt

Die Funktion *AddComponentInternal* ruft die *ComponentCreateFunction* des gewünschten Komponententyps auf und fügt die Komponente zur Entität hinzu.

5 Auswertung

In diesem Kapitel wird zunächst der Ablauf der Tests beschrieben, die zur Bewertung der Performance der EC und des ECS dienen. Anschließend werden die Testergebnisse präsentiert und die Implementierungen sowohl in Unity als auch in C++ evaluiert. Zum Abschluss werden außerdem die Testergebnisse der Unity-Applikationen mit denen der Applikationen in C++ verglichen.

5.1 Ablauf der Tests

Alle Applikationen werden unter den gleichen Bedingungen mit der gleichen Hardware¹⁹ getestet. Für die Tests werden die Build-Versionen der Anwendungen verwendet, da zum Beispiel im Editor der Unity Engine einige Debugging-Tools die Framerate beeinflussen. Das Ziel der Tests ist es die Performance der Anwendungen anhand der Framerate und der Anzahl der aktiven Entitäten zu bewerten. Dazu werden sukzessive Entitäten instanziert bis eine Framerate von 30 FPS erreicht wird. Bei den Tests werden zwei verschiedene Parameter verändert. Der erste Parameter ist die Wahl des Modells. Dafür stehen drei unterschiedlich komplexe Modelle zur Verfügung: der Würfel, der Fisch und der Affenkopf. Der zweite Parameter, der verändert werden kann, sind die Komponenten, die die Entitäten besitzen. Für diesen Parameter gibt es insgesamt vier Variationen:

1. Nur die Komponente *Movement* (M) wird hinzugefügt.
2. Die Komponenten *Movement* (M) und *AvoidPredator* (A) werden hinzugefügt.
3. Die Komponenten *Movement* (M) und *ChangeColor* (C) werden hinzugefügt.
4. Die Komponenten *Movement* (M), *AvoidPredator* (A) und *ChangeColor* (C) werden hinzugefügt.

5.2 Realisierung in Unity

In diesem Abschnitt werden zunächst die Testergebnisse für die EC und das ECS beschrieben und interpretiert. Anschließend werden die Ergebnisse miteinander verglichen.

¹⁹Prozessor: Intel Core i5-7600 @3,5 GHz, Arbeitsspeicher: 16 GB, Grafikkarte: GeForce GTX 1060 6GB

5.2.1 Evaluation der Entity Component Architecture

Modell	M	M + A	M + C	M + A + C
Würfel	19000	16000	4000	3800
Fisch	19000	16000	4000	3800
Affenkopf	3500	3500	3500	3500

Tabelle 5.1: Anzahl der instanzierten Entitäten bis 30 FPS erreicht sind in den Anwendungen mit der EC in Unity

Die Ergebnisse für die Modelle Würfel und Fisch sind exakt gleich, obwohl der Fisch mit 672 Dreiecken um ein Vielfaches komplexer ist als der Würfel mit 12 Dreiecken. Jedoch wird bei den Ergebnissen mit dem Modell Affenkopf, das aus 15744 Dreiecken besteht, deutlich, dass ab einer bestimmten Komplexität des Modells die Performance stark beeinflusst wird. Die Ergebnisse zeigen, dass bei relativ einfachen Modellen die Anzahl der Entitäten der limitierende Faktor ist. Dies zeigt, dass die Kapazität der CPU erschöpft ist. Bei komplexen Modellen scheint jedoch das Rendering und damit die Leistung der GPU zum limitierenden Faktor zu werden.

Eine weitere Erkenntnis aus den Ergebnissen ist, dass jede Komponente, die zu einem *GameObject* hinzugefügt wird, einen Einfluss auf die Performance hat. Dies lässt sich daran erkennen, dass schon beim Hinzufügen der Komponente *AvoidPredator* die Anzahl der Entitäten, die instanziert werden können, sinkt. Besonders deutlich ist die Verschlechterung der Performance beim Hinzufügen der Komponente *ChangeColor* zu sehen. Durch diese Komponente wird die Farbe des Materials in der Unity Engine verändert. Die Verschlechterung der Performance ist überraschend groß und lässt sich wohl nur dadurch erklären, dass durch die Farbveränderung Optimierungen im Bereich des Renderings, die Unity automatisch vornimmt, nicht genutzt werden können.

Es kann davon ausgegangen werden, dass die Anzahl der instanzierten Entitäten bei der Verwendung von mehreren und deutlich komplexeren Komponenten stark abnimmt. Um dies zu verhindern, soll die Verwendung des ECS in Verbindungen mit dem *Data-Oriented Technology Stack* von Unity helfen.

5.2.2 Evaluation des Entity Component Systems

Modell	M	M + A	M + C	M + A + C
Würfel	50000	50000	3100	3100
Fisch	50500	50500	3100	3100
Affenkopf	3500	3100	3100	3100

Tabelle 5.2: Anzahl der instanzierten Entitäten bis 30 FPS erreicht sind in den Anwendungen mit dem ECS in Unity

Die Verwendung des ECS von Unity hat wie erwartet dazu geführt, dass deutlich mehr Entitäten instanziert werden können. Jedoch zeigen die Ergebnisse auch, dass der Einsatz des ECS in einigen Bereichen der Anwendungen keine Verbesserung der Performance mit sich bringt.

Die überraschendste Erkenntnis ist die Verschlechterung der Performance, wenn die Farbveränderung des Materials aktiv ist. Dieser Einbruch lässt sich dadurch erklären, dass die Komponente *RenderMesh*, die das Material beinhaltet, von *ISharedComponentData* erbt [Unity, 2019b]. Dadurch muss nämlich für jede Entität ein neues Material und eine neue *RenderMesh* Komponente erstellt werden. Dies ist eine sehr nützliche Optimierung, die ohne die Farbveränderung genutzt wird. Um diesen Performanceeinbruch zu beheben, müsste ein eigenes *RenderSystem* geschrieben werden. Ein Beispiel für eine erfolgreiche Umsetzung eines eigenen *RenderSystem* ist das Projekt *Voxelman* von *kejiro*²⁰.

Eine weitere Erkenntnis ist, dass das ECS bei der Verwendung von komplexen Modellen, wie dem Affenkopf, zu keiner Verbesserung der Performance führt, was weiterhin dafür spricht, dass die Applikation bei komplexeren 3D-Modellen GPU-gebunden ist.

5.2.3 Vergleich der Architekturmuster

In diesem Abschnitt werden die Ergebnisse der Implementierungen der EC und des ECS in Unity miteinander verglichen. Sie sind in der Abbildung 5.1 dargestellt. Die Ergebnisse mit dem Fisch als verwendetes Modell sind aufgrund der Ähnlichkeit zu den Ergebnissen des Würfels nicht Teil der Abbildung. An der y-Achse ist die Anzahl der Entitäten aufgetragen und an der x-Achse die Kombination der aktiven Komponenten. Die Ergebnisse der EC sind für den Würfel in blau beziehungsweise für den Affenkopf in grau eingetragen. Die Ergebnisse des ECS sind für den Würfel in orange beziehungsweise für den Affenkopf in gelb eingetragen.

²⁰Das Projekt ist hier zu finden: <https://github.com/kejiro/Voxelman>

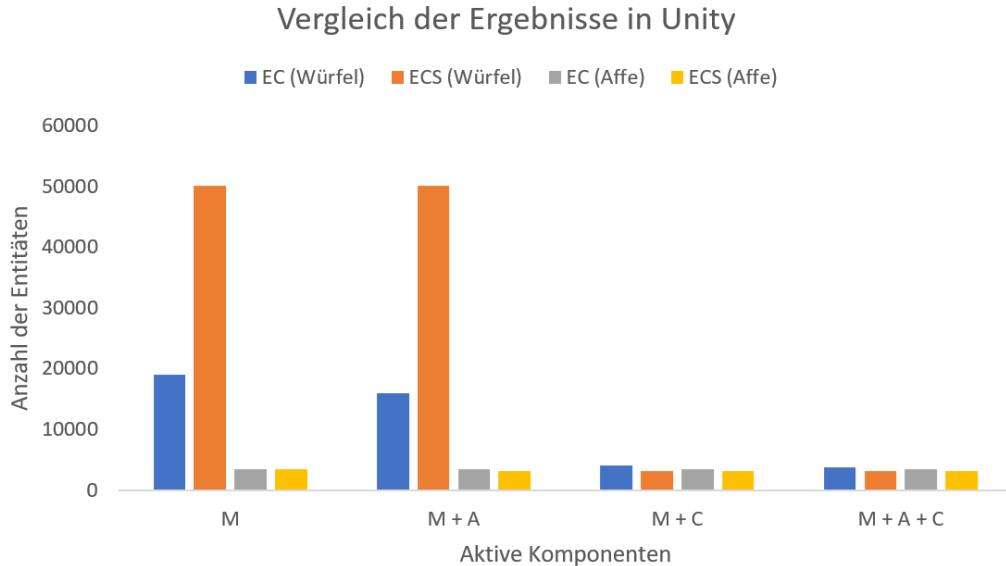


Abbildung 5.1: Diagramm zu den Ergebnissen in der Unity Engine

Aus dem Diagramm lässt sich erkennen, dass bei der Verwendung des Würfels und den aktiven Komponenten *Movement* und *AvoidPredator* die Anzahl der Entitäten im ECS im Vergleich zur EC um über 300 Prozent gesteigert werden konnte. Diese Steigerung beweist, dass der Einsatz des ECS in der Unity Engine zu einer deutlich besseren Performance führen kann. Jedoch zeigen die Ergebnisse für den Affenkopf, dass das ECS bei komplexen Modellen an seine Grenzen stößt.

Außerdem führt die Verwendung der Komponente *ChangeColor* im ECS zu solch starken Performanceeinbußen, dass die Anwendung in der EC sogar bessere Ergebnisse erzielt. Dies zeigt, dass das System *RenderMeshSystemV2*, das im ECS für das Rendering verantwortlich ist, nicht für diesen Anwendungsbereich optimiert ist.

Insgesamt ist hieraus zu schließen, dass obwohl Unitys ECS in den meisten Fällen zu einer deutlichen Performance-Steigerung beiträgt, nicht alle Anwendungsbereiche abgedeckt werden und es sich nicht als Allzweckwaffe zur Performanceoptimierung eignet.

5.3 Realisierung in C++

In diesem Abschnitt werden die Ergebnisse der eigenen Implementierung einer EC sowie eines ECS in der Programmiersprache C++ beschrieben und interpretiert. Anschließend werden die Ergebnisse der beiden Architekturmuster miteinander verglichen.

5.3.1 Evaluation der Entity Component Architecture

Modell	M	M + A	M + C	M + A + C
Würfel	18000	18000	18000	18000
Fisch	18000	18000	18000	18000
Affenkopf	4300	4300	4300	4300

Tabelle 5.3: Anzahl der instanzierten Entitäten bis 30 FPS erreicht sind in den Anwendungen mit der EC in C++

Die Ergebnisse der Implementierung der EC sind über allen Kombinationen von Komponenten konstant. Ob nun eine oder drei Komponenten verwendet werden, führt zu keinen messbaren Veränderungen. Außerdem fällt auf, dass die Komplexität des Modells erst bei der Verwendung des Affenkopfs einen Einfluss auf die Performance hat. Ein Grund für die konstanten Ergebnisse könnte sein, dass die Komponenten nur wenige Daten besitzen und ein simples Verhalten implementieren. Dadurch steigt die Prozessorauslastung in diesen Bereichen nur geringfügig. Ein anderer möglicher Grund ist, dass die Performance limitiert durch das Rendering der Anwendungen ist.

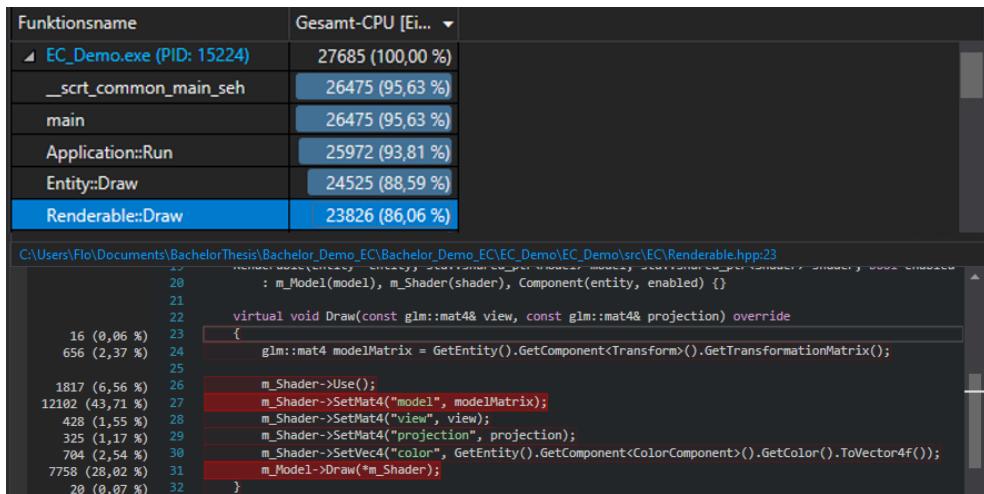


Abbildung 5.2: CPU-Auslastung der Anwendung mit der EC in C++

Abbildung 5.2 zeigt die CPU-Auslastung der Anwendung, wenn alle Komponenten aktiv sind. Dabei wird deutlich, dass ein Großteil der CPU von der Funktion *Draw* der Komponente *Renderable* in Anspruch genommen wird. Insbesondere das Setzen der Modelmatrix im *Shader* und der Aufruf der Funktion *Draw* der Klasse *Model* sind rechenintensiv. Dies könnte erklären, warum die Anwendung unabhängig von den verwendeten Komponenten ein Maximum von 18000 Entitäten erreicht. Bei der Verwendung komplexerer Systeme, die die CPU ebenfalls stark beanspruchen, sollte die Anzahl der Entitäten jedoch sinken.

5.3.2 Evaluation des Entity Component Systems

Modell	M	M + A	M + C	M + A + C
Würfel	20000	20000	19500	19500
Fisch	20000	19500	20000	20000
Affenkopf	4300	4300	4300	4300

Tabelle 5.4: Anzahl der instanzierten Entitäten bis 30 FPS erreicht sind in den Anwendungen mit dem ECS in C++

Die Ergebnisse der Implementierung des ECS in C++ sind ebenfalls über alle Parameter hinweg konstant. Die Abweichung bei den aktiven Komponenten bei der Verwendung der Modelle Fisch und Würfel sind für das Gesamtergebnis von geringer Bedeutung. Ansonsten fällt auf, dass die Werte bei der Verwendung des Affenkopfs erneut deutlich geringer als bei den anderen Modellen sind. Dies bestätigt die Schlussfolgerung, dass sich erst ab einer bestimmten Komplexität des Modells die Performance der Anwendung verschlechtert. Außerdem zeigt es, dass das ECS in diesem Bereich keine Verbesserung herbeiführen kann. In Abbildung 5.3 ist die CPU-Auslastung der Anwendung dargestellt. Es lässt sich erkennen, dass der Großteil der CPU von der Funktion *Update* der Klasse *RenderSystem* in Anspruch genommen wird. Insbesondere die Funktionen *Use* des *Shaders* sowie die Funktion *Draw* der Klasse *Model* beanspruchen die CPU sehr. Dies zeigt, dass in den Test-Applikationen mit dem ECS ebenfalls das Rendering der limitierende Faktor ist und nicht direkt durch das ECS verbesserbar ist.

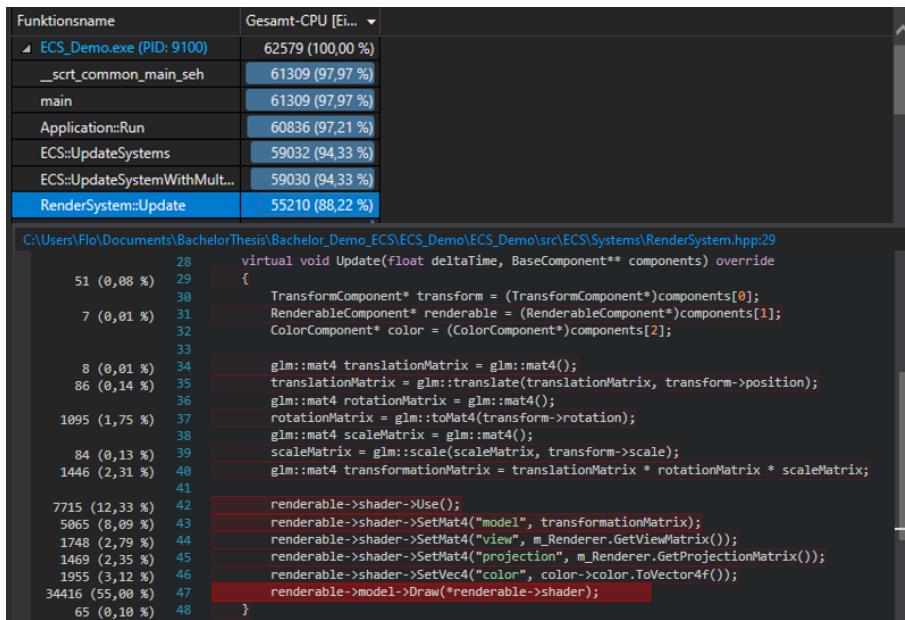


Abbildung 5.3: CPU-Auslastung der Anwendung mit dem ECS in C++

5.3.3 Vergleich der Architekturmuster

In diesem Abschnitt werden die Ergebnisse der Implementierung der EC sowie des ECS der eigenen Anwendung in der Programmiersprache C++ miteinander verglichen. Sie sind in der Abbildung 5.4 dargestellt. Da die Ergebnisse bei der Verwendung des Modells Fisch keine großen Unterschiede zu denen des Modells Würfel aufgewiesen haben, sind diese nicht Teil der Abbildung. An der y-Achse ist die Anzahl der Entitäten aufgetragen und an der x-Achse die Kombination der aktiven Komponenten. Die Ergebnisse der EC sind für den Würfel in blau beziehungsweise für den Affenkopf in grau eingetragen. Die Ergebnisse des ECS sind für den Würfel in orange beziehungsweise für den Affenkopf in gelb eingetragen.

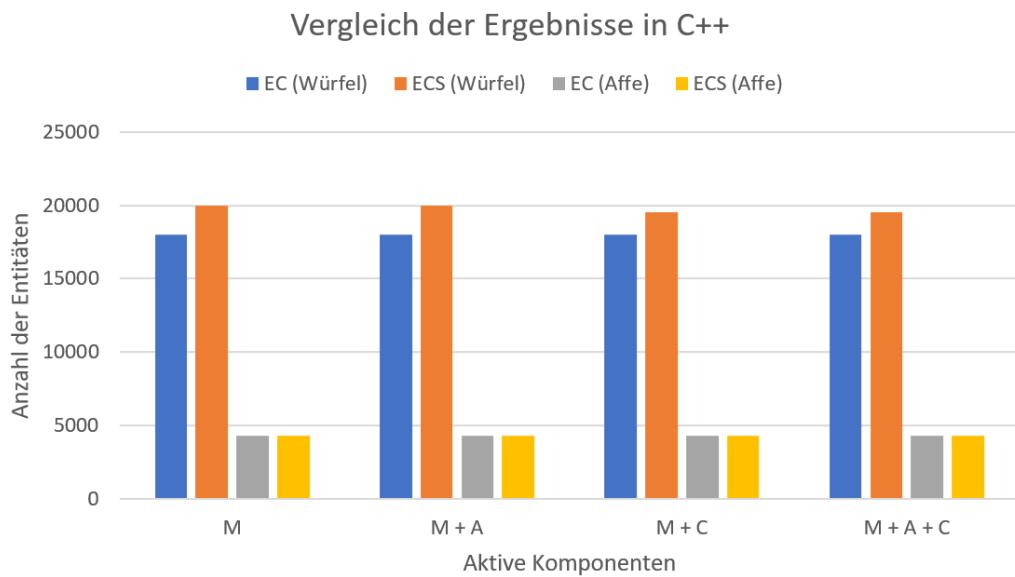


Abbildung 5.4: Diagramm zu den Ergebnissen in C++

Beim Vergleich der Ergebnisse fällt auf, dass es bei der Verwendung des Affenkopfs als Modell keine Unterschiede gibt. Dies zeigt, dass das ECS keinen Einfluss auf die Auslastung der GPU nehmen kann. Bei der Verwendung der Modelle Würfel und Fisch ist zu erkennen, dass sich die Performance mit dem ECS leicht verbessert hat. Die Verbesserung beträgt ungefähr 10 Prozent.

5.4 Vergleich der Testergebnisse in Unity und C++

In diesem Abschnitt werden die Testergebnisse aus den Anwendungen, die mit der Unity Engine entwickelt worden sind, mit den Ergebnissen aus den Anwendungen verglichen, die in der Programmiersprache C++ entwickelt worden sind.

Die Testergebnisse zeigen, dass die Verwendung des ECS in der Unity Engine zu einer deutlich stärkeren Verbesserung der Performance geführt hat als bei der eigenen Implementierung in C++. Dass die Performanceverbesserung so gering ausfällt, kann unterschiedliche Gründe haben.

Zum einen sind die Komponenten in der EC in C++ sehr simpel gehalten und besitzen keinerlei irrelevante Daten. In der Unity Engine sind, zumindest die von Unity bereitgestellten, Komponenten deutlich umfangreicher, weil sie große Bereiche der Funktionalität der Engine abdecken. Daher kann durch die Nutzung des ECS in der Unity Engine dieser Bereich deutlich besser optimiert werden als in der eigenen Implementierung.

Ein weiter Grund ist, dass, wie man in den Abbildungen 5.2 und 5.3 erkennen kann, das Rendering der limitierende Faktor der Anwendungen in C++ ist. In diesem Bereich nimmt das ECS keinerlei Optimierungen vor. In der Unity Engine wird hingegen dieser Bereich durch die Verwendung der *ISharedComponentData* für die Komponente *RenderMesh* stark optimiert. Dies könnte in der Anwendung in C++ ähnlich implementiert werden und würde dazu führen, dass die Funktionen zum Rendering nicht so oft ausgeführt werden müssen.

Neben diesen Faktoren gibt es bei der vorgenommenen Implementierung des ECS in der Programmiersprache C++ einige Bereiche, die optimiert werden können. Zum Beispiel werden die Systeme in jedem Frame für jede Entität, die die gesuchten Komponenten besitzt, aufgerufen. Eine mögliche Optimierung wäre, dass ein System nur ein einziges Mal pro Frame aufgerufen wird und innerhalb der Funktion *Update* alle Entitäten aktualisiert. Dies würde zu weniger Aufrufen der virtuellen Funktion *Update* führen. In der Unity Engine ist dies der Fall.

Des Weiteren kann das Schreiben und Lesen des Speichers weiter optimiert werden. In der Unity Engine werden die Komponenten einer Entität als *EntityArchetype* gruppiert. Diese Gruppen werden im Speicher innerhalb der gleichen Blöcke angelegt[Unity, 2019c]. So entstehen weniger *cache misses*, wenn nach einem bestimmten *EntityArchetype* gesucht wird. Bei der eigenen Implementierung in C++ sind die Komponenten als Blöcke ihres Komponententyps angelegt.

Das wohl überraschendste Ergebnis der Tests ist, dass die Verwendung der Komponente *ChangeColor* in der Unity Engine zu einer immensen Verschlechterung der Performance führt. In den Anwendungen in C++ führt die Verwendung zu keinerlei Verschlechterung der Performance. Dies lässt sich aber ebenfalls dadurch begründen, dass Unity für das Rendering eine *SharedComponent* verwendet. In der eigenen Implementierung konnten das Rendering und die Farbe des Modells in zwei voneinander getrennte Komponenten aufgeteilt werden.

6 Resümee

Dieses Kapitel bildet den Abschluss der Thesis. Zunächst soll ein Ausblick über die Weiterentwicklung und Verwendung des ECS in der Unity Engine gewährt werden. Abschließend werden im Fazit die Implementierung der Anwendungen sowie die entstandenen Ergebnisse reflektiert.

6.1 Ausblick

Die Entwicklung des ECS in der Unity Engine wird in den nächsten Jahren weiter vorangetrieben und wird wahrscheinlich die aktuelle Architektur, die in dieser Arbeit EC genannt wird, als Standard für die Entwicklung von Videospielen ablösen. Wie in allen anderen Bereichen der Videospielentwicklung wird Unity umfangreiche Tools und Schnittstellen für die Verwendung des ECS anbieten. Mit dem *Data-Oriented Technology Stack* bietet Unity außerdem neben dem ECS den Burst-Compiler und das *Job-System* an. Diese Tools erlauben weitere Verbesserungen der Performance und profitieren von der Verwendung des ECS. Da die Verwendung keinerlei Nachteile im Vergleich zur vorhandenen Technologie bietet, dürften in naher Zukunft ein Großteil der Anwendungen, die in der Unity Engine entwickelt werden, diese Tools zur Verbesserung der Performance nutzen.

Die Zukunft der Verwendung von ECS außerhalb der Unity Engine ist schwieriger zu bewerten. Jedoch spricht die Implementierung innerhalb der Unity Engine dafür, dass datenorientiertes Design immer beliebter in der Videospielindustrie wird. Einige große Videospielentwickler nutzen es bereits und daher ist anzunehmen, dass sich diese Denkweise generell durchsetzen wird. Dabei sollte aber bedacht werden, dass bei der Entwicklung der Unity Engine ein riesiges Team zur Verfügung steht. Des Weiteren ist die Engine das Hauptprodukt des Unternehmens. Andere Videospielentwickler spezialisieren sich hingegen auf bestimmte Genres oder Plattformen, was dazu führen kann, dass die Implementierung eines ECS zu viele Ressourcen in Anspruch nimmt.

6.2 Fazit

Die Implementierung eines ECS ist eine komplexe Aufgabe. Sie bietet jedoch die Möglichkeit bei einer erfolgreichen Realisierung die Performance eines Videospiels deutlich zu verbessern und ist daher die meisten Anwendungen sehr nützlich. Als Entwickler ist die Konzipierung eines ECS sehr interessant, da sie eine außergewöhnliche Denkweise verlangt. Hinzu kommt, dass viele Bereiche der Videospielentwicklung Teil dieses Architekturmusters sind. Von der manuellen Zuweisung des Speichers bis hin zur Implementierung einer Schnittstelle zur Verwendung bietet das ECS viele, spannende Aufgabenbereiche.

In der Unity Engine wird diese Schnittstelle zur Verwendung momentan erstellt. Einige Bereiche sind bereits verfügbar und sehr einfach zu nutzen. Die Ergebnisse dieser Arbeit haben jedoch auch gezeigt, dass in anderen Bereichen noch Entwicklungszeit benötigt wird. Außerdem ist die Dokumentation des ECS in der Unity Engine aktuell unzureichend und erschwert die Verwendung. Dies wird sich wahrscheinlich ändern, sobald das ECS vollständig in der Unity Engine implementiert ist. Was durch die Verwendung des *Data-Oriented Technology Stacks* in der Unity Engine möglich ist zeigen die vielen Anwendungen, die Unity selbst und Nutzer der Engine bereits veröffentlicht haben.

Diese Arbeit kann als Grundlage zum Einstieg in das ECS gesehen werden. Sie beschreibt den aktuellen Stand der Industrie, erklärt die theoretischen Grundlagen zum Verständnis des Architekturmusters und bietet einen Einblick in die Verwendung und Realisierung des ECS. Für eine erfolgreiche Implementierung sind jedoch deutlich mehr Erfahrung und Zeit nötig. Außerdem fehlt es den Anwendungen an Komplexität, um zu bewerten, unter welchen Bedingungen sich eine Verwendung des ECS zielführend ist. Sie sind zu weit von einem verkaufsfähigen Videospiel entfernt. Die Ergebnisse zeigen lediglich, dass sich eine Implementierung unter bestimmten Umständen zur Verbesserung der Performance lohnen kann.

7 Glossar

ECS Entity Component System

EC Entity Component Architecture

CPU Central Processing Unit

GPU Graphics Processing Unit

FPS Frames pro Sekunde

RTTI Runtime Type Information

MMORPGs Massive Multiplayer Online Roleplaying Games

RTSs Real Time Strategy Games

8 Abbildungsverzeichnis

Wenn nicht anders angegeben, eigene Grafik.

2.1	Überblick über die Systeme und Schichten einer Game Engine [Gregory, 2018, 39]	7
2.2	Beispiel einer Vererbungshierarchie in einem Videospiel [Jordan, 2018]	11
2.3	Veränderte Vererbungshierarchie mit Mehrfachvererbung	12
2.4	Deadly Diamond of Death [Martin, 1997]	13
2.5	Erstellung von Entitäten durch Komposition [Jordan, 2018]	14
2.6	In der Unity Engine kann man über den <i>Inspector</i> (rechts) Komponenten von <i>GameObjects</i> editieren, entfernen und erstellen	15
2.7	Skizze zum Zusammenhang von Entitäten und Komponenten [Jordan, 2018]	18
2.8	Skizze zum Zusammenhang von Komponenten und Systemen [Jordan, 2018]	20
2.9	Steigerung der Performance von Prozessor und Speicher von 1980 bis 2010 [Cheney, 2019]	21
2.10	Vereinfachte Darstellung des Speichers von <i>GameObjects</i> und ihren Komponenten [Ferreira and Geig, 2018]	22
2.11	Daten der Komponente <i>Transform</i> und eines Skript zur Bewegung von Entitäten in der Unity Engine [Ferreira and Geig, 2018]	24
2.12	Der Entity Debugger der Unity Engine	27
3.1	Skizziertes Design und Ablauf aller Test-Applikationen	30
3.2	UML-Diagramm, reduziert auf die relevanten Informationen, zur Konzeption der Applikation mit der EC in der Unity Engine	33
3.3	UML-Diagramm, reduziert auf die relevanten Informationen, zur Konzeption der Applikation mit dem ECS in der Unity Engine	35
3.4	Vereinfachtes UML-Diagramm der Klasse Application	38
3.5	UML-Diagramm zur Funktionsweise von Window, Events und GLFW	39
3.6	UML-Diagramm zur Funktionsweise von Renderer, Mesh, Model und Shader	40
3.7	UML-Diagramm der Klasse ImGuiLayer	41
3.8	UML-Diagramm zur EC in C++	43
3.9	UML-Diagramm zum ECS in C++	47
4.1	Die Test-Applikation mit der EC im Unity Editor	48
4.2	Die Komponenten des <i>Perfabs Fish</i> im Unity Inspector	49
4.3	Die Test-Applikation mit dem ECS im Unity Editor	50

8. Abbildungsverzeichnis

4.4	Die Komponenten des <i>Prefabs Fish</i> im Unity Inspector	51
4.5	Die Funktion <i>SpawnObjects</i> der Klasse <i>DemoManagerECS</i>	52
4.6	Die Komponenten des <i>GameObjects Predator</i> im Unity Inspector	53
4.7	Die Klasse <i>MovementSystem</i> implementiert die Bewegung von Entitäten	54
4.8	Die Klasse <i>ColorChangeSystem</i> dient zur Farbänderung des Materials	55
4.9	Das Makro CLASS_DECLARATION	56
4.10	Das Makro CLASS_DEFINITION	56
4.11	Die Datei <i>ComponentClasses.cpp</i>	57
4.12	So können Komponenten hinzugefügt, entfernt und erhalten werden .	57
4.13	Realisierung der Funktionen <i>AddComponent</i> , <i>RemoveComponent</i> und <i>GetComponent</i>	58
4.14	Definition der Funktionen <i>ComponentCreate</i> und <i>ComponentFree</i> . . .	59
4.15	Initialisierung der Variablen der Komponenten	60
4.16	Realisierung der <i>TransformComponent</i>	60
4.17	Realisierung des <i>RenderSystems</i>	61
4.18	Aktualisierung der Systeme	62
4.19	Mit der Funktion <i>GetComponentInternal</i> wird überprüft, ob eine Ent- ität eine bestimmte Komponente besitzt	63
4.20	Mit der Funktion <i>MakeEntity</i> werden neue Entitäten und ihre Kom- ponenten erstellt	63
5.1	Diagramm zu den Ergebnissen in der Unity Engine	67
5.2	CPU-Auslastung der Anwendung mit der EC in C++	68
5.3	CPU-Auslastung der Anwendung mit dem ECS in C++	69
5.4	Diagramm zu den Ergebnissen in C++	70

9 Tabellenverzeichnis

5.1	Anzahl der instanzierten Entitäten bis 30 FPS erreicht sind in den Anwendungen mit der EC in Unity	65
5.2	Anzahl der instanzierten Entitäten bis 30 FPS erreicht sind in den Anwendungen mit dem ECS in Unity	66
5.3	Anzahl der instanzierten Entitäten bis 30 FPS erreicht sind in den Anwendungen mit der EC in C++	68
5.4	Anzahl der instanzierten Entitäten bis 30 FPS erreicht sind in den Anwendungen mit dem ECS in C++	69

10 Literaturverzeichnis

- Acton, M. (2014). Data-Oriented Design and C++. <https://www.youtube.com/watch?v=rX0ItVEVjHc>(Zugriff: 08.09.2019).
- Alex (2018). The virtual table. <https://www.learncpp.com/cpp-tutorial/125-the-virtual-table/>(Zugriff: 15.09.2019).
- Busby, J., Parrish, Z., and Wilson, J. (2009). Introduction to Unreal Technology. <http://www.informit.com/articles/article.aspx?p=1377834>(Zugriff: 26.08.2019).
- Cheney, D. (2019). High Performance Go Workshop. <https://dave.cheney.net/high-performance-go-workshop/gopherchina-2019.html>(Zugriff: 14.09.2019).
- Ferreira, C. and Geig, M. (2018). Get Started with the Unity* Entity Component System (ECS), C# Job System, and Burst Compiler. <https://software.intel.com/en-us/articles/get-started-with-the-unity-entity-component-system-ecs-c-sharp-job-system-and-burst-compiler>. (14.09.2019).
- forsakenforgotten (2018). Still unsure whether ECS is right for your game? This might help. https://www.reddit.com/r/roguelikedev/comments/849x8p/still_unsure_whether_ecs_is_right_for_your_game/dvp5ucs/(Zugriff: 16.09.2019).
- Gamma, E., Vlissides, J., Helm, R., and Johnson, R. (1994). *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- Gaul, R. (2014). Sane Usage of Components and Entity Systems. <https://www.randygaul.net/2014/06/10/sane-usage-of-components-and-entity-systems/>(Zugriff: 16.09.2019).
- Gregory, J. (2018). *Game Engine Architecture*. CRC Press, Taylor & Francis Group, third edition.
- ITWissen.info (2009). Vererbung. <https://www.itwissen.info/Vererbung-inheritance.html>(Zugriff: 02.09.2019).
- ITWissen.info (2018). Architekturmuster. <https://www.itwissen.info/Architekturmuster-architectural-style.html>(Zugriff: 27.08.2019).
- Jordan, M. (2018). Entities, components and systems. <https://medium.com/ingeniouslysimple/entities-components-and-systems-89c31464240d>(Zugriff: 02.09.2019).

- Llopis, N. (2009). Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP). <http://gamesfromwithin.com/data-oriented-design>(Zugriff: 16.09.2019).
- Martin, A. (2007). Entity Systems are the future of MMOG development. <http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/>.
- Martin, R. C. (1997). Java and C++ A critical comparison. <https://web.archive.org/web/20051024230813/http://www.objectmentor.com/resources/articles/javacpp.pdf>(Zugriff: 02.09.2019).
- Mertens, S. (2019). Entity Component Systems FAQ. <https://github.com/SanderMertens/ecs-faq#what-are-examples-of-ecs-implementations>(Zugriff: 16.09.2019).
- Nikolov, S. (2018). OOP Is Dead, Long Live Data-Oriented Design. <https://www.youtube.com/watch?v=yy8jQgmhbAU&t=>(Zugriff: 25.08.2019).
- Nystrom, R. (2014). *Game Programming Patterns*. Genever Benning.
- Paradigm, V. (2019). UML Association vs Aggregation vs Composition. <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition/>(Zugriff: 02.09.2019).
- Prühs, N. (2014). Component-Based Entity Systems in Spielen. <https://www.heise.de/developer/artikel/Component-Based-Entity-Systems-in-Spielen-2262126.html?seite=all>(Zugriff: 02.09.2019).
- steamspy (2019). Games released in previous months. <https://steamspy.com/year/>(Zugriff: 25.08.2019).
- Taylor, R. N., Medvidovic, N., and Dashofy, E. M. (2007). *Software Architecture: Foundations, Theory and Practice*. Addison-Wesley.
- TheChernoProject (2018). What is a GAME ENGINE? <https://www.youtube.com/watch?v=vtWdgtMo1T4&t=>(Zugriff: 26.08.2019).
- Unity (2019a). Leistung als Standard (DOTS). <https://unity.com/de/dots>(Zugriff: 25.08.2019).
- Unity (2019b). Shared ComponentData. https://docs.unity3d.com/Packages/com.unity.entities@0.0/manual/shared_component_data.html.
- Unity (2019c). Struct EntityArchetype. <https://docs.unity3d.com/Packages/com.unity.entities@0.0/api/Unity.Entities.EntityArchetype.html>.

Ich versichere, die vorliegende Arbeit selbstständig ohne fremde Hilfe verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt zu haben. Die aus anderen Werken wörtlich entnommenen Stellen oder dem Sinn nach entlehnten Passagen sind durch Quellenangaben eindeutig kenntlich gemacht.

Hamburg, 14.10.2019

Florian Völkers