BACHELORARBEIT

im Studiengang Bachelor Informatik

Komponentenbasierte Game Engines und Data-Driven Game Objects

Ausgeführt von: Simon Dobersberger
Personenkennzeichen: 1010257008

BegutachterIn: DI Stefan Reinalter

Wien, 1. Juni 2013

FACHHOCHSCHULE
TECHNIKUM WIEN

Eidesstattliche Erklärung

„Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Ich versichere, dass die abgegebene Version jener im Uploadtool entspricht."

Ort, Datum

Unterschrift

Kurzfassung

Komponentenbasierte Spiele-Engines und ihre Data-Driven Game Objects bieten eine moderne Alternative zum üblichen Vererbungsmodell in der Spiele-Branche. Diese Bachelorarbeit im Rahmen des Studiums an der FH Technikum Wien behandelt folgende Themen:

- Die Analyse der Problematik bei der Verwendung von Vererbung für die Implementierung von Spielobjekten mündet im Vergleich mit den Lösungsansätzen der komponentenbasierten Engines. Weiters wird dies durch drei verschiedene Beispiele zum Einsatz von diesen in der Spiele-Industrie und der Unity3D-Engine gestützt.
- Der Vergleich von Vor- und Nachteilen dieser Engines und von Data-Driven Game Objects ergibt, dass zahlreiche Vorteile in der Bereichen der Speicherverwaltung, des Multithreadings, der Ausführungsreihenfolge und der Serialisation gegeben sind. Dem gegenüber steht eine erhöhte Komplexität und ein geringfügiger Leistungsaufwand.
- Eine Analyse der verschiedenen Implementierungsstrategien zeigt die Vorteile eines Komponenten-Managers. Zudem helfen spezielle Systeme bei der Abkoppelung der Logik von den Komponenten. Die Kommunikation zwischen den Komponenten stellt eine schwierige Aufgabe dar und lässt sich mithilfe des Komponenten-Managers und Event-Systemen lösen.

Schlagwörter: Komponentenbasierte Spiele-Engines, Komponenten, Game Engines, Unity3D, Data-Driven Game Objects

Abstract

Component-based game engines and their data-driven game objects provide a modern alternative to the traditional inheritence model. This bachelor thesis, which is within the scope of the studies at the UAS Technikum Vienna, introduces the following topics:

- The analysis of the problems with the inheritence model and its implementation of game objects leads to a comparison with the solutions of the component-based engines. Furthermore, this will be backed up by three different examples of their usage in the gaming industry and the Unity3D engine.
- The comparison of the pros and cons of these engines and data-driven game objects shows that there are many advantages in the fields of memory management, multithreading, execution order and serialization. The disadvantages are a higher complexity and minimal performance costs.
- An analysis of the different implementation possibilities shows the advantages of a component manager. Moreover, special systems help detaching the component's logic from the components themselves. The communication between components is a difficult task that is solved by the component manager and event systems.

Keywords: Component-based game engines, components, game engines, Unity3D, data-driven game objects

Danksagung

Ich bedanke mich vielmals bei meiner Familie die mich während meiner Zeit in Hamburg tatkräftig unterstützt hat. Mein Dank gilt ebenfalls meinem Betreuer der mir stets bei Fragen geholfen hat und mich mit seinem Feedback auf dem korrekten Weg gehalten hat. Nicht zuletzt sei hier auch mein Programming-Lead bei Daedalic Entertainment erwähnt, der mir bei Verständnis-Problemen mit komplexen Event-Systemen und komponentenbasierten Engines mit Rat und Tat zur Seite stand.

Inhaltsverzeichnis

1 Einleitung

Die Hauptaufgabe einer Spiele-Engine ist es dem Entwickler die Möglichkeit zu bieten, sein Projekt mit beliebigen Spielinhalten zu füllen und diese dem Endnutzer zu präsentieren. Um das zu bewerkstelligen existieren zahlreiche verschiedene Ansätze, was die Programmierung und die Bearbeitung der Daten im Spiel betrifft. Ein häufig gewählter und naheliegender Ansatz ist das Prinzip der Vererbung, welches dem Gedanken der objektorientierten Programmierung entspricht. Hierbei werden nach und nach Spielobjekte innerhalb der gewählten Programmiersprache erstellt, welche jeweils von verschiedenen Basisklassen und anderen Objekten erben. Mithilfe dieser Vererbungsstrategie werden Logik und Eigenschaften unter den oftmals als *Game Objects* bezeichneten Spielobjekten geteilt.

Als eine Alternative zu diesem Ansatz bieten sich sogenannte "Komponentenbasierte Game Engines" an. Anstatt das Verhalten und die Eigenschaften eines Spielobjekts direkt zu implementieren, wird dieses durch einzelne Komponenten definiert. Dabei handelt es sich um kleine Logik- und Engine-Bestandteile die in ihrer Summe eine Entität ergeben. Folglich wird der Prozess zur Erstellung von Spielobjekten wesentlich dynamischer und es wird dem Entwickler ermöglicht, selbst während der Laufzeit das Verhalten eines Objektes im Spiel komplett zu verändern. Es gilt daher diesen Ansatz anhand von zahlreichen Beispielen und der Implementierung zu analysieren.

Um diesen Prozess der Objekt-Definition komplett aus der Programmierung abzukoppeln, wird oftmals der Ansatz der *Data-Driven Game Objects* verfolgt. Dabei handelt es sich um den praktischen Einsatz des Data-Driven Designs zur Erstellung von Spielobjekten aus eigenen Datenquellen. Es kann sich diesbezüglich um eine Datenbank sowie eine XML- oder auch eine einfache Text-Datei handeln, welche die Zusammensetzung einer Entität durch ihre Komponenten beschreibt.

Ein populäres Beispiel für diesen Ansatz findet sich in der Unity3D-Engine, welches für die Entwicklung von Spielen und Simulationen verwendet wird. Mithilfe seines intuitiven Aufbaus und den flexiblen Möglichkeiten zur Erstellung von Komponenten und Skripten, hat sich Unity3D als Spiele-Engine etabliert. Zudem existieren zahlreiche weitere Beispiele zur Verwendung und Entwicklung von komponentenbasierten Spiele-Engines und Data-Driven Game Objects, welche von diversen Veteranen der Spiele-Branche näher beschrieben werden.

2 Entwicklung der komponentenbasierten Engines und Data-Driven Game Objects

2.1 Umstieg vom Vererbungsmodell

In der Spiele-Industrie existieren diverse Herangehensweisen, wenn es um die Struktur einer Spiele-Engine geht. Das komponentenbasierende System ist lediglich eines davon. Der Aufbau einer Engine anhand eines Vererbungsmodells war lange Zeit ein weit verbreiteter Standard in der Branche und wurde von zahlreichen Entwicklern praktiziert.

Dieser Ansatz wird unter anderem auch in der Unreal Engine 3 eingesetzt[1]. Hierbei wird eine “Actor”-Klasse zur Verfügung gestellt. Diese Basisklasse liefert diverse Funktionalitäten wie etwa Rendering, Physik oder Sound. Bei der Implementierung einer Unterklasse wie zum Beispiel "Rakete"wird dieser Funktionsumfang erweitert. Zwar lassen sich auf diese Weise in möglichst kurzer Zeit Ergebnisse erzielen, doch auch wenn es dem Programmierer mithilfe des objektorientierten Ansatzes natürlich erscheint, so stößt dieser alsbald auf diverse Komplikationen. Einer dieser Entwickler ist Terrance Cohen der auf der Game Developers Conference 2010 in Canada die Problematiken dieses Modells näher beleuchtet[2] (siehe auch Kapitel 2.2.3).

2.1.1 Die Problematik

In einem Spieleprojekt welches mithilfe einer tiefen Objekt-Hierachie funktioniert, werden während der Laufzeit zahlreiche Spiel-Objekte mit ihren diversen Funktionen und Variablen allokiert. Unabhängig ist dabei, ob der Großteil dieser Member überhaupt für das Spiel benötigt werden. Dies ist fix vorgeben, wie es sich anhand eines Beispiels mit einer Rakete leicht erklären lässt. Sie verhält sich unabhängig von der aktuellen Spielsituation stets gleich. Sprich, sie erbt von der Klasse “Flugobjekt”, wodurch ihr Flugeigenschaften zugesprochen werden, sowie von “Waffe”, mit welcher sich Feinde besiegen lassen. Dabei wird jeder einzelner Raketentyp neu implementiert bzw. erben alle Typen wiederum von der Klasse “Rakete”. Die folgende Abbildung 2.1 soll dies noch einmal verdeutlichen.

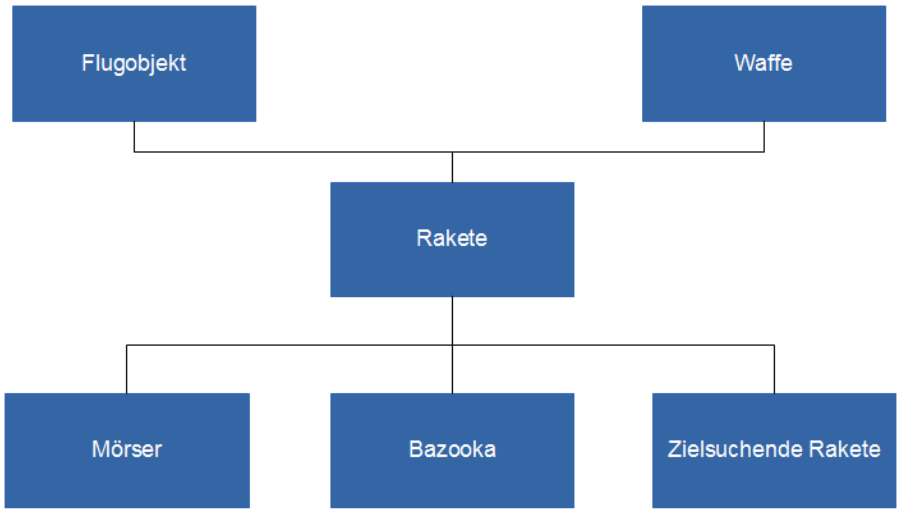

Abbildung 2.1: Klassenhierarchie des Raketenbeispiels

So entsteht für den Spieler ein Arsenal aus verschiedenen Raketen und für den Programmierer ein selbiges aus vielen verschiedenen Klassen. Bei einem Spiel der entsprechenden Größe führt dies unweigerlich zu einer Vielzahl von solchen sowie einer unübersichtlichen Hierarchie. Weiters werden Resourcen einer übergeordneten Klasse eventuell gar nicht benötigt. Zum Beispiel benötigt die Kindklasse "Mörser"keine Beinflussung durch den Wind. Die Member werden aber dennoch durch die Basisklasse mitgeliefert.

Die Eigenschaften dieser Spielobjekte sind während der gesamten Spielzeit gleich und können programmiertechnisch nicht verändert werden. Für eine Veränderung des Verhaltens wird eine neue Klasse erstellt und das entsprechende Objekt instanziert. Dieser Prozess führt unter anderem zu Code-Duplikaten oder der Notwendigkeit einer Mehrfachvererbung. Letztere ist insofern bedenkenswert, da auf diese Weise nicht jede Programmiersprache für die Entwicklung der Engine in Frage kommt [1]. Im schlechtesten Fall hat der Programmierer sämtlichen Code in die oberste gemeinsame Basisklasse zu transferieren, wodurch zum Beispiel “Entity” oder “Game Object” im Laufe der Entwicklung zu einer unübersichtlichen Größe anwächst. Dies führt wiederum zu dem besagten Problem, dass jede Entität im Spiel Features erbt, die sie gar nicht benötigt. Dieser Overhead übt in weiterer Folge einen negativen Einfluss auf die Performance und den Speicherverbrauch der Anwendung aus.

Für neue oder außenstehende Programmierer die für die Implementierung neuer Features verantwortlich sind, führt solch ein Ansatz dazu, dass sich diese in den kompletten Code der Basisklassen einlesen müssen. Ansonsten haben sie zu befürchten, dass das ganze Codegerüst beschädigt wird. Dabei besteht auch die Möglichkeit eines vergessenen Funktionsaufrufs der Basisklasse, wenn eine

Funktion überschrieben wird. Durch die massive Verberbung kommt es langfristig ebenfalls zu Problemen mit der Reihenfolge der Aufrufe. Dies kann zum Beispiel der Fall sein, wenn die überschriebene "Update"-Methode der Unterklasse die Position der Entität verändert, aber die Physik bereits berechnet wurde. Dadurch ist die Transformation nicht mehr gültig, da die Position bereits von der Physik-Engine bearbeitet wurde bzw. ist die physikalische Berechnung ungenau, da noch nachträglich eine Positionsveränderung stattfindet.

Ein weiterer Nachteil ist die fehlende Dynamik im Entwicklungsprozess. Der Programmierer hat stets auf die Wünsche und Ideen des Designers zu reagieren und dementsprechende Klassen zu erstellen und implementieren. Da dieser Arbeitsschritt für den Programmierer jedes Mal wieder einen Aufwand darstellt, kann dies dazu führen, dass sich der Designer in seiner Kreativität eingeschränkt fühlt.

2.1.2 Ein Lösungsansatz

Eine Lösung für die angesprochenen Probleme bietet das komponentenbasierende Modell. Hierbei werden anstelle von festen Spielobjekt-Klassen lediglich sogenannte "Komponenten" implementiert. Dabei handelt es sich um ein kleines Stück einer in sich geschlossenen Logik, welches im Zusammenschluss mit anderen Komponenten ein Objekt im Spiel darstellt. Dieses wird auch als Entität bezeichnet. Im vorher beschriebenen Raketen-Beispiel würde es sich dabei etwa um eine Grafik-, Physik- und eine Partikeleffekt-Komponente für die Explosion sowie eine Schadenskomponente handeln. Diese ergeben zusammen eine Raketen-Entität. Wenn dem Game Designer nun der Partikeleffekt nicht mehr gefällt, braucht der Programmierer lediglich die entsprechende Komponente zu entfernen. Der Programmierer kann sich dadurch im Wesentlichen auf die Entwicklung der eigentlichen Logik im Spiel konzentrieren.

In weiterer Folge bietet sich auch das Konzept der Data-Driven Game Objects an. Dadurch kann der Game oder Level Designer vollkommen unabhängig vom Programmierer agieren, sofern die benötigten Komponenten vorhanden sind. Indem die Komponenten den Entitäten im Spiel durch eine externe Quelle wie etwa einer XML-Datei oder einer Datenbank zugewiesen werden, wird es möglich durch einen einfachen Texteditor oder ein anderes Tool komplett neue Objekte zu erstellen. Ein Beispiel stellt hierfür die Spiele-Engine Unity3D parat. Mit seinem umfangreichen Editor können ohne größeren Aufwand neue Spielobjekte erstellt und mit verschiedenen Komponenten ausgestattet werden, welche in Binär- oder Text-Dateien gespeichert werden.

Ein weiterer Vorteil ist, dass Komponenten jederzeit dynamisch hinzugefügt oder entfernt werden können. Dadurch ist eine Veränderung des Verhaltens während des Spiels oder beim Bearbeiten der Szene möglich, ohne neue Klassen zu erstellen oder das komplette Projekt neu zu kompilieren.

Konkret bedeutet dies, dass auf Vererbung fast komplett verzichtet wird und stattdessen jede Entität eine Referenz zu ihren Komponenten enthält [1]. Dabei handelt es sich zum Beispiel um Transformations-, Render- oder Physik-Komponenten. Verfolgt man diesen Ansatz nun naiv und fügt nach und nach Verweise auf Komponenten in der Entität hinzu, so entstehen zahlreiche Null-Referenzen. Denn nicht jede Entität enthält stets sämtliche Komponenten und wird geupdatet,

wenn eine Referenz hinzugefügt oder entfernt wurde. Es bietet sich daher an, eine Basisklasse für alle Komponenten zur Verfügung zu stellen. Das System wird dadurch wesentlich modularer und Spielobjekte enthalten eine Liste mit ihren Komponenten anstelle von einzelnen Referenzen.

Im Sinne der Data-Driven Game Objects ist es im weiteren Sinne möglich, komplett sämtliche Logik aus Komponenten zu entfernen. Hierzu werden sogenannte "Systeme" eingesetzt, die für ihre jeweiligen Komponenten zuständig sind. Zum Beispiel ist eine Klasse namens "RenderSystem" für sämtliche "RenderComponent"-Objekte zuständig. Die eigentlichen Entitäten werden auf diese Weise komplett aufgegeben und existieren lediglich als ID in ihren Komponenten. Stattdessen wird das Hinzufügen und Entfernen von Komponenten direkt von den Systemen oder zum Beispiel von einer "Entity Manager"-Klasse verwaltet. Diese Architektur ermöglicht es den Klassen untereinander möglichst unabhängig zu agieren. Für die Kommunikation zwischen den Systemen und Komponenten wird auf Mittel wie Skripting, Events oder Message-System zurückgegriffen.

2.2 Erfahrungen aus der Spiele-Branche

2.2.1 Gas Powered Games: Dungeon Siege

Scott Bilas hat zwischen 1999 und 2003 bei Gas Powered Games gearbeitet und war unter anderen an der Entwicklung von Dungeon Siege beteiligt. Besonders hervorzuheben ist sein Vortrag bei der GDC San Jose im Jahre 2002. Er beschrieb darin das Konzept des komponentenbasierenden Systems und der Data-Driven Game Objects. Wie aus seinen Unterlagen zu der Veranstaltung hervorgeht[3], sind hart gecodete Daten innerhalb des C++-Codes mit diversen Schwierigkeiten verbunden. Das Hauptproblem stellt dabei der statische Prozess bei gewünschten Veränderungen dar, bei dem der Programmierer für jeden Prototypen ein neues Build zu erstellen hatte.

Data-Driven Design behebt diese Probleme, weil sämtliche Daten von außerhalb des Programmcodes kommen. Somit können Designer, Animatoren, Grafiker, etc. mit einem Texteditor oder Tool die diversen Einstellungen und Änderungen selbst vornehmen und testen. Auf lange Sicht wird dadurch die Geschwindigkeit des Entwicklungsprozesses beschleunigt.

Für die Verwaltung der Data-Driven Game Objects ist ein spezielles System zuständig. Dieses hat verschiedene Aufgaben. Zum einen setzt es die Objekte zusammen und verwaltet ihre jeweilige ID. Zum anderen ist es zuständig für die Löschung der Objekte und ihrer Daten sowie der Verwaltung von externen Anfragen und der Kommunikation zwischen den einzelnen Data-Driven Game Objects.

Bei der konkreten Umsetzung dieses Systems verweist Bilas zunächst auf die in zahlreichen Büchern empfohlene Vererbung und die Erstellung eines UML-Diagramms für die diversen Objekte im Spiel. Ein anderer Ansatz wäre wiederum die Aufteilung der Aufgaben und Nutzung einer Mehrfachvererbung

In beiden Fällen kommt es im Laufe eines größeren Projekts zur Bildung eines großen, unübersichtlichen Diagramms. Zudem werden zahlreiche virtuelle Funktionen vorausgesetzt. Ein weiteres Problem stellen die Ansprüche an die Game Engine dar, sich auf Veränderungen für das Konzept und Design des Spiels einzustellen. Ein Programmierer hat daher laufend auf die Wünsche eines Designers zu reagieren. Darunter fallen Dinge wie die Erstellung neuer Objekte für einen Prototypen oder das Anpassung des Vererbungsmodells, wenn zum Beispiel eine Rakete nun doch zielsuchend sein oder der Hauptcharakter nun auch schwimmen können soll.

Laut Bilas liegt die Lösung darin nicht nur die Eigenschaften der Objekte im Data-Driven Design umzusetzen, sondern auch deren gesamten Aufbau. Das Ziel ist es die komplette Hierachie des Systems als Data-Driven zu implementieren. In Dungeon Siege war das Ergebnis ein komponentenbasiertes System. Dabei enthält jede Komponente ein in sich geschlossenes Stück Logik. Eine Ansammlung von diesen definiert die verschiedenen Objekte im Spiel und werden zusammen mit ihren Ausganswerten von außen vorgegeben. Dies geschieht zum Beispiel durch eine XML-Datei die vom Level Designer direkt oder per Tool bearbeitet wird. Spielobjekte die auf diese Weise definiert werden, besitzen keine Hierarchie mehr. Stattdessen sind sie für verschieden Aufgaben wie etwa die Verwaltung der Komponenten verantwortlich. Fast die komplette Logik des Spiels läuft auf diese Weise in den einzelnen Komponenten ab.

2.2.2 Neversoft: Tony Hawk's Pro Skater

Wie Mick West 2007 auf seinem Entwickler-Blog berichtet [4], hat Neversoft für die Tony Hawk Serie eine Objekt-Komposition von Komponenten eingesetzt. Prinzipiell hat jedes Object im Spiel eine bestimmte Aufgabe. Darunter fallen Dinge wie etwa das Explodieren einer Rakete, eine Animation, das Folgen eines vorgegebenen Pfades und auch nur die Ausführung eines Skriptes. Der traditionelle Ansatz würde hierbei die Implementierung mittels Vererbung und verschiedenen Unterklassen verfolgen. Dieser Vorhergehensweise bietet sich jedoch unter anderem deshalb nicht an, da es bei Tony Hawk um die Entwicklung einer Serie geht. Daher ist die Wiederverwendung der vorangegangenen Arbeit von großer Wichtigkeit.

Folglich lautet auch hier der Grundsatz "Aggregation statt Vererbung". Anstatt sich auf die verschiedenen Implementierungen eines Spielobjektes zu verlassen, wird dieses lediglich als Container für die einzelnen Komponenten verwendet. Soforn sich ein Großteil des Codes auf dieses konkrete Objekt verlässt, kann es im weiterer Folge als Schnittstelle dienen. Jedoch sollte auch dieser Ansatz nach und nach fallengelassen werden und stattdessen direkt auf die einzelnen Komponenten zugegriffen werden. Die finale Version des Modells nennt sich "Pure Aggregation" und steht dafür, dass ein Spielobjekt bzw. eine Entität lediglich aus der Gesamtheit seiner Komponenten besteht. Stattdessen kümmert sich ein Komponenten-Manager um die Komposition der Objekte. Die nachfolgende Grafik in Abbildung 2.2 verdeutlicht dieses Konzept.

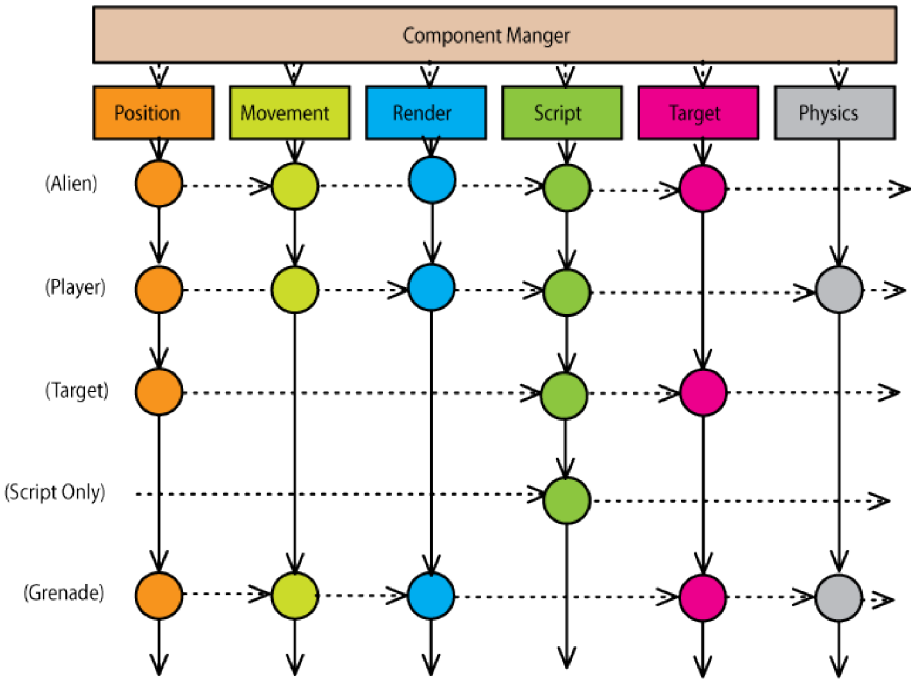

Abbildung 2.2: Aufbau der Entitäten als Komponenten, © cowboyprogramming.com [4]

Bei der Entwicklung der Tony Hawk Spiele bei denen Mick West mitgearbeitet hatte, begann alles mit den Problemen welche bei der intensiven Nutzung von Vererbung auftreteten. Die Spielobjekte beinhalteten zahlreiche Funktionen und Daten, die nicht von Bedeutung sind. Dies zog folglich auch einen Leistungsverlust nach sich und die diversen Funktionalitäten des Objekts wurden ohne Absicht dupliziert.

> "It suffered from all the same problems: the objects tended to be heavyweight. Objects had unnecessary data and functionality. Sometimes the unnecessary functionality slowed down the game. Functionality was sometimes duplicated in different branches of the tree." *Mick West, cowboyprogramming.com, 2007* [4]

In der laufenden Entwicklung stellt sich eine Umstellung auf eine komponentenbasiertes System als schwierig dar. Wie Mick West weiters beschreibt, trägt dieser Ansatz eine Komplexität mit sich, die von anderen Programmierern missverstanden werden kann. Sofern die zunächst aufwändige Umstellung jedoch erfolgreich verläuft, folgt daraus eine schnellere Implementierung neuer Funktionalitäten. Mithilfe von Data-Driven Design lassen sich Objekte direkt von den Designern erstellen, was eine Kapselung und vereinfachte Konfiguration des Spiels mit sich bringt. Ein weiterer, nachträglicher Vorteil ist, dass der Code lesbarer und fehlerfreier wird.

2.2.3 Insomniac Games: Resistance

Mit Insomniac Games hat sich auch eine Firma aus der Konsolen-Branche für das dynamische Konzept der komponentenbasierenden Spiele-Engines entschieden. Bekannt geworden sind diese durch diverse Spieleserien wie etwa Ratchet & Clank und Spyro. Darunter fällt auch die dreiteilige Shooter-Serie namens Resistance. Diese Spiele sind insofern interessant, da sie zu den populäreren Beispielen des Einsatzes von Komponenten und Systemen zählen. Die Verwendung dieses Konzepts ermöglichte Insomniac Games die Performance der Playstation-Konsole vollends auszunutzen. Dies war unter anderem durch den Einsatz von Multithreading und die damit verbundene Nutzung von mehreren Prozessoren möglich. Auf der GDC 2010 in Kanada erklärte Terence Cohen deshalb die Architektur, welche zum Beispiel die dynamische Verwendung von Komponenten mithilfe von Speicher-Pools und eine hochgradig Parallelisierung aufweist [2].

Wie bereits in *Kapitel 2.1.1. Die Problematik* angesprochen, sorgt der traditionelle Ansatz der Vererbung für diverse Schwierigkeiten bei der Umsetzung einer Spiele-Engine. Darunter fallen Dinge wie die Allokation neuer Daten, die niemals verwendet werden oder die Mehrfachvererbung. Während letztere in manchen Programmiersprachen gar nicht erst eingesetzt werden kann, sorgt der Aufbau des äußerst schnell wachsenden Klassendiagramms im späteren Verlauf eines Projekts für Verwirrung und Diskrepanzen zwischen Programmierern und Designern.

Terence Cohen rät daher dazu einen Schritt zurückzutreten und die Gesamtheit der Architektur von einem anderen Blickwinkel zu betrachten. Er empfiehlt einen Lösungsansatz, den das Studio für die Umsetzung der Resistance-Reihe eingesetzt hatte: Die dynmaische Erstellung von Entitäten im Spiel, während der Laufzeit durch die Komposition von verschiedenen Komponenten. Hierbei werden sämtliche Daten in kleinere Teile getrennt, welche nun vom Spiel transformiert werden. Diese Bildung von bestimmten Zuständen eines Objekts anhand seiner Komponenten bietet eine sehr natürliche Herangehensweise. Ein Beispiel hierfür ist unter anderem die Verwendung einer *State Machine*, welche den aktuellen Zustand des Spiels beschreibt, wie etwa in welchem Level sich der Spieler gerade befindet.

Er nennt diesen Ansatz das “Dynamische Komponentsystem”, welches inzwischen in vielerlei Projekten eingesetzt wird. Hierbei ist es nicht zwingend notwendig den bestehenden Code in das neue System zu implementieren. Der bisherige auf Vererbung aufgebaute Programmcode kann folglich neben den Komponenten bestehen. Für letztere wird zunächst eine Basisklasse wie etwa “BaseComponent” erstellt. Dabei werden diese via Pools zur Verfügung gestellt. Das bedeutet, dass beim Start der Applikation eine gewisse Anzahl von Komponenten vorab initialisiert wird, die während des Spielverlaufs dynamisch verwendet und wieder zurückgegeben werden. Es werden zudem die verwendeten Instanzen nach Typen getrennt aufbewahrt, um auf diese später schneller zugreifen zu können. All dies hat eine hohe Performance zur Folge und ermöglicht es die diversen Operationen in einer kurzen, konstanten Zeit zu erledigen. Dazu zählt zum Beispiel die Typenbestimmung sowie die Allokation und Deallokation von Objekten. Auf diese Weise können außerdem sämtliche Komponenten des gleichen Typs hintereinander aktualisiert werden. Nicht nur, dass dieser Vorgang äußerst Cache-freundlich ist, er ermöglicht auch die Verwendung von Multithreading um den Prozess zu beschleunigen. Indem die voneinander getrennten Listen mit Instanzen partitioniert werden, können diese den verschiedenen Threads zur Abarbeitung übergeben werden. Die Kommunikation zwischen Komponenten erfolgt dabei über Handles.

Die eigentliche Dynamik entfaltet das System erst in der praktischen Anwendung der Komponenten während der Laufzeit. Dadurch dass Entitäten aus verschiedenen Einzelteilen zusammengesetzt werden, können sich diese jederzeit den aktuellen Gegebenheiten des Spiels anpassen. Auf diese Weise wird kein Ballast aus vorherigen Spielverläufen angehäuft. Dies lässt sich unter anderem dadurch umsetzen, indem alle allokierten Komponenten zum aktuellen Zeitpunkt auch tatsächlich genutzt werden. Dieser Ansatz sorgt folglich für häufige Allokationen, die dank des Pools schnell und gleichmäßig vonstatten geht. Dabei wird lediglich auf die Verfügbarkeit einer Komponente geprüft. Sollte dies nicht der Fall sein, so wird der Pool vergrößert sowie eine neue Komponente erstellt und initialisiert.

3 Vor- und Nachteile der Komponenten und Data-Driven Game Objects

3.1 Vorteile

Die Verwendung einer komponentenbasierten Engine bringt zahlreiche Vorteile mit sich. Diese unterscheiden sich jedoch je nach Ansatz, denn eine allgemein gültige Implementierung existiert nicht. Eine genauere Analyse dieser Ansätze befinden sich in Kapitel 5.

Verfolgt man den Ansatz von Nick Prühs [1] bei dem sämtliche Komponenten ein System als Gegenstück besitzen, so lassen sich konkret vier Vorteile feststellen. Einer davon ist, dass die Aktualisierungsreihenfolge dank der Verwendung von Systemen für die verschiedenen Komponententypen, wie zum Beispiel einem Rendersystem, einem Physiksystem oder einem Soundsystem, ganz genau festgelegt wird. Im Gegensatz zu Engines bei denen die Komponenten selber für die Berechnungen und die Aktualisierung des Spielgeschehens verantwortlich sind, werden hierbei lediglich die Methoden der entsprechenden Systeme in der korrekten Reihenfolge vom Spiel veranlasst. Folglich kommt es zu keinen Schwierigkeiten bei Data-Driven Game Objects, wenn zum Beispiel alle Entitäten der Reihe nach aus einer XML-Datei geladen werden und eine Physik-Komponente vor einer Animations-Komponente definiert ist. Die Reihenfolge des Aktualisierungsvorgangs ist stets die gleiche.

Das Memory Management einer Spiele-Engine findet für sich ebenfalls einige Vorteile in einer komponentenbasierenden Architektur. Durch die Verwendung eines Pool-Allokators lassen sich Komponenten jederzeit wiederverwenden, ohne zu große Speicherkosten zu verursachen. Bei der Freigabe werden diese in einem Pool mit anderen freien Komponenten gespeichert, anstatt diese zu deallokieren. Eine Performance-Steigerung und einer Verringerung der Speicherfragmentierung sind die Folge.

Mithilfe von Multithreading existieren zwei Möglichkeiten die Abarbeitung der Komponenten in den Systemen aufzuteilen. Einerseits können die Systeme selbst in jeweils einem Thread aktualisiert werden. Dies hat jedoch zur Folge, dass es zu Problemen bei der Synchronisierung kommt. Zum Beispiel wenn das Animationssystem die Position des Spielobjekts ändert und dies der Physikberechnung in die Quere kommt. Dadurch wird der vorhin angesprochene Vorteil von der Aktualisierungsreihenfolge aufgehoben. Weiters ist die Anzahl der maximal möglichen Threads von der Menge der verschiedenen Systeme abhängig. Deshalb besteht unter anderem die Möglichkeit, dass ein System die Liste der Komponenten auf verschiedene Threads aufzuteilt. Zum Beispiel werden bei der Aktualisierung des Physiksystems die 100 Komponenten auf zehn verschiedene Threads

aufgeteilt. Jedoch ist auch hierbei auf die Synchronisation zwischen den einzelnen Objekten zu achten. Zum Beispiel wenn es passend zum vorherigen Beispiel zur Kollision kommt.

Der vierte und von Prühs letzte genannte Vorteil betrifft die Leichtigkeit der Serialisation. Das Zusammenspiel von Data-Driven Game Objects und einer komponentenbasierten Spiele-Engine ermöglicht es dem Entwickler, sämtliche Level-Daten in einer Datei oder Datenbank zu speichern. Dadurch dass eine Entität lediglich als ID existiert und die Komponenten rein aus Daten bestehen, können diese relativ einfach bearbeitet und abgespeichert werden. Dies ist mit ein Grund warum dieser Aufbau in zahlreichen MMOGs verwendet wird. Hierbei wird für jeden Komponententyp eine Datenbanktabelle erstellt und die Komponenten mit all ihren Werten sowie der Entity-ID abgespeichert.

3.2 Nachteile

Zwar bieten die Ansätze zu komponentenbasierten Spiele-Engines und Data-Driven Game Objects zahlreiche Vorteile, doch Spieleentwickler haben ebenso mit den Nachteilen eines solchen System zu kämpfen.

Einer der deutlichesten Nachteile ist der wesentlich höhere Komplexitätsgrad einer Anwendung nach diesem Modell. Dazu kommt es unter anderem auch, weil es die diverseste Implentierungsarten für den Einsatz eines komponentenbasierten Systems gibt. Dabei hat sich der Programmierer mit einer exakten Planung und einem großen Aufwand zum Beginn des Projekts auseinanderzusetzen. Dies erfordert es, dass sich sämtliche Entwickler der Engine intensiv damit befassen bzw. sich bei einem späteren Einstieg in das Spieleprojekt zunächst für einen gewissen Zeitraum einarbeiten. Dadurch ist unter anderem auch die Gefahr für den leichtsinnigen Einsatz von Abkürzungen gegeben. In diesem Fall wird zum Beispiel in einer Render-Komponente direkt auf die Positions-Komponente verwiesen, um sich die Verwendung von Handles, Events oder anderen Kommunikationskanälen zu ersparen. Doch dies führt unweigerlich zu einem Verlust der Dynamik und bereitet folglich Probleme bei der Verarbeitung der Komponenten durch mehrere Threads oder bei der Speicherung mithilfe des Data-Driven-Designs. Im Rahmen dieser Erklärungen liefert Nick Prühs [1] folgendes Beispiel:

> "If you think about making your HealthComponent reference a VisualEffectComponent for displaying a nice particle effect every time our fellow knight is hit, remember that there could be several VisualEffectComponents attached to the knight... In most cases, just firing an event and making the event system do its job is still the best idea." *Nick Prühs, npruehs.de, 2012* [1]

Frei übersetzt bedeutet dies, dass bei der Verwendung einer Lebens-Komponente, sogleich auf eine Effekt-Komponente verwiesen werden kann, damit diese einen schönen Partikeleffekt abspielt, wenn ein Ritter getroffen wird. Jedoch kann dieser Ritter auch mehrere und andere Effekt-Komponenten besitzen. An dieser Stelle wäre es daher ratsam, auf ein Event-System zurückzugreifen, welches diese Aufgabe erledigt.

Eine weitere Herausforderung stellt die Auswahl der Komponenten dar. Es gilt bei der Anzahl der verschiedenen Komponentenarten sparsam vorzugehen, um eventuelle Performance-Verluste zu vermeiden. Es besteht folglich eine Ähnlichkeit zum Vererbungsmodell, bei dem auf die Typen der verschiedenen Unterklassen im selben Maße zu achten ist.

Die Implementierung einer solchen komponentenbasierten Architektur und der Data-Driven Game Objects beansprucht folglich auch einen größeren Zeitraum bei der Umsetzung des Projekts. Dies trifft jedoch nur auf den Beginn der Entwicklung zu. Zwar dauert es seine Zeit bis konkrete Ergebnisse vorgelegt werden können, doch langfristig gesehen kommt es dadurch zu einer deutlich Zeitersparnis, wie es Nick Prühs anhand seiner Erfahrungen mit Game Jams bestätigt [1].

Die Verwendung einer solch aufwändigen Architektur hat je nach Implementierungsart auch seine Kosten, wie es Mick West bestätigt [4]. Dadurch dass jedes Komponente von einer gemeinsamen Basisklasse erbt, kommt es zur Verwendung von virtuellen Funktionen, welche auf den ersten Blick für einen Overhead sorgen. Jedoch gilt es, das Schema beizubehalten, da sich dies durch die wesentliche Vereinfachung der Gesamtstruktur ausgeleicht. Stattdessen wird es durch die gemeinsame Schnittstelle einfacher möglich, diverse Debug-Prozesse in sämtliche Logik des Spiels zu implementieren.

Weiters spricht der Neversoft-Entwickler aus Erfahrung, wenn er davon berichtet, dass es durch die komplizierte Referenzierung zwischen Komponenten über eine eigene Komponentenverwaltung zu einem entsprechenden Performance-Verlust kommt. Deshalb habe man sich gegen Ende eines Projekts dafür entschieden, die Verweise direkt in die Komponenten selbst zu platzieren.

Bezüglich der Erstellung von Entitäten via Daten-Driven-Design kommt es unter Umständen im Laufe eines Projekts zu Problemen mit der Aktualisierungsreihenfolge. Wenn zum Beispiel die XML-Datei zunächst eine Physik-Komponente beinhaltet und erst später die hinzugehörige Animations-Komponente erstellt wird, kommt es zu Problemen bei der Synchronisation zwischen den beiden. Hierbei empfiehlt sich jedoch die von Nick Prühs erwähnte Variante [1], dies mithilfe von einzelnen Systemen für die verschiedenen Komponenten zu umgehen. Ebenfalls bietet sich der Ansatz von Terence Cohen an [2], welcher die allokierten Objekte voneinander trennt und geordnet nach Typ aktualisiert.

4 Analyse der Unity3d Engine

Ein zurzeit äußerst bekanntes Beispiel einer komponentenbasierten Engine ist Unity3D. Dabei handelt es sich um eine 3D-Spiele-Engine, die dem Entwickler sämtliche Funktionalitäten zur Erstellung von hochwertigen Spielen zur Verfügung stellt. Unity3D besteht ebenfalls aus einem umfangreichen Editor, der es Level Designern und Skriptern ermöglicht, direkt mit der Engine zu arbeiten. Die Projekte lassen sich dabei jederzeit mithilfe eines einfachen Klicks auf den Start-Knopf direkt im Editor testen. Es werden zahlreiche Ressourcen mitgeliefert, sodass von Haus aus ohne großem Aufwand einfache Spieleprojekt erstellt werden können.

Als Skriptsprachen werden C# in Form von Mono, eine erweiterte Version von Javascript, welche auch Unityscript genannt wird, und Boo angeboten. Als Standard-Entwicklungsumgebung wird Mono Develop sowie ein Texteditor namens UniScite mitgeliefert. Bei der Verwendung von C# können zahlreiche Bibliotheken aus .NET mitverwendet werden, wie zum Beispiel LINQ oder Generic Collections. Die Engine selbst wurde allerdings in C++ programmiert.

Wie aus der PR-Kategorie der Unity-Webseite [5] und dessen Fast Facts hervorgeht, stammt die erste Version von Unity3D aus dem Jahre 2005 und wurde auf Apples hauseigener Entwicklerkonferenz präsentiert. Unity legt seinen Schwerpunkt auf die Entwicklung von Spielen für mehrere Plattformen gleichzeitung und ermöglicht es dem Anwender Projekte für PC, Mac, Web Browser, Android, IOS und viele weitere Plattformen zu erstellen. 2009 wurde auf der Unity3D-Entwicklerkonferenz namens *Unite* mit Version 2.6 die erste frei verfügbare Version vorgestellt. Dadurch ist es Indie-Entwicklern möglich ohne Lizenzkosten anspruchsvolle Spiele mithilfe der Engine und des Editors zu veröffentlichen.

Ab einem jährlichen Umsatz von über 100.000 US-Dollar ist der Spieleentwickler jedoch dazu verpflichtet eine Pro-Lizenz für 1.500 US-Dollar zu erwerben. Diese liefert allerdings auch zusätzliche Features wie etwa einen umfangreichen Profiler, der es dem Entwickler ermöglicht, Performance-Probleme und den Speicherverbrauch genauestens zu debuggen. Weiters beinhaltet diese Version *Level of Detail*, Echtzeit-Schattenberechnung, *Post-Processing*-Effekte und vieles mehr. Von Unity wird zudem eine spezielle Team-Lizenz angeboten, welche es einer Gruppe von Spieleentwicklern vereinfacht gleichzeitig an einem Projekt zu arbeiten. Dies geschieht mithilfe von eigenen Unity-Tools wie dem Asset Server oder dem Cache Server. Mit Versionsverwaltungs-Programmen wie SVN oder Git ist es jedoch möglich, ohne diese Lizenz den Ordner-Inhalt zu synchronisieren. Hierfür stellt Unity die Option bereit, sämtliche Verknüpfungen zwischen den Komponenten in eigenen .meta-Dateien zu speichern.

4.1 Einsatzgebiete

Aufgrund der Möglichkeit Spieleprojekte für viele verschiedene Plattformen zu veröffentlichen, steht es dem Entwickler relativ frei, für welche Plattform bzw. für wieviele Plattformen er sich entscheidet. Dabei liegt der Fokus jedoch nicht nur auf Spielen, sondern auch auf Simulationen und sogenannten Serious Games. Bei letzteren handelt es sich zum Beispiel um Trainingsprogramme für das Militär oder Lernsoftware für die Ausbildung in verschiedensten Bereichen.

4.1.1 Videospiele

Der Hauptaugenmerk der Unity3D-Engine liegt auf der Entwicklung von Videospielen für verschiedene Plattformen. Zu den Vertretern der Unity-Spiele zählen unter anderem Rovios *Bad Piggies* oder auch MMORPGs wie *Three Kingdoms Online* von NDOORS. Alleine auf der Webseite von Unity3D werden insgesamt 48 verschiedene Spiele vorgestellt, welche auf den verschiedensten Plattformen erschienen sind.

4.1.2 Simulationen

Mit dem *Unity Pro Modeling Simulation and Training Bundle* oder auch kurz Unity MS&T [6], bietet Unity eigene Angebote für die Sparte der Serious Games an. Dieses Paket beinhaltet neben der Pro-Edition inklusive sämtlichen Mobil- und Team-Funktionalitäten von Unity3D auch die Verwendung des Pakets zur Importierung von GIS-Terrain-Daten. Dieses Plugin ermöglicht es dem Anwender komplexe Terrain-Strukturen in Unity-Terrain-Objekte umzuwandeln und als 3D-Modelle im .obj-Format abzuspeichern. Dabei können auch mehrere einzelne Gebiete auf einmal importiert und zusammengefügt werden.

Ein weiterer Bestandteil von Unity MS&T ist das *SCORM Integration kit*. Es handelt sich hierbei um ein Plugin von der *Advanced Distributed Learning Initiative*, welches ein *Sharable Content Object Reference Model* anbietet. Mithilfe dieses Tools lassen sich Punkte, Aufgaben und Interaktionen festlegen, die dazu verwendetet werden können, Lerninhalte einfach und in verschiedenen Umgebungen wiederzugeben. Ein Beispiel für den Einsatz von Unity3D im Bereich der Simulationen ist das Projekt TerraViz, welches unter anderem für die *Federal Virtual Worlds Challenge* 2012 entwickelt wurde. Dabei handelt es sich um ein plattformunabhängiges Tool zur Daten-Visualisierung. TerraViz ist einsetzbar auf Desktop-PC, Web-Browsern sowie auf mobilen Geräten. Bei der Arbeit mit der Unity3D-Engine wurde eine Software entwickelt, welche Daten in einer tatsächlichen 3D-Umgebung darstellen kann. Zu den Zielen des Projekts zählte unter anderem die Erforschung von Benutzer-Eingaben abseits von Tastatur und Maus sowie die Untersuchung von Visualisierungs-Strategien zur Kommunikationen zwischen Echtzeit-Informationen. Konkret bedeutet dies, dass zum Beispiel mithilfe von Gesten verschiedene Informationen über bestimmte Bereiche der Erde wie etwa Umweltstatistiken interaktiv dargeboten werden.

4.2 Aufbau

Bei der Unity3D-Engine und dem inkludierten Editor handelt es sich um Werkzeuge, die es auf intuitive Art und Weise ermöglichen, ein Spiel für diverse Plattformen zu entwickeln. Doch auch für professionelle Studios eignet sich Unity, da es aufgrund seines komponentenbasierten Ansatzes ohne Probleme möglich ist, auch in größeren Teams an Projekten zu arbeiten. Zum Beispiel liefern Programmierer die notwendigen Komponenten, während nach und nach die diversen Assets von Animatoren, Grafikern und Musikern in das Projekt geladen werden. In der Zwischenzeit ist es dem Level Designer bereits möglich mit dem Bau des eigentlichen Levels zu beginnen. Ermöglicht wird dieser Ablauf durch die dynamische Erstellung von Spielobjekten sowie den diversen Skriptsprachen, die sowohl für einfache Aufgaben als auch für komplexe Systeme eingesetzt werden können.

4.2.1 Komponenten

Komponenten bestimmen das Aussehen, das Verhalten sowie die Geräuschkulisse eines Spiels. Da sie jederzeit hinzugefügt, entfernt oder im Editor bzw. durch ein Skript bearbeitet werden können, bieten sie eine Möglichkeit zur dynamischen Erstellung von Spielobjekten im Rahmen der Engine. Unity stellt von Haus aus verschiedene Komponenten wie etwa Collider, Rigidbodies, Mesh-Renderer und Lichtquellen bereit. Selbst bei Skripten handelt es sich um einzelne Komponenten die per Drag & Drop auf eine Entität gezogen werden.

Diese Entitäten werden in Unity3D *"Game Objects"* genannt und verfolgen dabei einen eher naiven Implementierungsansatz. Grund dafür ist, dass mit den Game Objects die Entitäten selbst als Klasse vorhanden sind und hierbei direkt auf zahlreiche Komponenten wie etwa den Renderer oder den Rigidbody verweisen. Dies hat jedoch den Nachteil, dass der Programmierer bzw. Skripter stets auf Null-Werte zu prüfen hat und sich stets eine Ansammlung von Null-Referenzen auf dem Objekt befinden, wie es bereits in Kapitel 2.1.2 angesprochen wurde. Ein weiterer Negativpunkt ist, dass jedes Game Object immer exakt eine Transform-Komponente besitzt. Dabei handelt es sich um die Position, Rotation und Skalierung eines Objekts, welche stets angegeben werden müssen. Diese Komponente ist bei bestimmten Spielobjekten wie etwa Skript-Containern oder *Directional Lights* (= gerichtetes Licht) theoretisch nicht notwendig und verbraucht daher in diesem Fall unnötig Speicher-Ressourcen. In der Dokumentation von Unity [7] wird dementsprechend erklärt, dass ein Game Object ohne Transformation nicht in der Szene bzw. in der Editor-Ansicht von Unity3D existieren könnte. Ein weiterer wichtiger Punkt ist die Eltern-Kind-Beziehung zwischen den Spielobjekten, welche ebenfalls über die Transform-Komponente stattfindet.

4.2.2 Scripting

Wie bereits eingangs von diesem Kapitel erwähnt, unterstützt Unity3D drei verschiedene Skriptsprachen mit welchen der Entwickler die Spiel-Logik erstellt und direkt auf andere Komponenten zugreift. Dabei handelt es sich um C#, Javascript/Unityscript sowie Boo, wobei letzteres lediglich sehr selten eingesetzt wird, da oftmals bereits die Kenntnisse der beiden anderen Sprachen

vorhanden sind[1].

C# als Skriptsprache

Mithilfe der Laufzeitumgebung Mono wird es dem Skripter oder Programmierer ermöglicht Microsofts Programmiersprache C# als Skriptsprache zu verwenden. Dank Mono und den diversen Produkten der Firma Xamarin ist es weiters kein Problem sämtlichen Code welcher mit C# erstellt wurde, auf iOS oder Android zu kompilieren. Als IDE wird standardmäßig MonoDevelop mitinstalliert und ausgeführt. Es werden im jeweiligen Projekt-Ordner jedoch auch Microsoft Visual Studio Solutions erstellt, die sich ohne Probleme verwenden lassen. Allerdings wird der Unity Debugger nur mit MonoDevelop mitgeliefert, weshalb Benutzer von Visual Studio bis auf weiteres auf dieses Feature zu verzichten haben.

Der Aufbau eines C#-Skripts sieht prinzipiell wie folgt aus.

```
using UnityEngine;
using System.Collections;

public class MyBehaviour : MonoBehaviour {

        // Use this for initialization
        void Start () {

        }

        // Update is called once per frame
        void Update () {

        }
}
```

Eine Klasse welche von *MonoBehaviour* erbt, wird von der Unity3D-Engine als spezielles Skript behandelt. Wenn diese Skript-Komponente einer Entität hinzugefügt wird, werden im Spiel automatisch diverse Callback-Methoden aufgerufen. Wie oben im Skript zu sehen ist, handelt es sich hierbei zum Beispiel um eine Start-Methode, die zu Beginn der Anwendung ausgeführt wird oder eine Update-Methode, welche im Laufe jedes Frames im Spiel aufgerufen wird. Unity bietet zahlreiche weitere dieser Callback-Methoden an, wie etwa zur Erkennung von Kollisionen. Es besteht jedoch keine Pflicht für die Verwendung der MonoBehaviour-Basisklasse, womit auch komplexere Systeme und Patterns sowie Interfaces eingesetzt werden können.

Unity selbst stellt im Rahmen seiner UnityEngine-Bibliothek zahlreiche Funktionalitäten zur Berechnung von Physik, Mathematik sowie zur Verwaltung der Komponenten und Game Objects zur Verfügung. Im Bezug auf das System der komponentenbasierten Spiele-Engines ist jedoch vor allem der Umgang mit den Enitäten und Komponenten interessant. Aus einem Skript welches von

[1] vgl. Will Goldstone, S. 39ff [8]

MonoBehaviour erbt, kann jederzeit auf das besitzende *GameObject* sowie einige Standard-Unity-Komponenten zugegriffen werden. Referenzen sind wiederum ebenfalls in jedem GameObject-Objekt verfügbar.

```
void MyMethod()
{
        //Zugriff auf das GameObject des Skripts.
        GameObject myGameObject = this.gameObject;

        //Zugriff auf diverse Standardkomponenten.
        Transform myTransform = this.transform;
        Renderer myRenderer = this.renderer;

        //Zugriff auf die Standardkomponenten via GameObject.
        Transform gameObjectTransform = myGameObject.transform;
        Renderer gameObjectRenderer = myGameObject.renderer;
}
```

Wie bereits angesprochen, führt dies dazu, dass sich der Entwickler beim Zugriff auf diese Referenzen stets sicher sein muss, dass diese keinen Null-Wert besitzen. Zwar vereinfacht dies die Verwendung von oft benötigten Komponenten, jedoch wird der Programmierer dadurch auch zu Fehlern verleitet. Folglich empfiehlt es sich auf Null-Werte zu prüfen und dies per Log-Meldung zu signalisieren.

Abseits der standardmäßig referenzierten Komponenten können diese wie folgt angesprochen und in einer lokalen Variable gespeichert werden.

```
void MyComponentMethod()
{
        //GetComponent-Methode, welche die gefragte Komponente
        //oder null zurückliefert.
        MeshFilter myMeshFilter = this.GetComponent(typeof(MeshFilter))
            as MeshFilter;

        //Überladung der GetComponent-Methode mit Generics.
        Transform myTransform = this.GetComponent<Transform>();
}
```

Mithilfe der generischen Methode erspart sich der Entwickler eine Konvertierung der zurückgelieferten Referenz.

Bei dem letzten wesentlichen Punkt der Komponentenverwaltung in Unity handelt es sich um das Hinzufügen bzw. Entfernen von Komponenten.

```
void MyOtherComponentMethod()
{
        //Füge neue Komponenten (auch Skripte) zu dem GameObject hinzu.
        MeshRenderer myRenderer =
                this.gameObject.AddComponent<MeshRenderer>();
```

```
        AnotherBehaviour myOtherBehaviour =
                this.gameObject.AddComponent<AnotherBehaviour>();

        //Entferne Komponenten von diesem GameObject.
        Destroy (myRenderer);
        Destroy (myOtherBehaviour);
}
```

Komponenten können ebenfalls per nicht-generischer Methode hinzugefügt werden, erfordern bei der Zuweisung allerdings eine Konvertierung. Anstelle der Verwendung einer ähnlichen Methode wie etwa "RemoveComponent" greift Unity auf eine Destroy-Methode zurück. Diese wird auch für die Entfernung von Game Objects aus der aktuellen Szene während eines Spiels verwendet.

Unityscript

Bei Unityscript handelt es sich um eine Abwandlung von Javascript, die speziell für das Scripting in der Unity3D-Engine entwickelt wurde. Einer der wesentlichsten Unterschiede zwischen den beiden Sprachen ist die Verwendung von Klassen. Während bei Javascript mit Prototypen gearbeitet wird, bietet Unityscript Klassen und Vererbung an. Weiters ist es möglich Variablen zu typisieren und die Sichtbarkeit mithilfe der Schlüsselwörter *public, protected* und *private* genauso wie bei Methoden per Kapselung zu bestimmen[9].

In Unityscript werden Klassen automatisch mithilfe des Dateinamens erzeugt und erben von der Basisklasse *MonoBehaviour*. Daher ist es nicht notwendig dies wie bei C# explizit anzugeben. Der Aufbau eines neu erstellten Skripts mit Unityscript sieht wie folgt aus.

```
#pragma strict

function Start () {

}

function Update () {

}
```

Der wesentlich schlankere Stil des Codes rückt den Fokus mehr auf das eigentliche Skripten der Spiel-Logik. Für die Umsetzung von komplexen Systeme von erfahrenen Programmierern empfiehlt sich daher die Verwendung von C#. Dies hat nicht zuletzt den Grund, dass dadurch Features wie etwa Interfaces, abstrakte Klassen oder Generics sowie externe Bibliotheken eingesetzt werden können.

Die Verwendung von Komponenten verändert sich in der Unityscript-Syntax nur geringfügig.

```
function MyMethod()
```

```
{
        //Zugriff auf das GameObject des Skripts.
        var myGameObject : GameObject = this.gameObject;

        //Zugriff auf diverse Standardkomponenten.
        var myTransform : Transform = this.transform;
        var myRenderer : Renderer = this.renderer;

        //Zugriff auf die Standardkomponenten via GameObject.
        var gameObjectTransform : Transform = myGameObject.transform;
        var gameObjectRenderer : Renderer = myGameObject.renderer;
}

function MyComponentMethod()
{
        //GetComponent-Methode, welche die gefragte Komponente
        //oder null zurückliefert.
        var myMeshFilter : MeshFilter = this.GetComponent("MeshFilter");

        //Alternative zur Angabe des Typs per string.
        var myTransform : Transform = this.GetComponent(Transform);
}

function MyOtherComponentMethod()
{
        //Füge neue Komponenten (auch Skripte) zu dem GameObject hinzu.
        var myRenderer : MeshRenderer =
                this.gameObject.AddComponent("MeshRenderer");
        var myOtherBehaviour : AnotherBehaviour =
                this.gameObject.AddComponent("AnotherBehaviour");

        //Entferne Komponenten von diesem GameObject.
        Destroy (myRenderer);
        Destroy (myOtherBehaviour);
}
```

Der wesentliche Unterschied liegt bei den Überladungen und Argumenten der *AddComponent*- und *GetComponent*-Methoden. Durch den Wegfall von Generics und des *typeof*-Statements verändern sich für den Programmierer die Argumente der Methoden. Zum einen kann dieser auf die in C# ebenfalls vorhandene Überladung mit dem Typ als String zurückgreifen, zum anderen kann anstelle von *typeof* der Typ der Komponente direkt angegeben werden.

Boo

Bei Boo handelt es sich um eine objektorientierte, statische und typisierte Programmiersprache die von Rodrigo Barreto de Oliveira entwickelt wurde. Die Programmiersprache ist ebenfalls Teil der Mono-Laufzeitumgebung welche in Unity verwendet wird. Dabei greift de Oliveira auf diverse Features von Python und C# zurück. Ein Nachteil der Verwendung von Boo ist die geringe

Verbreitung und der damit mangelnde Support in der Community.

Die Syntax von Boo lehnt sich sehr stark an Python an, was sich anhand des Standard-Codes bei der Erstellung eines solchen Skripts zeigt.

```
import UnityEngine

class MyBooScript (MonoBehaviour):

        def Start ():
                pass

        def Update ():
                pass
```

Anstelle von geschwungenen Klammern werden bei Boo Doppelpunkte und die Einrückung durch Tabulatoren genutzt. Dennoch sind alle Callbacks und Methoden wie auch in C# und Unityscript ganz normal einsetzbar.

```
def MyMethod():
        #Zugriff auf das GameObject des Skripts.
        myGameObject = gameObject

        #Zugriff auf diverse Standardkomponenten.
        myTransform = transform
        myRenderer = renderer

        #Zugriff auf die Standardkomponenten via GameObject.
        gameObjectTransform = myGameObject.transform
        gameObjectRenderer = myGameObject.renderer

def MyComponentMethod():
        #GetComponent-Methode, welche die gefragte Komponente
        #oder null zurückliefert.
        myMeshFilter = GetComponent("MeshFilter")

        #Alternative zur Angabe des Typs per string.
        myTransform = GetComponent(Transform)

def MyOtherComponentMethod():
        #Füge neue Komponenten (auch Skripte) zu dem GameObject hinzu.
        myRenderer = gameObject.AddComponent("MeshRenderer")
        myOtherBehaviour = gameObject.AddComponent("AnotherBehaviour")

        #Entferne Komponenten von diesem GameObject.
        Destroy (myRenderer)
        Destroy (myOtherBehaviour)
```

Genau wie bei Unityscript hat der Programmierer hierbei auf den Einsatz von Generics zu verzichten und auf andere Überladungen bei den GetComponent und AddComponent-Methoden zurückzu-

greifen. Was beim Anblick von Boo unverzüglich auffällt, ist die extrem schlanke Syntax. Zudem wird der Ersteller des Skripts dazu gezwungen einen übersichtlichen Code-Stil zu benutzen.

5 Vergleich verschiedener Implementierungsarten

Die Implementierung einer komponentenbasierten Game Engine stellt einen der komplexeren Schritte des Konzepts dar. Es existieren viele verschieden Ansätze, die sich teilweise grundlegend unterscheiden. Eine Idealumsetzung ist nicht gegeben, weshalb jede Implementierungsart mit seinen Vor- und Nachteilen zu kämpfen hat. Nick Prühs erläutert zwei mögliche Ansätze zur Umsetzung des Modells[1]. Zudem schildert Mick West seine Erfahrung damit und berichtet von einer Möglichkeit der Impementierung [4].

Zwecks Vereinfachung sind die in den folgenden Kapiteln angeführten Codebeispiele stark zusammengefasst. Folglich findet die Implementierung der Klassen direkt in den Header-Dateien statt und auf *define*-Statements wird verzichtet.

5.1 Naive Implementierung

Hierbei handelt es sich um eine direkte Implementierung, welche auf schnelle Art und Weise Ergebnisse liefert und relativ einfach umzusetzen ist.

Bei der naiven Implementierung eines solchen Modells, setzt sich jede Komponente aus den Daten und der Logik zusammen, die es für die Durchführung seiner Aufgaben benötigt. Im Falle einer Komponente welche für die Verwaltung der Lebensenergie zuständig ist, würde die konkrete Implementierung in C++ in etwa wie folgt aussehen.

```
class HealthComponent
{
public:
        //Konstruktor. Initialisiere Lebenspunkte zu Beginn.
        HealthComponent(unsigned int startHealth)
                : m_health(startHealth)
                , m_alive(true)
        {
        }

        //Aktualisiere die Logik dieser Komponente im aktuellen Frame.
        inline void Update(void)
        {
                GUI::DrawHealthBar(m_health);
        }

        //Füge der Entität Schaden hinzu.
        inline void ApplyDamage(unsigned int damage)
        {
                if(damage >= m_health)
                {
                        m_health = 0;
                        m_alive = false;
                }
                else
                {
                        m_health -= damage;
                }
        }

        //Heile die Entität
        inline void ApplyHealing(unsigned int healPoints)
        {
                m_health += healPoints;
        }

        //Getter für aktuelle Lebenspunkte.
        inline unsigned int GetHealth(void) const
        {
                return m_health;
        }

        //Getter. Ist die Entität am Leben?
        inline bool IsAlive(void) const
        {
                return m_alive;
        }

private:
        unsigned int m_health;
        bool m_alive;
};
```

Der Aufbau dieser Klasse lässt sich in gewissere Hinsicht mit den Skript-Komponenten und *MonoBehaviours* der Unity3D-Engine vergleichen. Während jedes Frames wird die Update-Methode der Komponenten-Klasse aufgerufen. Folglich aktualisiert sich diese mitsamt ihrer Daten selbstständig. Anstelle eines Setters für die Lebenspunkte werden Methoden zur Verfügung gestellt, die diese Aufgaben erledigen.

Für die Verwaltung der Komponenten ist die Entität selbst verantwortlich. Die Klasse stellt daher Methoden zum Hinzufügen und Entfernen der einzelnen Komponenten bereit und kümmert sich um die Aktualisierung von diesen.

```
class Entity
{
public:
        //Konstruktur. Weise übergebene Entity ID zu.
        Entity(unsigned int entityId)
                : m_entityId(entityId)
        {
        }

        //Aktualisiere Komponenten
        inline void Update(void)
        {
                m_healthComponent->Update();
        }

        //Füge Health-Komponenten zur Entität zu.
        inline void AddHealthComponent(unsigned startHealth)
        {
                if(m_healthComponent == nullptr)
                {
                        m_healthComponent = new HealthComponent(
                            startHealth);
                }
                else
                {
                        cerr << "Entity: Can't add new health component -
                            already exists." << endl;
                }
        }

        //Enteferne die Health-Komponente der Entität.
        inline void RemoveHealthComponent(void)
        {
                delete m_healthComponent;
        }

        //Getter für die Health-Komponente.
        inline HealthComponent* GetHealthComponent(void) const
        {
                return m_healthComponent;
        }
private:
```

```
        unsigned int m_entityId;
        HealthComponent* m_healthComponent;
};
```

Bei genauerer Betrachtung wird schnell deutlich, dass sich diese Klasse im Laufe eines Projektes enorm vergrößern wird. Grund dafür sind der Getter sowie die Methoden zum Hinzufügen und Entfernen der jeweiligen Komponente. Mithilfe von enums, eines switches oder einer Abwandlung des Factory-Patterns lässt sich der Code sicherlich übersichtlicher gestalten. Hierdurch wird jedoch dem Hauptproblem ausgewichen: Ein in OOP erfahrener Programmierer wird feststellen, dass die Verwendung einer Basisklasse für Komponenten angebracht ist. Auf diese Weise können diese unter anderem in einer Liste oder einem std-Vector gespeichert werden. Dieser Schritt beseitigt auch ein weiteres Problem: Dadurch dass die Entity-Klasse Pointer für jede Komponente benötigt, entsteht ein Speicher-Overhead. Denn zahlreiche Entitäten enthalten in weiterer Folge mehrere Pointer die auf Null-Werte verweisen.

Eine Basisklasse oder ein Interface für Komponenten ermöglicht es ebenfalls, deren Verwaltung einer Art Komponenten-Manager-Klasse zu überlassen. Diese enthält sämtliche Komponenten, welche nach Typ geordnet sind oder sich in einer std-Map bzw. einem Dictionary mit dem Typen als Schlüsselwert befinden. Folglich können alle Komponenten des gleichen Typs der Reihe nach aktualisiert werden. Dies erleichtert unter anderem die Verwendung von Multithreading bei dem sich die Threads die verschiedenen Komponenten zur Abarbeitung aufteilen. Auf diese Weise wird daher die Synchronisation zwischen verschiedenen Komponententypen nicht gefährdet.

5.2 Entity-Systeme

Bei der Verwendung dieses Ansatzes und dem Komponent-Manager bietet sich sogleich ein weiterer Schritt an: Der komplette Verzicht auf eine Entity-Klasse und Speicherung der entsprechenden ID in der Komponent-Manager-Klasse. Letztere sieht in einer simplen Form demnach wie folgt aus.

```
class ComponentManager
{
public:
        //Destruktor. Gibt Komponenten frei.
        ~ComponentManager(void);

        //Füge eine neue Komponente für die Entität hinzu.
        void AddComponent(ComponentType type, unsigned int entityId)
        {
                m_componentList[type][entityId] = ComponentFactory::
                    Create(entityId, type);
        }

        //Enferne eine Komponente von der Entität.
        void RemoveComponent(ComponentType type, unsigned int entityId)
        {
                delete m_componentList[type][entityId];
```

```
                m_componentList[type].erase(entityId);
        }

        //Liefere die Komponente für die Entität zurück.
        BaseComponent* GetComponent(ComponentType type, unsigned int
            entityId)
        {
                return m_componentList[type][entityId];
        }
private:
        map<ComponentType, map<unsigned int, BaseComponent*>>
            m_componentList;
};
```

Auf diese Weise besteht eine Entität nur noch aus einer ID und setzt sich komplett aus der Gesamtheit ihrer Komponenten zusammen. Anstelle der Deallokation einer Komponente bei der Verwendung bietet sich unter anderem die Verwendung eines Pool-Allokators an, wodurch Komponenten erneut eingesetzt werden können. Eine Serialisation der Komponenten wird mithilfe dieses Ansatzes wesentlich erleichtert. Da es sich bei den Entitäten lediglich um IDs handelt, können diese gemeinsam mit dem Komponententyp und den Daten in einer Datei oder Datenbank gespeichert werden.

```
class BaseComponent
{
public:
        //Konstruktor. Initialisiert die Komponente.
        BaseComponent(unsigned int entityId, ComponentType componentType)
                : m_active(true)
                , m_entityId(entityId)
                , m_componentType(componentType)
        {
        }

        //Virtueller Destruktor, um Ressourcen der Kindklassen
            freizugeben.
        virtual ~BaseComponent();

        //Getter. Ist die Komponente aktiviert?
        inline bool Active() const
        {
                return m_active;
        }

        //Setter: Aktiviert/deaktiviert Komponente
        inline SetActive(bool enabled)
        {
                m_active = enabled;
        }

        //Getter. Liefert den Komponententyp zurück.
        inline ComponentType GetComponentType() const
```

```
        {
                return m_componentType;
        }

        //Getter. Liefert die Id der besitzenden Entität zurück.
        inline unsigned int GetEntityId() const
        {
                return m_entityId;
        }
private:
        bool m_active;
        ComponentType m_componentType;
        unsigned int m_entityId;
};
```

Die BaseComponent-Klasse selbst stellt Informationen bezüglich ihres eigenen Komponententyps und ihres Besitzers zur Verfügung. Zudem bietet sie eine Möglichkeit zur Aktivierung bzw. Deaktivierung der Komponente an.

Komponenten enthalten in diesem Fall keine virtuelle Update-Methode. Denn es stellt sich die Frage, wer die Verantwortung für die Logik hinter der Verarbeitung der Komponent trägt. Einen möglichst optimalen Ansatz bieten hierfür die sogenannten Entity-Systeme. Anstatt die Logik direkt in den Komponenten selbst zu implementieren, stellen diese lediglich Daten-Container dar, welche von einem System verwaltet werden. Sie bilden das Gegenstück zum jeweiligen Komponententyp und ermöglichen eine Verarbeitung und Speicherung mit, allerdings auch ohne der Verwendung eines zentralen Komponenten-Managers.

Daher bildet in weiterer Folge die Klasse *HealthSystem* das Gegenstück zur *HealthComponent* welche die Logik nun vollständig dem System überlässt.

```
class HealthSystem
{
public:
        //Konstruktor.
        HealthSystem(ComponentManager* componentManager)
                : m_componentManager(componentManager)
        {
        }

        //Füge Entität Schaden hinzu.
        void ApplyDamage(unsigned int entityId, unsigned int damage)
        {
                HealthData* data = new HealthData;
                data->entityId = entityId;
                data->healthModified = -damage;
                m_healthModificationsList.push_back(data);
        }

        //Heile Entität
```

```
        void ApplyHealing(unsigned int entityId, unsigned int
            healingPoints)
        {
                HealthData* data = new HealthData;
                data->entityId = entityId;
                data->healthModified = healingPoints;
                m_healthModificationsList.push_back(data);
        }

        //Starte System. Kopiere Liste von Komponenten.
        void Start()
        {
                m_localComponentList = m_componentManager->
                    GetComponentsOfType(HEALTH_COMPONENT);
        }

        //Aktualisiere Lebenspunkte in den Komponenten
        //und prüfe auf Änderungen in der Liste der Komponenten.
        void Update()
        {
                if(m_componentManager->
                        ComponentsChanged(HEALTH_COMPONENT))
                {
                        m_localComponentList = m_componentManager->
                            GetComponentsOfType(HEALTH_COMPONENT);
                }

                for(HealthData* data : m_healthModificationsList)
                {
                        m_localComponentList[data->entityId] += data->
                            healthModified;
                }

                for(list<HealthData*>::const_iterator it =
                    m_healthModificationsList.begin(); it !=
                    m_healthModificationsList.end(); it++)
                {
                        delete *it;
                }
                m_healthModificationsList.clear();
        }

        //Beende System.
        void Destroy()
        {
                m_localComponentList.clear();
                m_componentManager = nullptr;
        }
private:
        //Struct für Modizifierung der Lebenspunkte.
        struct HealthData
        {
                unsigned int entityId;
```

```
            int healthModified;
        };

        std::map<unsigned int, BaseComponent*> m_localComponentList;
        ComponentManager* m_componentManager;
        std::list<HealthData*> m_healthModificationsList;
};
```

Die HealthSystem-Klasse übernimmt sämtliche Aufgaben zur Berechnung der Lebenspunkte eines Spielobjekts. Aus Gründen der Performance wird die Liste der Komponenten in einer lokalen std-Map zwischengespeichert. Die Verantwortung zur Freigabe des Speichers trägt nach wie vor der Komponenten-Manager. Hierfür bietet sich jedoch auch die Verwendung von *Shared Pointern* an.

Anhand dieses Beispiels lässt sich eine weitere Herausforderung im Zusammenhang mit komponentenbasierten Engines feststellen: Die verschiedenen Systeme und Komponenten referenzieren sich nicht gegenseitig und haben daher auch keinen Zugriff aufeinander. Es ist unklar, wodurch die Methoden *ApplyDamage* und *ApplyHealing* aufgerufen werden. Weiters ist es zum Beispiel nicht direkt möglich einen Partikeleffekt bei der Heilung der Entität über das Partikel-System abzuspielen.

5.3 Leistungsorientierte Kommunikation

Die Kommunikation zwischen den einzelnen Systemen und Komponenten stellt eine Herausforderung dar. Mick West berschreibt diesen Prozess bei der Implementierung eines Spiels der "Tony Hawk's Pro SkaterReihe[4]. Im Optimalfall kennen Komponenten bzw. Systeme einander nicht direkt. In der Praxis sind diese jedoch notwendig. Dies liegt nicht zuletzt an der Performance eines Spiels, wodurch sich Komponenten direkt gegenseitig referenzieren. Dies ist auch über den Komponenten-Manager möglich, jedoch sorgt dieser für eine zusätzliche Belastung der CPU, welche vermieden werden soll. Jedoch kommt es durch die Verwendung dieses Modells zu Problemen bei der Reihenfolge und mit Multithreading, wovor Nick Prühs ausdrücklich warnt[1].

Eine Alternative in dieser Hinsicht stellen Event-Systeme dar. Auf diese Weise erfolgt die Kommunikation ohne Kenntnisse über das Gegenüber. Mithilfe einer Event-Daten-Struktur lassen sich zum Beispiel Informationen über den gewünschten Partikel-Effekt und dessen Position an das Partikel-System senden. Auch die Verwendung von Handles erleichtert die Kommunikationen zwischen den einzelnen Komponenten.

6 Zusammenfassung

Komponentenbasierte Game Engines bieten eine empfehlenswerte Alternative zum bekannten Vererbungs-Modell. So werden von ihnen zahlreiche Probleme wie etwa die Mehrfachvererbung oder eine unübersichtliche Klassenstruktur vermieden. Entitäten welche mittels Vererbung implementiert werden, können ihr Verhalten lediglich durch die Änderung des Codes und der Basisklassen erreichen. Mithilfe von Komponenten wird diese Logik innerhalb von kleinen Klassen gekapselt, welche erst in ihrer Gesamtheit eine Entität definieren. Mithilfe des Data-Driven-Designs lassen sich diese auf einfachem Wege per Text-Datei oder Datenbank von allen Projekt-Beteiligten verändern.

Quer durch die Spieleszene haben Veteranen wie Gas Powered Games mit Dungeon Siege, Neversoft mit Tony Hawk's Pro Skater oder Insomniac Games mit Resistance einen großen Gefallen an der Verwendung von komponentenbasierten Spiele-Engines gefunden. Die Vorteile umfassen dabei Dinge wie das Memory Management eines Spiels. Mithilfe eines Pools aus Komponenten werden dynamische Speicherallokationen minimiert. Weiters erleichtert die Architektur des Modells die Nutzung von Multithreading, indem Komponenten unabhängig voneinander bearbeitet werden können.

Ein Nachteil solcher Engines ist jedoch der deutlich gesteigerte Komplexitätsgrad, welcher vor allem zum Beginn des Projekts eine entsprechende Planung erfordert. Zudem gilt es bei der Auswahl der Komponenten behutsam vorzugehen, um die Größe und den Speicheraufwand der Anwendung möglichst gering zu halten. Ein weiterer Punkt ist die Kommunikation zwischen den einzelnen Komponenten, was gewissermaßen eine Herausforderung bezüglich solcher Engines darstellt.

Unity3D stellt eine der derzeit populärsten komponentenbasierten Spiele-Engines zur Verfügung. Mithilfe von Gratis-Lizenzen findet der Engine-Entwickler großen Anklang in der Indie-Entwickler-Szene vor. Mithilfe des mächtigen Editors stellt auch die Entwicklung von größeren Projekten kein Problem dar. Mit C#, Unityscript und Boo stellt Unity drei verschiedene Skriptsprachen zur Verfügung mit welchen sich Spiele für viele verschiedene Plattformen realisieren lassen. Jedoch wird die Engine auch zu Trainingszwecken und für Serious Games eingesetzt. Dafür wird mit dem *Unity Pro Modeling Simulation and Training Bundle* ein Paket für genau diese Zwecke zur Verfügung gestellt. Unitys Komponentensystem ist jedoch keineswegs optimal, dafür aber besonders benutzerfreundlich. Zahlreiche Standard-Komponenten befinden sich direkt in den *MonoBehaviour*- und *GameObject*-Klassen, weshalb hier auf Null-Referenzen geachtet werden muss.

Bei der Implementierung von komponentenbasierten Game Engines existieren viele verschiedene Ansätze. Dabei kann sich zum einen die komplette Logik des Spiels in den Komponenten selbst befinden, zum anderen bieten Systeme und Komponenten-Manager die Möglichkeit sie vollständig von den Komponenten zu entfernen. Dadurch werden sie lediglich als Daten-Container genutzt.

Eine weitere Aufgabe stellt die Kommunikation zwischen den Systemen und Komponenten dar. Die direkte Referenz empfiehlt sich aufgrund der engen Koppelung nicht. Die dadurch verursachten Probleme bei der Verwendung von Multithreading und der Ausführungsreihenfolge sind nicht empfehlenswert. Stattdessen ist die Umsetzung mithilfe des Komponenten-Managers oder eines Event-Systems vorteilhaft.

Literaturverzeichnis

[1] N. Pruehs, "Game Models - A Different Approach," http://www.npruehs.de/game-models-a-different-approach-i/, 2012, Online, [Zugang am 17.05.2013].

[2] T. Cohen, "A Dynamic Component Architecture for High Performance Gameplay," http://www.insomniacgames.com/a-dynamic-component-architecture-for-high-performance-gameplay/, 2010, Online - PDF-Präsentation inkl. Notizen, [Zugang am 17.05.2013].

[3] S. Bilas, "A Data-Driven Game Object System," http://scottbilas.com/files/2002/gdc_san_jose/game_objects_slides_with_notes.pdf, 2002, Online - PDF-Präsentation inkl. Notizen, [Zugang am 17.05.2013].

[4] M. West, "Evolve Your Hierarchy," http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/, 2007, Online, [Zugang am 05.05.2013].

[5] Unity3D, "Unity - Webseite," http://unity3d.com/, 2013, Online, [Zugang am 23.05.2013].

[6] ——, "Unity Pro MS&T Bundle," http://download.unity3d.com/company/sim/UnityPro_MS&T_Bundle_ver_1.2.pdf, 2012, Online-Broschüre, [Zugang am 23.05.2013].

[7] ——, "Unity-Dokumentation: The GameObject-Component Relationship," http://docs.unity3d.com/Documentation/Manual/TheGameObject-ComponentRelationship.html, 2013, Online, [Zugang am 24.05.2013].

[8] W. Goldstone, *Unity 3.x - Game Development Essentials*, 2nd ed. Packt Publishing Ltd., Birmingham, 2011.

[9] Unity3D, "UnityScript versus JavaScript," http://wiki.unity3d.com/index.php?title=UnityScript_versus_JavaScript, 2013, Online - Moderiertes Unity-Wiki, [Zugang am 25.05.2013].

Abbildungsverzeichnis

Abkürzungsverzeichnis

UAS	University of applied sciences
OOP	Objekt-orientiertes Programmieren
MMOG	Massive Multiplayer Online Game
ID	Identifikationsnummer
SCORM	Sharable Content Object Reference Model
GDC	Game Developers Conference