FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Unreal vs Unity

Ein Vergleich zwischen zwei modernen Spiele-Engines

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing / Software und Information Engineering

eingereicht von

Przemyslaw Gora
Matrikelnummer 0825680

Lukas Leibetseder
Matrikelnummer 1028385

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Univ.Ass. Dipl.-Ing. BSc Bernhard Steiner

Wien, 25. Oktober 2016

Przemyslaw Gora	Michael Wimmer
Lukas Leibetseder	Michael Wimmer

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Unreal vs Unity

Comparison between two modern Game-Engines

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing / Software and Information Engineering

by

Przemyslaw Gora
Registration Number 0825680

Lukas Leibetseder
Registration Number 1028385

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Univ.Ass. Dipl.-Ing. BSc Bernhard Steiner

Vienna, 25th October, 2016

Przemyslaw Gora	Michael Wimmer
Lukas Leibetseder	Michael Wimmer

Erklärung zur Verfassung der Arbeit

Przemyslaw Gora
Nordbahnstraße 6/18, 1200 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 25. Oktober 2016

Przemyslaw Gora

Erklärung zur Verfassung der Arbeit

Lukas Leibetseder
Lobaugasse 36/3, 1220 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 25. Oktober 2016

Lukas Leibetseder

Kurzfassung

Diese Bachelorarbeit beschäftigt sich mit der Gegenüberstellung zweier Game Engines der Unreal Engine 4 und der Unity 5 Engine. Bei dem Vergleich sehen wird uns die einzelnen Aspekte, die uns als wichtig erscheinen, genauer an, beschreiben diese und stellen anschließend einen Vergleich auf. Angefangen mit der Content-Pipeline, die sich mit der Verwendung von extern erstellten Inhalten beschäftigt, betrachten wir drei große Kategorien: Audio, Images und 3D-Assets. Dabei werden wir feststellen, dass Unity 5 vor allem im Audio und 3D-Assets Bereich weitaus mehr Formate beim Import unterstützt als die Unreal Engine. Zusätzlich hat man bei Unity die Möglichkeit Files direkt aus 3DModellingtools wie Maya in das Projekt einzubinden, allerdings mit Vor- und Nachteilen. In einigen Fällen wird man dennoch oft auf den normalen Import von FBX-Dateien zurückgreifen. Bei der Verwendung der Assets innerhalb der Engine bietet Unreal dem Benutzer einen wesentlich umfangreicheren Spielraum. Im nächste Kapitel betrachten wir die angebotenen Features beider Engines. Beide bieten hier ein umfangreiches Repertoire an Tools, die einem Entwickler in den unterschiedlichsten Bereichen unter die Arme greifen. Auch bei den Features werden wir sehen, dass die Unreal Engine ein umfangreicheres und ausgefeilteres Angebot bietet.

Im Anschluss wird ein einfaches Beispielprojekt in beiden Engines umgesetzt, um die Usability zu vergleichen und die Herangehensweise zu demonstrieren. Das Beispielprojekt wird die Erstellung eines Levels, eines Spielercharakters, des Userinterfaces sowie Spiele-Logik beinhalten. Die Logik wird in Unity 5 mittels eines C#-Scripts beschrieben, im Gegensatz zur Unreal Engine 4, wo Visual-Scripting zum Einsatz kommt. Im Anschluss betrachten wir noch die Vor- und Nachteile dieser beiden Scriptsysteme.

Im weiteren Verlauf werfen wir einen Blick auf die Effekte-Liste aus LVA UE Computergraphik (186.831) und überprüfen deren Verfügbarkeit in beiden Engines.

Im letzten Kapitel wird auf die rechtlichen Aspekte und Einschränkungen bei der Verwendung der Engines und des mit ihnen erstellten Materials eingegangen. Hierbei gehen wir unter anderem auf die Verwendbarkeit der Engines im universitären Rahmen ein.

Abstract

This bachelor's thesis focuses on the comparison of two game engines, the Unreal Engine 4 and Unity 5 Engine. We will take a closer look at the different aspects that we find important, describe and compare them. Starting with the content-pipeline, which includes the usage of externally created content, we will focus on three big categories: Audio, Images and 3D-Assets. During this process it will be shown that Unity 5 supports much more formats to import than the Unreal Engine 4. This is especially noticeable with Audio and 3D-Assets. For the latter there is a feature in Unity 5 that allows you to directly import formats of various modelling tools like Maya, although it is fair to mention that in a few cases one will be reverting to the standard way of importing FBX files. While Unreal Engine 4 doesn't have a huge support for external formats it offers more options to use the assets within the engine.

In the following chapter we will take a look at the features each engine has to offer. Both, Unreal and Unity, have a big arsenal of tools to simplify various aspects of the development process. Yet again the Unreal Engines offers a greater set of options.

Afterwards we will create a simple small project in Unreal Engine 4 and Unity 5 to demonstrate the usability and tools both engines have to offer. As we will see, the level design and placing of some objects in the editor is very similar. The interesting part starts with the creation of a controllable player character. The behaviour of such is realized differently on both sides. In Unity 5 one uses C#-scripts whereas Unreal Engine 4 offers visual scripting. We will compare those two systems and point out their pros and cons.

In the further course we will take a look at the list of effects from the lecture UE Computergraphik (186.831) and check if they are available in either of both engines.

In the last chapter, we'll take a look at the legal aspects and limitation when using Unreal and Unity. It's interesting to see how far it is possible to use those engines in university lectures.

Contents

CHAPTER 1

Introduction

Unter einer Game Engine versteht man im Allgemeinen ein Software Framework, das die Entwicklung von Videospielen oder anderen Anwendungen, die Echtzeitgrafik benötigen, ermöglicht. Darunter fallen zum Beispiel Visualisierung im architektonischen Bereich oder Simulationen. Generell besteht eine Game Engine aus einer Rendering-, Physik-, Sound- und Skript Engine sowie Funktionalitäten für Netzwerkhandling, Steuerung und Speichern des Spielzustands. Darüber hinaus verfügen Game Engines meist über zusätzliche Komponenten, welche abhängig vom Hersteller variieren können. Die Rendering Engine ist für die grafische Darstellung von Inhalten zuständig. Sie greift im Normalfall über Schnittstellen wie OpenGL oder DirectX direkt auf die Grafikkarte zu und kümmert sich um die Ausgabe und Berechnung von Meshes, Texturen und Text. Weiters ist es über Shader möglich diverse Oberflächeneffekte zu erzeugen. Die Physik Engine ist, wie der Name schon sagt, für die Physik in einem Spiel verantwortlich. Sie sorgt dafür, dass Spielobjekte unter anderem den Gesetzen der Newtonschen Mechanik folgen. Die Sound Engine hilft bei der Verarbeitung von Soundeffekten und Musik. Und darüber hinaus ist sie auch für den Raumklang verantwortlich. Die Skript Engine dient dem Umsetzen der Spiele-Logik, meist in der Form von Code. In den hier vorgestellten Game Engines wird man zwei unterschiedliche Ansätze für Script Engines sehen. Zum einen codebasiertes Skripten in der Unity Engine und zum anderen visuelles Skripten in der Unreal Engine.

1.1 Allgemeine Informationen

Die Unreal Engine 4 wurde im Mai 2012 von Epic Games veröffentlicht. Sie ist der direkte Nachfolger des UDK(Unreal Development Kit), welches als Basis die Unreal Engine 3 verwendete. Anders als die erste Version der Engine, deren Fokus First Person Shooter waren, wurde die Unreal Engine 4 als eine Entwicklungsumgebung konzipiert die in den unterschiedlichsten Bereichen eingesetzt werden kann. Von Computerspielen

über Simulationen und Film finden sich zahlreiche Gebiete, für die die Engine Tools zur Unterstützung der User anbietet.

Die Unity 5 Engine wurde im März 2015 von Unity Technologies veröffentlicht. Im Gegensatz zur Unreal Engine 4 handelt es sich bei Unity 5 um die Version 5.x einer fortwährend entwickelten Engine, die erstmalig am 8 Juni 2005 mit der Version 1.0 veröffentlich wurde.

Beide Engines wurden als Multi-Plattform Entwicklungsumgebungen konzipiert, wodurch sich in den jeweiligen Engines entwickelte Projekte auf jedem relevanten Betriebssystem deployen lassen.

In den folgenden zwei Kapiteln werden wir uns insbesondere mit der Content Pipeline und den Engine Features befassen. Dabei sehen wir uns an, wie viele Formate von extern erstellten Assets beim Import unterstützt werden, und inwiefern man auf deren Eigenschaften zugreifen kann. In unserem Kontext handelt es sich bei Assets um digitales Gut, welches von einer Person oder einem Unternehmen erstellt wurde und von dieser/diesem zu einem gewissen Preis oder auch kostenlos angeboten wird. Danach gehen wir auf die wichtigsten Features beider Engines ein und stellen jeweils einen Vergleich auf. Anschließend beschreiben wir den genauen Ablauf vom Erstellen eines einfachen Spiele-Projekts in beiden Engines. Zum Schluss gehen wir die Effekte-Liste der LVA UE Computergraphik (186.831) durch und überprüfen das Vorhandensein der Effekte in beiden Engines. Danach werfen wir noch einen Blick auf die rechtlichen Aspekte und Einschränkungen sowie die universitäre Tauglichkeit beider Engines.

1.2 UI-Übersicht

Im Grunde sind sich beide Engines vom Aufbau sehr ähnlich. Nachdem ein neues Projekt erstellt wurde, landet man im Hauptfenster der Engines, welche in den Abbildungen 1.1 und 1.2 zu sehen sind. Im unteren Bereich befindet sich der Content Browser, über den man auf die Assets und andere Projektdateien zugreifen kann. In der Mitte wird die aktuelle Szene abgebildet. Zu dieser kann man neue Objekte hinzufügen oder bereits vorhandene bearbeiten. Das Hierarchie-Fenster verschafft uns diesbezüglich einen Überblick und eine Möglichkeit auf diese zuzugreifen. Außerdem lassen sich an dieser Stelle alle Objekte in der Szene hierarchisch anordnen. Zieht man ein Objekt in der Liste über ein anderes, so wird es diesem als Komponente hinzugefügt und untergeordnet. Alle Transformationen eines Objekts wirken sich auch auf seine Komponenten aus. Standardmäßig befindet sich dieses Panel in Unity 5 oben links und in Unreal Engine 4 oben rechts. Auf der rechten Seite hat man zusätzlich ein Fenster, indem Details zu dem aktuell ausgewählten Objekt aufgelistet sind und deren Parameter verändert werden können. In Unity 5 heißt dieses Panel "Inspector" und in Unreal Engine 4 "Details".

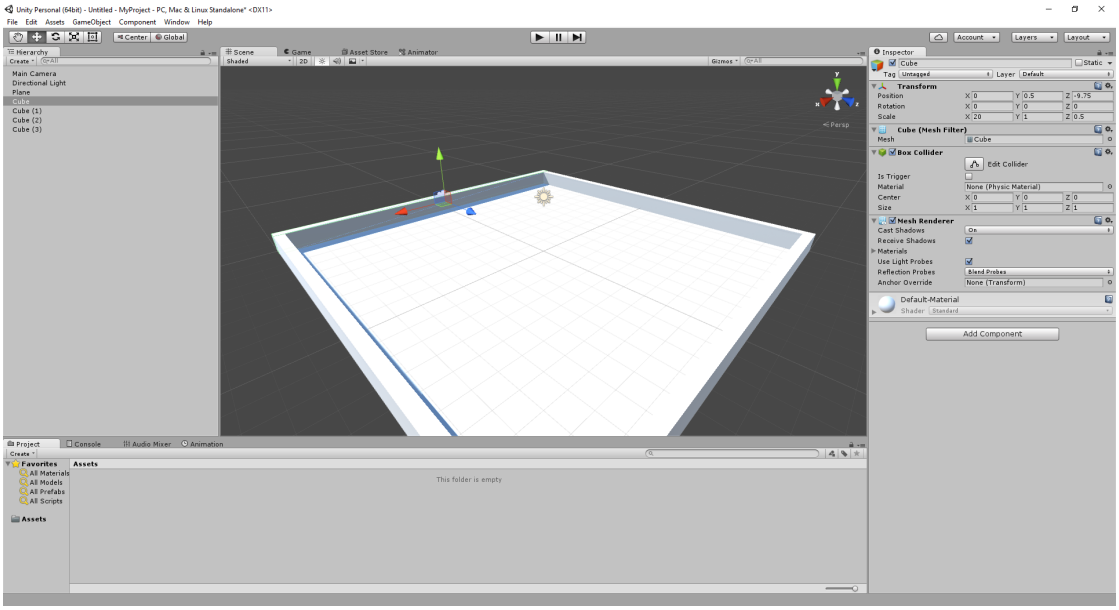

Abbildung 1.1: Unity Editor Fenster

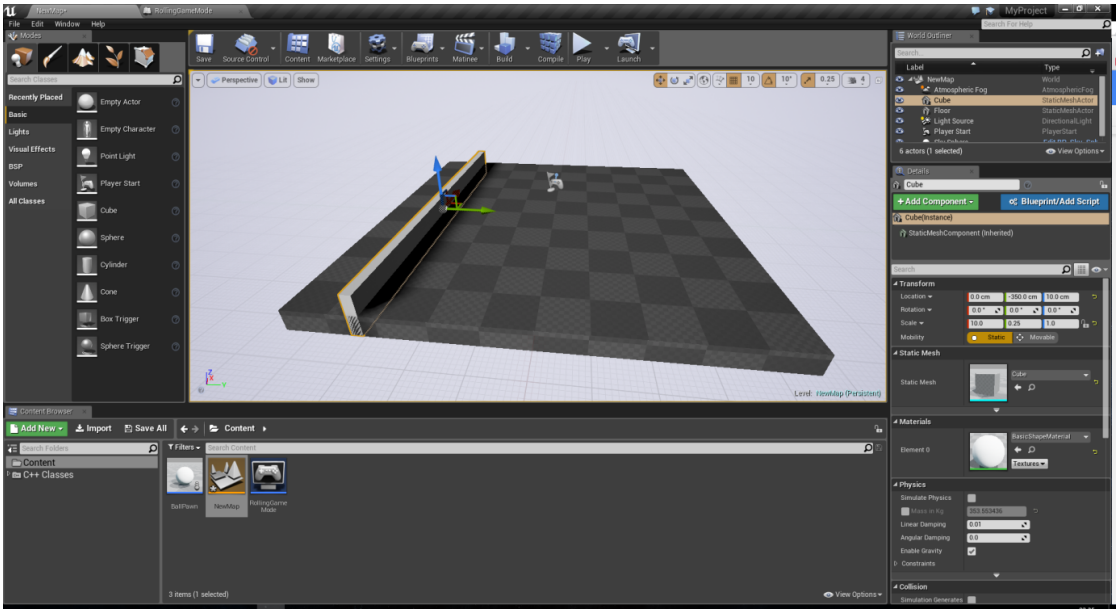

Abbildung 1.2: Unreal Editor Fenster

CHAPTER 2

Content Pipeline

Die Content Pipeline stellt einen wichtigen Bestandteil einer Engine dar. Sie ermöglicht es, verschiedene extern erstellte Assets zu importieren um sie in der Projekt-Entwicklung zu verwenden. Dabei spielt die Menge an diversen unterstützten Formaten eine große Rolle, da es dem Benutzer erspart bleibt, gewisse Umformatierungen mit Hilfe anderer Programme durchführen zu müssen. Anschließend stellt sich die Frage, in welchem Ausmaß das soeben importierte Asset innerhalb der Engine genutzt werden kann. Auf welche Eigenschaften haben wir nach dem Import Zugriff und inwiefern können diese nach Wunsch des Entwicklers modifiziert werden? Anhand von diesen Punkten werden wir in den folgenden Abschnitten dieses Kapitels einen Vergleich zwischen Unreal Engine 4 und Unity 5 aufstellen. Dabei richtet sich unser Augenmerk auf die drei wichtigsten Asset-Typen: Audio, Images und 3D-Assets.

2.1 Audio

Bei der Entwicklung von Spielen, sind Audio und Sound oft ein ausschlaggebender Faktor. Der Autor möchte eine bestimmte Atmosphäre erzeugen und den Spieler mit der entsprechenden Sound-Kulisse in eine passende Stimmung versetzen. Deshalb ist es wichtig zu überprüfen, inwieweit er diesbezüglich von der Engine unterstützt wird und welche Methoden ihm zur Verfügung stehen. Zunächst befassen wir uns mit dem Umfang des Imports von Audio-Files. Anschließend vergleichen wir die Möglichkeiten, welche uns geboten werden, um mit unseren Sound-Assets zu arbeiten.

2.1.1 Import

Beim Import treffen wir auf einen gewaltigen Unterschied zwischen den beiden Engines.

Die Unity 5 unterstützt zurzeit vier Audio-Formate; AIFF, WAV, MP3, OGG (Tabelle 2.1, zweite Spalte). Diese werden je nach Deployment-Umgebung anders verarbeitet, jedoch

Format	Unity 5	Unreal Engine 4
.mp3	x	
.ogg	x	
.wav	x	x
.aiff/ .aif	x	
.mod	x	
.it	x	
.s3m	x	
.xm	x	

Tabelle 2.1: Unterstützte Audio-Formate

gibt es bei der Codierung grundsätzlich die Optionen PCM, ADPCM und Compressed. Der Sound wird als eine Menge an aufeinanderfolgenden Sample-Werten gespeichert, die anhand der Sound-Wave und einer Abtastrate bestimmt werden. Bei PCM handelt es sich um eine fixe Abtastrate und hat daher die bessere Qualität auf Kosten von höherem Speicherverbrauch. ADPCM verwendet eine Abtastrate, welche mit Hilfe der Differenz-Werte der Samples und der Quantisierungsstufen der Sound-Wave bestimmt wird. Dadurch wird zwar die Qualität etwas beeinträchtigt, jedoch verbraucht diese Methode weniger Speicher. Compressed wird in Unity beim Deployen auf mobile Platformen eingesetzt und schneidet Sample-Werte aus um die Dateigröße zu reduzieren. Nicht zu vernachlässigen sind auch die vier Tracker Modules Formate, welche ebenfalls importiert werden können. Im Gegensatz zu normalen Audio-Files werden bei Tracker Modules nicht die Sample-Werte gespeichert, sondern die Sounds von Instrumenten und der Zeitpunkt zu dem sie abgespielt werden sollen. Aus diesem Grund steht uns auch keine Waveform als Vorschau zur Verfügung. Unterstützt werden dabei Impulse Tracker (.it), Scream Tracker (.s3m), Extended Module File Format (.xm) sowie das normale Module File Format (.mod). Tracker Modules werden bei der Spiele-Entwicklung auf Grund ihres geringen Speicherverbrauchs verwendet. Die Unreal Engine unterstützt derzeit nur das Audio-Format WAV (Tabelle 2.1, dritte Spalte).

Somit liegt Unity 5 mit 8 unterstützten Formaten weit vor der Unreal Engine 4. In beiden Fällen lassen sich Audio-Dateien mit bis zu 8 Kanälen importieren allerdings ist dieser Vorgang bei der UE4 um einiges umständlicher, da man jeden Kanal einzeln in Form eines separaten Files mit einer bestimmten Namenskonvention importieren muss.

2.1.2 Processing

Nun kommen wir zum interessanten Teil der Audio-Verwendung in den Engines. Sowohl die Unreal Engine 4 als auch die Unity 5 sind sich in dem Bereich sehr ähnlich. Der schnellste und einfachste Weg ist, ein Audio-File direkt nach dem Import in die Szene zu ziehen wodurch eine Audio-Source erstellt wird, welche als Ambient-Sound dienen kann. Des Weiteren hat man die Möglichkeit mit Hilfe einer Vielzahl an Effekten und Filtern,

das Audio-Signal zu verändern. In Unity 5 gibt es mehrere Wege dies umzusetzen: In der Audio-Source selbst, welche man in die Szene gesetzt hat, gibt es die Möglichkeit Filter als Komponenten hinzuzufügen, deren Eigenschaften man nach Belieben einstellen kann. Eine andere Variante den Sound zu verändern wäre durch die Verwendung eines Audio-Mixers, der es ermöglicht, ein oder mehrere Eingangssignale mit den gewünschten Effekten zu versehen und anschließend weiterzuleiten. Bei der Unreal Engine 4 sieht dieser Vorgang vom Aufbau ganz ähnlich aus: Über das importierte Audio-File wird ein SoundCue erstellt, welches im SoundCue-Editor bearbeitet werden kann. Hier erhalten wir Zugriff auf Eigenschaften und Werte, welche wir beliebig umstellen können, sowie eine Auswahl an Filtern und Effekten. Zu Letzteren lässt sich sagen, dass diese in beiden Engines ausreichend vertreten sind. Darunter fallen wichtige Effekte/Filter: Attenuation, Ducking, Echo, Low-/Highpass Filter und Reverb Zone.

Zusätzlich verfügt die Unreal Engine 4 bereits über einige Funktionen in Bezug auf Dialoge, welche bei der Unity 5 nicht vorhanden sind und somit selber nach Bedarf umgesetzt werden müssen. Außerdem punktet die UE4 mit der Übersichtlichkeit, je komplexer der Einsatz von Audio-Mixern und Effekten wird. Zwar bieten beide Engines die Möglichkeit Sounds und Audio zu gruppieren bzw. zu kategorisieren, jedoch behält man in der Unreal Engine 4 dank der Graphen-Darstellung im Editor leichter den Überblick.

2.2 Image

Beim Arbeiten mit Game Engines werden Bilder in Form von Texturen eingesetzt, welche auf eine 2- oder 3-dimensionale Oberfläche gemappt werden. Dabei unterscheidet man zwischen verschieden Mappings, von denen jede in gewisser Weise mehr Detail in die Szene bringt. Somit werden wir uns in diesem Abschnitt mit der Anzahl an unterstützten Bild-Formaten beider Engines beschäftigen und anschließen die weiteren Einsatzmöglichkeiten miteinander vergleichen. Wie gut lassen sich diverse Mappings umsetzen und auf welche Eigenschaften der Texturen haben wir Zugriff?

2.2.1 Import

Im Gegensatz zum großen Unterschied bei den unterstützten Audio-Formaten sind im Bild-Bereich beide Engines ziemlich gleich auf (Tabelle 2.2). Unter anderem lassen sich alle gängigen Formate wie JPG, PNG und BMP sowie Photoshop-Dateien importieren. Für Cubemaps haben wir des Weiteren die Möglichkeit .dds oder .hdr zu verwenden.

2.2.2 Processing

Nachdem eine Bild-Datei erfolgreich importiert wurde, ist sie als Textur verfügbar. In Unity 5 kann man diese nun per Klick auswählen und erhält im Inspector-Panel Zugriff auf eine Vielzahl an Einstellungen, die man nach Bedarf verändern kann. Diese Möglichkeit wird uns ebenfalls von der Unreal Engine 4 geboten mit dem Unterschied, dass die Textur in einem neuen Editor-Fenster geöffnet wird.

Format	Unity 5	Unreal Engine 4
.bmp	x	x
.png	x	x
.jpg	x	x
.tga	x	x
.psd	x	x
.float		x
.tif	x	
.pcx		x
.dds	x	x
.hdr	x	x
.exr	x	x

Tabelle 2.2: Unterstützte Bild-Formate

Im Folgenden werden wir die am häufigsten verwendeten Textur-Optionen betrachten.

- **Filtering:** Beide Engines verfügen über bilineare, trilineare und anisotropische Textur-Filterung. Letzteres lässt sich in Unity 5 mit Hilfe eines Reglers und einem Wert von 0 bis 16 einstellen. In der Unreal Engine 4 ist anisotropische Filterung prinzipiell aktiv, solange man den Filter-Mode auf "Default" lässt. Diese wird erst deaktiviert sobald man auf ein bilineare, trilineare oder nearest Filterung wechselt.
- **Mipmapping:** Im Bereich Mipmap-Generierung unterscheiden sich die Engines hauptsächlich in den Filter-Einstellungen. Unity stellt uns zwei Methoden zur Verfügung, welche die Schärfe der Mipmaps bestimmen. Unreal Engine enthält ein Dropdown-Menü mit über zehn Einstellungen. Außerdem lässt sich bei der Unreal Engine der Index des obersten Mipmap-Levels über den Punkt "LOD bias" festlegen. Zusätzlich kann man noch die Anzahl an Mipmaps bestimmen, die während eines Cinematics zum Einsatz kommen können. Eine kleine Funktion, welche Unity 5 in den Textur-Einstellungen beinhaltet ist "Fadeout Mip Maps". Mit dieser lässt sich eine Entfernung einstellen, ab der Texturen langsam ins Grau Verblassen.
- **Tiling:** Auch hier bietet uns die Unreal Engine 4 einen etwas größeren Spielraum. Abgesehen von den beiden Modi "Clamp" und "Wrap", welche in Unity 5 zur Auswahl stehen, erhalten wir eine weitere Option "Mirror" sowie die Möglichkeit, diese Eigenschaft separat für die X- und Y-Achse einzustellen.

Grundsätzlich kann man sagen, dass die Unreal Engine 4 eine deutlich größere Auswahl an Einstellungen bietet, wenn auch vieles davon seltener zum Einsatz kommt. Allerdings könnte die schiere Anzahl an veränderbaren Eigenschaften auf einige Benutzer etwas abschreckend und unübersichtlich wirken, während Unity 5 mit ihrer kompakteren Auswahl auf jeden Fall anfängerfreundlicher ist.

Format	Unity 5	Unreal Engine 4
.fbx	x	x
.obj	x	
.3ds	x	
.dxf	x	
3d-app formats	x	

Tabelle 2.3: Unterstützte 3D-Assets-Formate

2.3 3D-Assets

Im letzten Abschnitt der Content Pipeline befassen wir uns mit 3D-Assets. Dabei handelt es sich grundsätzlich um Geometrie-Daten, welche ein bestimmtes Objekt darstellen und/oder animieren sollen. Beim Level Design werden geometrische Strukturen, welche den Grundaufbau des Levels bestimmen sollen, meist direkt in der Engine mit den gegebenen Tools erstellt. Für Gebäude oder geschlossene Areale reichen oft simple Objekte wie Würfel, Zylinder und Kugeln, während für offene Landschaften Terrain zum Einsatz kommt. Im Falle detaillierterer Objekte nimmt man in der Regel externe 3D-Modellierungstools, wie Autodesk Maya oder 3DS Max, zur Hilfe. Terrain lässt sich im Prinzip ebenfalls mit diesen externen Applikationen erstellen, allerdings hat man dadurch keine Vorteile, da die Engines diesbezüglich ein ausreichend umfangreiches Set an Werkzeugen zur Verfügung stellen. Außerdem kann man in der Engine jederzeit Änderungen vornehmen und diese anschließend testen ohne die Applikation verlassen zu müssen.

2.3.1 Import

Beim Import bietet uns die Unity 5 einen größeren Spielraum bezüglich der unterstützten Formate, wie in Tabelle 2.3 in der zweiten Spalte gezeigt wird. Zusätzlich zu den Standardformaten (FBX, OBJ) haben wir noch die Möglichkeit, Dateien von diversen 3D-Applikationen, wie Maya, Max, Blender etc., direkt zu importieren. Allerdings muss dazu auch eine lizenzierte Kopie der genannten Software auf der Maschine installiert sein. Dieser Import-Vorgang erweist sich in erster Linie als sehr praktisch, zieht allerdings auf lange Sicht ein paar Nachteile mit sich. Da es sich direkt um die von externen 3D-Programmen verwendeten Formate handelt, kann man mit Sicherheit davon ausgehen, dass viele nicht brauchbare Daten mitimportiert werden, wodurch die Dateien größer und die Dauer von Aktualisierungen länger ausfallen. Diese Möglichkeit des Imports wird uns von der Unreal Engine 4 (Tabelle 2.3, dritte Spalte) nicht zur Verfügung gestellt, da sich diese fast ausschließlich auf das FBX-Format beschränkt. 3D-Assets lassen sich je nach Dateninhalt in folgende Kategorien unterteilen:

- **Static Meshes** sind geometrische Objekte welche sich ohne Weiteres nicht verformen lassen. Sie können dennoch problemlos bewegt, rotiert und skaliert werden.

- **Skeletal Meshes** sind geometrische Objekte deren Vertices an ein Skelett gebunden sind und sich abhängig davon verformen lassen.
- Mit **Animations** lassen sich ein Skelett und somit auch das Skeletal Mesh animieren.
- Mit **Morph-targets und Blendshapes** kann man Meshes verformen.
- **Materials** bestimmen das Aussehen der Oberfläche eines Meshes.

All diese Typen werden in beiden Engines unterstützt und lassen sich sowohl beim Importieren als auch danach anpassen.

Wie üblich hat man nach dem Import in Unity per Klick auf das 3D-Asset die Möglichkeit, über den Inspector auf Eigenschaften des Objekts zuzugreifen. In dem Fall handelt es sich um Import-Einstellungen, welche wir abhängig von der Art des Assets anpassen können. In der Unreal Engine stehen uns diese Einstellungen in einem separat geöffnetem Fenster zur Verfügung, allerdings nur einmalig während des Imports. Danach lassen sich diesbezüglich auch mit der Reimport-Funktion keine Änderungen vornehmen, ohne das Asset vorher zu löschen und manuell einzufügen. Die Import-Einstellungen in beiden Engines stimmen größtenteils überein. Bei einigen Punkten gibt es dennoch Unterschiede in der Anzahl oder Art von einstellbaren Parametern. Beispielsweise lässt sich in Unity das Importierte Model nur uniform skalieren, während man in der Unreal Engine die gesamte Transformation angeben kann. Zusätzlich hat man in der Unreal Engine die Möglichkeit, Mesh LODs zu importieren. Bei einem Mesh LOD handelt es sich um das Ursprungs-Model, jedoch mit einer geringeren Anzahl an Polygonen und somit einem weniger detaillierten Mesh. Ein weiteres Import-Feature in der Unreal Engine 4 ist das Erstellen oder Importieren von PhysicsAssets (Komponenten mit physikalischen Eigenschaften) für ein Skeletal Mesh.

2.3.2 Processing

Im Folgenden betrachten wir wie weit man die unterschiedlichen 3D-Assets, welche im oberen Abschnitt kurz genannt wurden, in den jeweiligen Engines weiterverwenden kann.

Static Meshes können nach dem Import direkt über den Content-Browser in die Szene gezogen werden, wo sich anschließend ihre Position, Rotation und Skalierung anpassen lässt. Des Weiteren hat man auch die Möglichkeit sie als Komponenten zu einem anderen Objekt hinzuzufügen. Die meisten Eigenschaften für ein Mesh werden in Unity bereits in den Import-Settings festgelegt, während die Unreal Engine uns zusätzlich einen eigenen Editor zur Verfügung stellt und allgemein einen etwas größeren Spielraum bietet.

Skeletal Meshes lassen sich ebenfalls nach dem Import in die Szene ziehen und anpassen. Anders als bei den Static Meshes bieten uns hier beide Engines einen Editor zum Bearbeiten des Skelettes an. In Unity wird eine Avatar-Datei erstellt, welche das

Skeletal Mesh mit seinem Skelett repräsentieren soll und zum Bearbeiten geöffnet werden kann. Dies erfolgt alles im selben Fenster, wodurch Änderungen in der aktuellen Szene vorher abgespeichert werden müssen. In Unreal Engine 4 wird das Skeletal Mesh in einem neuen Editor-Fenster geöffnet und bietet eine größere Auswahl an Einstellungsmöglichkeiten.

Animations werden in beiden Engines sehr ähnlich gehandhabt. In Unity 5 erhalten wir nach dem Import sogenannte AnimationClips, welche an Skeletal Meshes angeheftet werden können um diese zu animieren. Selbiges erreicht man mit den Animation Sequences der Unreal Engine 4. Möchte man den Wechsel zwischen mehreren Animationen kontrollieren, lässt sich dies mit Hilfe von AnimationControllers (Unity) oder AnimationBlueprints (Unreal Engine) bewerkstelligen.

Morph-targets/Blendshapes werden in Unity in der SkinnedMeshRenderer Komponente der importierten Mesh angezeigt, wo sich unter anderem die Stärke der Veränderung anpassen lässt. In der Unreal Engine kann man Morph-Targets im Editor-Fenster der importierten Mesh unter dem entsprechenden Tab betrachten.

Materials kann man wie die meisten Assets in Unity im Inspector betrachten und bearbeiten. Dort lassen sich unter anderem Image-Dateien für die verschiedenen Mappings und Masks auswählen. Falls beim Import eines Meshes keine namentlich passenden Materialien gefunden wurden, erstellt die Engine sie automatisch. In der Unreal Engine erhalten wir wiederum einen eigenen Material-Editor, der um einiges umfangreicher gestaltet ist. Allerdings kann es vorkommen, dass trotz eines erfolgreich importierten Materials der Benutzer selbst noch manuell einige Korrekturen vornehmen muss.

2.4 Fazit

Schlussendlich kann man zur Content-Pipeline der beiden Engines sagen, dass Unity eine deutlich höhere Anzahl an externen Datei-Formaten unterstützt und vor allem durch das eher einfach und kompakt gestaltete Interface besonders angenehm für Einsteiger ist während Unreal Engine 4 im ersten Moment mit ihrer überwältigen Auswahl an Asset- Einstellungen abschreckend wirkt, jedoch dem erfahrenen Nutzer deutlich mehr Spielraum überlässt.

CHAPTER 3

Engine Features

Moderne Engines wie die Unreal Engine 4 und die Unity 5 Engine bieten von Haus aus einige Tools und Effekte an, um den Benutzern das Umsetzen ihrer Projekte leichter zu machen. Bei diesen Tools handelt es sich um Hilfestellungen in Bereichen wie Audio, Physik, UI Design, Performance Profiling etc. Im folgenden Abschnitt werden wir uns mit den Features beschäftigen, die in beiden Engines standardmäßig enthalten sind und stellen anschließend, soweit möglich, Vergleiche zwischen den beiden Seiten auf.

3.1 Scripting

3.1.1 Blueprint Visual Scripting

- **Unreal**
  Der Hauptfokus des Scriptings bei der Unreal Engine liegt bei dem Visual Scripting System, das nur wenige Einschränkungen im Vergleich zum richtigen Coden in C++ besitzt. Das Visual Scripting System funktioniert, indem im Editor Nodes erstellt werden, die vorgefertigten Funktionen entsprechen. Diese können miteinander verknüpft werden um einen sogenannten "Event Graph" zu formen. Diese Visual Scripts werden intern in C++ Code übersetzt und dienen als Objektorientierte Klassen für das Projekt. Üblicherweise werden diese Scripts in der Unreal Engine als Blueprints bezeichnet. Der Vorteil der Blueprints liegt darin, dass man sie sehr leicht wiederverwenden kann.

- **Unity**
  Die Unity Engine bietet von Haus aus kein Visual Scripting.

- **Fazit**
  Das Visual Scripting der Unreal Engine ist ein sehr ausgeklügeltes System, welches

das Prototyping sehr vereinfacht. Allerdings wird man beim Umsetzen von komplexeren Algorithmen trotzdem auf C++ zurückgreifen, da in solchen Fällen das visuelle Script sehr schnell unübersichtlich wird.

3.1.2 Programming

- **Unreal**
  Die Unreal Engine erlaubt neben dem Visual Scripting System auch das direkte Programmieren in C++. Da man bei der Unreal Engine auf den Source Code zugreifen kann, hat man die Möglichkeit eigene Module zur Engine hinzuzufügen.

- **Unity**
  Das Scripting in der Unity Engine dient nur dem Umsetzen von Gamelogik und dem Verhalten von Objekten inklusive Shadern.

- **Fazit**
  Die Unreal Engine bietet zwei grundverschiedene Systeme, die sich gegenseitig vervollständigen. Das Visual Scripting ist ein guter Einstieg für Programmier-Neulinge und ermöglicht sehr schnelles Prototyping. Auf der anderen Seite hat man das C++ Programmieren, das einem Low Level Zugriff auf die Engine ermöglicht und für das Entwickeln komplexer Algorithmen und Funktionen gedacht ist. Durch diese Dualität hat man das Beste aus beiden Welten. Mit der Unity Engine hat man im Prinzip dieselben Möglichkeiten, man ist jedoch auf das C#/JavaScript Scripting beschränkt. Das stellt für Programmierneulinge eine etwas Größere Einstiegshürde dar. Die Unity Engine würde sehr von einem Visual Scripting System profitieren, da es auch das Prototyping wesentlich angenehmer macht.

3.2 Graphik

3.2.1 Partikel System

- **Unreal**
  Unreals Partikel System Tool nennt sich Cascade und ist ein Editor für unterschiedlichste Effekte wie Nebel, Rauch, Funken oder Feuer. Über den Editor kann man über diverse Parameter direkt das Verhalten der Partikel beeinflussen. Über Cascade lässt sich ein Array aus Partikel Emittern zusammenschließen, welches fortan als kombinierter Effekt eingesetzt werden kann. Dies hat den Vorteil, dass man Effekte direkt in Cascade aufeinander abstimmen kann.

- **Unity**
  Das Unity Partikel System Tool nennt sich Shuriken Particle System und ermöglicht wie auch das Unreal Pendant die Erstellung diverser Partikeleffekte. Mit dem Shuriken Particle System lassen sich nur einzelne Partikel Emitter erstellen, die über eine Vielzahl an Parametern einstellbar sind.

- **Fazit**
  Das Unreal Engine Partikel System (Cascade) bietet durch das zusammenschließen von mehreren Emittern, die direkt aufeinander abgestimmt werden können, eine genauere Kontrolle und ein insgesamt besseres Ergebnis als das Shuriken Partikel System der Unity Engine. Von den einstellbaren Parametern bieten beide Systeme dieselben Möglichkeiten. Im Endeffekt hat die Unreal Engine das ausgereiftere Partikel System.

3.2.2 Post Processing

- **Unreal**
  In der Unreal Engine wird Postprocessing über das PostProcessVolume gesteuert, da die Unreal Engine keine Postprocessing Kette verwendet. Das PostProcessVolume ist ein Bereich, der zu einem Level hinzugefügt werden kann und bietet eine enorme Anzahl von einstellbaren Effekten mit diversen Parametern, wobei in einem Bereich mehrere Effekte gleichzeitig aktiv sein können. Damit der Effekt eines PostProcessVolumes sichtbar ist, muss sich die Kamera darin aufhalten. Weiters kann man mehrere solcher Bereiche übereinanderlegen und somit viele Effekte stapeln, die je nach Bedarf aktiviert werden können.

- **Unity**
  Bei Unity werden Postprocessing Effekte zur Kamera hinzugefügt, wobei auch mehrere Effekte gleichzeitig aktiv sein können. Auch bei der Unity Engine gibt es eine enorme Auswahl an einstellbaren Effekten und Parametern.

- **Fazit**
  Die Entscheidung der Unreal Engine Entwickler auf PostProcessingVolumes zu setzen ist eine gute Idee, dadurch lassen sich sehr einfach visuelle Übergänge schaffen, während sich der Spieler durch ein Level bewegt. Die Unity Engine geht im Bereich Postprocessing den klassischeren Weg. Die Unreal Engine bietet durch ihre Herangehensweise einen größeren Spielraum beim Level Design.

3.2.3 Material Editor

Ein Material Editor dient der Erstellung und des Managements von Materials, die zu Objekten in einer Szene hinzugefügt werden können. Diesbezüglich verwenden sowohl die Unreal- als auch die Unity Engine "Physically Based Shading". Physically Based Shading versucht die Interaktion zwischen Licht und Material so realitätsnah wie möglich zu simulieren.

- **Unreal**
  Die Unreal Engine bietet einen äußerst umfangreichen Material Editor an, der in der Verwendung so gestaltet ist wie das visuelle Programmieren. Es ist sogar

möglich, dynamische Materials zu erstellen, die Leuchteffekte oder eine morphende Oberflächenstruktur bieten.

- **Unity**
  Der Material Editor der Unity Engine ist wesentlich spartanischer ausgestattet. Es lassen sich alle wichtigen Parameter einstellen, jedoch nicht mehr. Extra Shader oder besondere Material Effekte müssen über Skripte programmiert werden.

- **Fazit**
  Die Unreal Engine bietet eindeutig den mächtigeren Material Editor an. Die Visual Programming Herangehensweise erlaubt es im Handumdrehen beliebige Material Effekte zu erzeugen.

3.2.4 Lighting

Precomputed Lighting

- **Unreal**
  Im Bereich Precomputed Lighting stellt die Unreal Engine mehrere Features bereit. Lightmass Global Ilumination erzeugt Lightmaps mit komplexen Licht- und Schatteninteraktionen sowie Diffusen Interreflexionen. Das Reflection Environment Feature dient dazu, glänzende Reflexionen effizient darstellen zu können. Der Indirect Lighting Cache komplementiert die Lightmass Global Illumination. Lightmass erzeugt nur Lightmaps für statische Objekte. Für dynamische Objekte kommt der Indirect Lighting Cache ins Spiel. Er verwendet Samples, die zur Buildtime durch die Lightmass erzeugt wurden und berechnet aufgrund dieser indirekte Beleuchtung für dynamische Objekte, wie Charaktere, zur Runtime. Weiters bietet die Unreal Engine Static Lights sowie Stationary Lights. Static Lights werden nur in den Lightmaps berechnet, sie können weder bewegt noch auf irgendeine andere Weise zur Runtime verändert werden. Sie sind die einfachste Form der Beleuchtung und haben nach der initialen Berechnung keinen Einfluss mehr auf die Performance. Allerdings ist ihr Nutzen eingeschränkt, da sie zum Beispiel keinen Schatten von bewegten Objekten werfen können. Stationary Lights sind dafür gedacht, dass sie an einem Ort bleiben, können im Gegensatz zu Static Lights jedoch ihre Helligkeit und Farbe ändern. Die Unreal Engine bietet als Lichtquellen Directional Lights, Point Lights, Sky Lights und Spot Lights.

- **Unity**
  Wie die Unreal Engine stellt auch Unity verschiedenste Precomputed Light Features von Haus aus bereit. Im Bereich Global Illumination bietet Unity Baked Global Illumination Lighting, bei dem einfache Lightmaps erstellt werden, die komplexe direkte und indirekte Beleuchtung realistisch, jedoch statisch darstellen. Um Änderungen der Beleuchtung in der Szene auch mit Global Illumination wiederspiegeln zu können, bietet Unity Precomputed Realtime Global Illumination

Lighting. Über mehrere Stufen des Precompute Prozesses der Lichtberechnung werden Lightmaps erstellt, die in einem Spiel Realtime Änderungen der Beleuchtung, wie zum Beispiel ein Tag-Nacht Zyklus, darstellen können. Unity unterscheidet nicht zwischen statischen und bewegbaren Lichtquellen wie die Unreal Engine. Alle Lichtquellen, die bei der Precomputing Phase in der Szene vorhanden sind, werden zur Berechnung herangezogen. Als Lichtquellen bietet Unity Directional Lights, Point Lights, Spot Lights, und als Besonderheit im Vergleich zur Unreal Engine, Area Lights.

- **Fazit**
  Beide Engines bieten sehr ausgereifte Precomputed Lights Systeme an. Allerdings spielt die Unreal Engine mit der indirekten Beleuchtung für dynamische Objekte in einer höheren Liga.

Dynamic Lighting

- **Unreal**
  Movable Lights bieten vollkommen dynamisches Licht, dessen Eigenschaften zur Runtime verändert werden können. Sämtliche Beleuchtung durch Movable Lights wird zur Laufzeit berechnet, dadurch steigen die Performancekosten mit jedem Mesh bzw. Triangle, das zur Berechnung herangezogen wird.
- **Unity**
  Wie schon bei den Precomputed Lights erwähnt, unterscheidet Unity nicht zwischen statischen und bewegbaren Lichtquellen. Als bewegbare Lichtquellen können alle im Precomputed Bereich aufgezählten Lichtquellen verwendet werden. Für sie gelten dieselben Möglichkeiten und Einschränkungen wie bei der Unreal Engine.
- **Fazit**
  Beide Engines sind im Bereich Movable Lights gleich auf.

IES Light Profiles

IES steht für Illuminating Engineering Society. Die IES entwickelten ein Standardfileformat, das die Verteilung des Lichtes bestimmter Leuchtkörper im ASCII Format speichert. Solche Profile gibt es von allen großen Herstellern.

- **Unreal**
  In der Unreal Engine können solche Profile direkt auf Point Lights und Spot Lights angewendet werden, um für eine realistischere Beleuchtung zu sorgen.
- **Unity**
  Die Unity Engine bietet von Haus aus nichts Vergleichbares.

- **Fazit**
  Die Unreal Engine Punktet hier klar beim Realismus in Bezug auf Beleuchtung.

3.3 Landscape

3.3.1 Terrain

- **Unreal**
  Das Erstellen von Terrain funktioniert in der Unreal Engine in etwa wie das Malen in Paint. Über unterschiedliche Pinsel und andere Werkzeuge kann per Hand eine Landschaft modelliert werden.

- **Unity**
  Die Unity Engine funktioniert in Bezug auf das Erstellen des Terrains genauso wie die Unreal Engine. Auch hier wird wie in Paint mit unterschiedlichen Pinseln und Werkzeugen die Landschaft modelliert.

- **Fazit**
  Beide Engines bieten im Bereich Terrain dieselben Möglichkeiten.

3.3.2 Open World Tools

- **Unreal**
  Seit der Version 4.8 bietet die Unreal Engine zwei Tools an, die bei der Erstellung von großen Außenarealen bzw. Grünflächen helfen. Beide Tools dienen der prozeduralen Erstellung von Flora und Fauna. Die Tools sind das Procedual Foilage Tool und das Grass Tool.

  - **Procedural Foilage Tool**
    Dieses Tool dient dazu größere Strukturen wie Bäume oder Sträucher prozedural auf einem Terrain zu verteilen. Dabei kann die Verteilung über mehrere Parameter gesteuert werden. Nach dem Einstellen der Parameter wird ein Wald simuliert, der stufenweise um Jahre gealtert wird. Als Endergebnis erhält man einen möglichst realistischen Wald.
  - **Grass Tool**
    Das Grass Tool dient dazu kleinerer Strukturen wie Gras, Steine und Blumen möglichst dicht auf einem Terrain zu verteilen. Im Gegensatz zum Procedural Foilage Tool wird beim Grass Tool keine Simulation durchgeführt, sondern einfach eine vorgegebene Fläche mit einer, als Parameter angegebenen, Dichte mit einem statischen Mesh gefüllt. Als Ergebnis hat man eine dichte Grasslandschaft.

- **Unity**
  Bei der Unity Engine ist das Setzen von Foilage in die Terrain Engine integriert.

Damit ist es möglich, Gras sowie Bäume mit einer über Parameter eingestellten Dichte zu setzten. Das Setzen der Foilage funktioniert genauso wie das "Malen" des Terrains.

- **Fazit**
  Die Unity Engine bietet alles was man braucht, um eine Landschaft mit Fauna (oder Stacheldraht oder anderen Meshes) zu versehen. Die Unreal Engine geht dabei noch einen Schritt weiter und ermöglicht prozedurales setzen der Fauna. Wenn möglichst natürlich wirkende Außenareale erstellt werden sollen hat die Ureal Engine einen klaren Vorteil.

3.4 Sprites

- **Unreal**
  Paper 2D ist ein Sprite System, das bei der Erstellung von 2D und 2D/3D Hybridspielen hilft. Im Sprite Editor können diverse Parameter eingestellt werden und Sprites mit Materials versehen werden. Weiters kann die Render Geometry einzelner Sprites angepasst werden, um die Performance zu verbessern. Mit Hilfe des Paper 2D Flipbooks können individuelle Spritesequenzen zu einer 2D-Animation zusammengefügt werden. Zusätzlich hat man bei Unreal die Option, aus einer Textur ein Tile-Set zu erstellen oder, falls es sich um einen Atlas handelt, einzelne Sprites zu extrahieren.

- **Unity**
  Die Unity Engine bietet einen Sprite Creator, einen Sprite Editor und einen Sprite Packer. Der Sprite Creator erlaubt simple Geometrische Formen zu erstellen, die als Platzhalter dienen können. Mit dem Sprite Editor können diverse Parameter der Sprites eingestellt werden. Weiters bietet der Sprite Editor ein Feature, um ein minimales Rechteck für ein Sprite zu finden, damit es optimal ausgeschnitten werden kann. Der Sprite Packer erlaubt mehrere Sprites optimal in eine Textur zu verpacken, sodass möglichst wenig leerer Raum zwischen Sprites liegt.

- **Fazit**
  Die Unity Engine bietet im Bereich 2D die besseren Features. Der Sprite Packer funktioniert erstaunlich gut und die separate 2D-Physik Engine, welche die Unity Engine verwendet, rücken sie an die Spitze.

3.5 User Interface

3.5.1 UI Designer

- **Unreal**
  Beim UMG UI Designer handelt es sich um ein Tool, mit dem UI Elemente wie

Ingame-HUDs oder Menüs erstellt werden können. Dem UMG UI Designer liegen, wie bei so gut wie allem in der Unreal Engine, Blueprints zugrunde. Mit Hilfe von Blueprints werden im UMG UI Designer Widgets erstellt. Widgets stellen Dinge wie Buttons und Progress Bars dar. Sie können untereinander verschachtelt werden und mit ihnen lassen sich sowohl HUDs als auch Ingame Menüs aufbauen.

- **Unity**
  Das Unity UI System ist vom Aufbau her HTML nicht unähnlich. Der Ausgangspunkt des User Interfaces ist ein Canvas, zu dem Elemente hinzugefügt werden. Die Elemente sind auf dem Canvas hierarchisch angeordnet. Ein Element hat einen Parent und kann beliebig viele Child Elemente besitzen. Der oberste Parent ist das Canvas. Basierend auf der Hierarchie werden die Elemente auch gezeichnet. Zuerst wird der Parent gezeichnet, danach das Child gefolgt vom nächsten Child. Das UI System bietet auch diverse Parameter, um Elemente beliebig zu platzieren, zu rotieren, zu fixieren und vieles andere. Das Unity UI System dient vor allem der Darstellung von Ingame Menüs.

- **Fazit**
  Der Blueprint Ansatz der Unreal Engine erlaubt es sehr einfach sehr dynamische UIs aufzubauen. Weiters können mit dem UMG UI Designer wirklich alle UI artigen Elemente umgesetzt werden. Von Ingame Menüs über Healthbars und andere Anzeigen sind alle Elemente über den UMG UI Designer erstellbar. In der Unity Engine ist es komplizierter, Elemente wie eine Health Bar darzustellen und es ist nicht so zentralisiert möglich wie in der Unreal Engine. Die Unreal Engine bietet das bessere UI System.

3.6 Animation

3.6.1 3D

- **Unreal**
  Das Animationssystem der Unreal Engine nennt sich Persona und ist ein System, das skelettbasierte Deformation mit morphbasierter Deformation verbindet und so komplexe Animationen ermöglicht. Persona ist aufgeteilt in vier Tools. Das Skeleton Tool ermöglicht das Manipulieren des Skeletal Mesh und der Bone- und Joint Hierarchie. Das Mesh Tool erlaubt das Manipulieren des polygonalen Meshes. Mit dem Animation Tool lassen sich Animationssequenzen anpassen. Das letzte Tool ist das Graph Tool mit dem sich über State Machines Animationsübergänge erzeugen lassen.

- **Unity**
  Bei der Unity Engine heißt das Animationssystem Mecanim. Vor allem menschenähnliche Figuren sind in Unity leicht zu animieren, da einige Tools existieren,

die einem die Arbeit erleichtern, wie zum Beispiel das Avatar Tool, das beim Mappen der Bones eines importierten Meshes hilft. Bei nicht humanoiden Charakteren sieht die Situation etwas trister aus. Auch nicht humanoide Charaktere lassen sich mit der Unity Engine animieren, jedoch ohne Hilfestellung durch die Unity Engine. Unity bietet auch eine übersichtliche State Machine an mit der sich einfach Animationsübergänge erstellen lassen. Weiters existiert auch ein Tool mit dem sich die maximalen Muskelbewegungen für die Knochen einstellen lassen.

- **Fazit**
  Die Unity Engine hat den offensichtlichen Nachteil, dass sie ihre Helper Tools nur auf humanoide Charaktere beschränkt. Die Unreal Engine bietet in diesem Bereich einen übersichtlichen Editor mit dem man sowohl humanoide als auch nicht humanoide Meshes animieren kann. Wie in vielen anderen Bereichen kristallisiert sich auch beim Animationssystem heraus, dass die Unity Engine schneller zugänglich ist da man nicht von Features erschlagen wird, jedoch die Unreal Engine weitaus mehr Möglichkeiten bietet, wenn man sich darauf einlässt.

3.7 Physics

3.7.1 PhysX

PhysX sorgt für die Kollisionserkennung und kümmert sich um alle physikalischen Interaktionen zwischen Objekten in der Welt.

- **Unreal**
  Die Unreal Engine verwendet Nvidia PhysX 3, und bietet vom Physics Engine Aspekt alles was Nvidia PhysX bietet.

- **Unity**
  Wie die Unreal Engine verwendet Unity auch Nvidia PhysX 3 und bietet die selben Vorteile und Einschränkungen. Allerdings bietet die Unity Engine speziell für 2D Projekte auch die Box2D Physics Engine, die Physikberechnungen im 2D Bereich stark vereinfacht und dadurch die Performance in 2D Projekten steigert.

- **Fazit**
  Die Unity Engine punktet im Physik Sektor durch eine zusätzliche performante Engine für 2D Physikberechnungen. Ansonsten sind in diesem Punkt beide Engines gleich auf.

3.8 Artificial Intelligence

- **Unreal**
  Behavior Trees dienen der Entscheidungsfindung für Agenten in einer virtuellen

Umgebung. Der Unreal Engine Behavior Tree besteht aus dem Blackboard und dem eigentlichen Behavior Tree. Das Blackboard speichert dabei Werte, die anschließend vom Beahavior Tree verwendet werden. Weiters dient das Blackboard als Speicher der AI und dient als entscheidungstreffende Komponente.

- **Unity**
  Die Unity Engine erlaubt das Erstellen von AI über die interne Scripting Engine.

- **Fazit**
  Unity bietet standardmäßig keinen Behaviour Tree an. Dadurch ist das Scripten von AI ein wesentlich langwierigerer Vorgang. Ein klarer Vorteil für die Unreal Engine.

3.9 Project

3.9.1 Profiling

- **Unreal**
  Der Unreal Engine Profiler ist ein Tool zum Monitoring der Spieleperformance. Wenn er aktiviert ist sammelt er Performance Daten und ermöglicht das Auffinden von Performance Flaschenhälsen oder anderen Problemen. Es werden genaue Prozess-Timings und diverse Graphen verwendet, um einen möglichst tiefen Einblick zu bieten.

- **Unity**
  Die Unity Engine bietet ein paar Funktionen an, aus denen man ableiten kann, dass etwas nicht stimmt. Aber für richtiges Profiling wird von Unity auf externe Tools verwiesen.

- **Fazit**
  Die UE4 bietet mit dem voll integrierten Profiling eine gute Lösung um Performance Probleme aufzuzeigen. Unity bietet nichts Vergleichbares.

3.10 Virtual Reality

Da die Entertainmentindustrie im Moment stark auf die Weiterentwicklung von Virtiual Reality setzt, und diese Entwicklung auch vor Game Engines keinen Halt macht, bieten beide Engines Unterstützung für HMD (Head Mounted Display) VR Geräte an.

- **Unreal**
  Die Unreal Engine bietet weitreichende Unterstützung für die im Moment erfolgreichsten HDMs Oculus Rift, Samsung Gear VR und Steam VR an. Diesbezüglich gibt es keine speziellen Tools aber ausführliche Best Practice Guides und debugging Hilfen.

- **Unity**
  Unity bietet derzeit Unterstützung für Oculus Rift und PlaystationVR an. Allerdings ist die Dokumentation im Bereich VR noch sehr dürftig.

- **Fazit**
  Keine der beiden Engines bietet ein eigenes Tool an, welches bei der VR-Entwicklung hilft, aber UE4 hat inzwischen schon eine recht ausführliche Dokumentation im Bereich VR und greift dem Entwickler dadurch mehr unter die Arme.

3.11 Cinematics

- **Unreal**
  Die Unreal Engine bietet das Tool Matinee an um dynamische Gameplay oder cinematische ingame Sequenzen zu erstellen. Wie bei einem Video Editor kann bei beliebigen Keyframes jedes Gelenk eines Actors angepasst werden um ingame Videosequenzen umzusetzen.

- **Unity**
  Unity bietet kein Tool zum Umsetzen von Cinematics.

- **Fazit**
  Die Unreal Engine bietet in diesem Bereich ein umfangreiches Tool an. Unity 5 hat in dem Fall Aufholbedarf.

3.12 Networking

- **Unreal**
  Da die Unreal Engine mit Multiplayer als eines der Kernkonzepte entwickelt wurde, haben die Entwickler darauf geachtet, dass es möglichst einfach ist ein Spiel multiplayerfähig zu machen. Die Unreal Engine verwendet das Server/Client Modell. Der Server fungiert als autoritative Einheit, welche die relevanten Änderungen der Aktoren an die Clients weiterleitet.

- **Unity**
  Auch die Unity Engine wurde mit Multiplayer als einem der Kernkonzepte entwickelt. Wie auch die Unreal Engine bietet Unity ein Server/Client Modell, bei dem alle relevanten Änderungen an die Clients weitergeleitet werden.

- **Fazit**
  Beide Engines bieten im Bereich Networking dieselben Möglichkeiten. Wobei die Unreal Engine über Blueprints ein schnelleres und übersichtlicheres Prototyping erlaubt. Für komplexere Netzwerk-Angelegenheiten muss bei der Unity Engine schnell auf die Netzwerk-"High Level Scripting API" zurückgegriffen werden.

CHAPTER 4

Ein einfaches Projekt erstellen

In diesem Abschnitt wollen wir ein einfaches Rolling Ball Game in beiden Engines erstellen, um die Vorgehensweise sowie den Aufwand zu demonstrieren und zu vergleichen. Das Spiel soll eine Arena mit Hindernissen enthalten, in der man einen Ball aus der 3rd-Person Perspektive steuert. Benötigte Komponenten:

- Level
- Spieler Charakter
- Hindernisse und andere Objekte
- Physik und Kollision
- Collectible
- Menü und Pause
- HUD

Es ist anzumerken, dass bei der Unity Engine die y-Achse der z-Achse in der Unreal Engine entspricht und umgekehrt.

4.1 Neues Projekt starten

Im ersten Schritt wird ein neues Projekt in der Engine gestartet.

4.1.1 Unity 5

In Unity 5 kann man ein neues Projekt im Home-Screen über den "NEW"-Button oder im Editor selbst unter "File/New Project" erstellen. Anschließend muss sowohl Name als auch Speicherort angeben werden und ob es sich um ein 2D oder 3D Game handelt (Abbildung 4.1). Zusätzlich hat man die Option diverse Asset packages auszuwählen, welche im Projekt zur Verfügung stehen. Nach Klicken auf "Create Project" wir das neu erstellte Projekt im Editor geladen und kann bearbeitet werden (Abbildung 4.2).

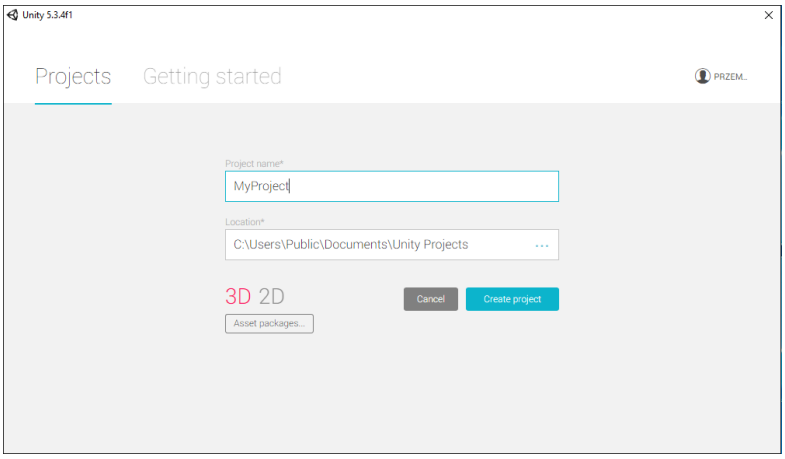

Abbildung 4.1: (Unity) Neues Projekt

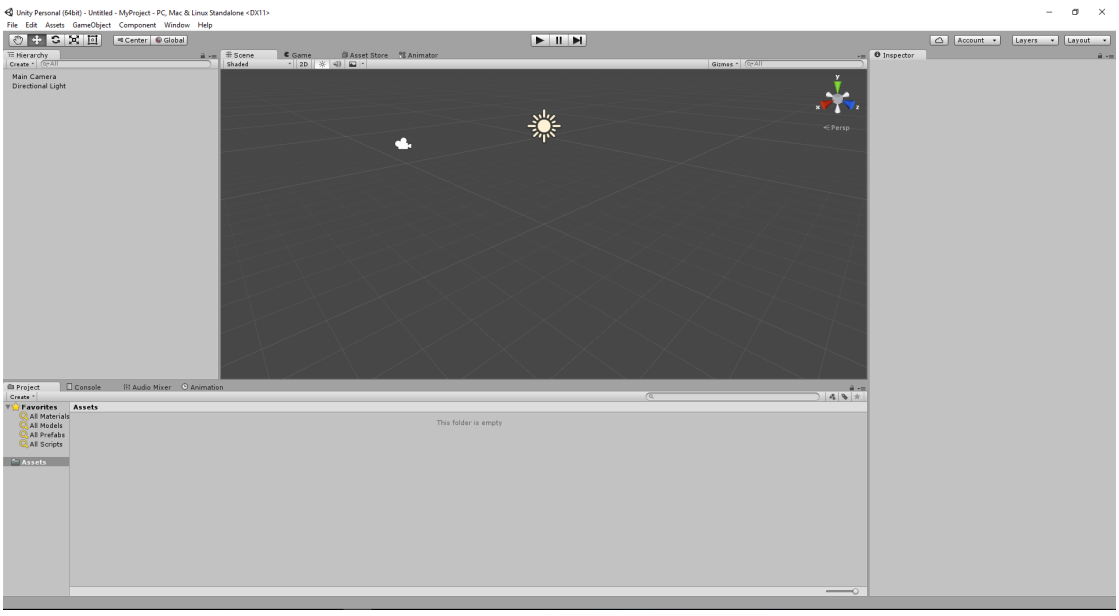

Abbildung 4.2: (Unity) Editor

4.1.2 Unreal Engine 4

Um in der UE4 ein neues Projekt zu erstellen muss zuerst der Unreal Launcher gestartet werden. Im Launcher findet man neben dem Button zum Starten des UE4 Editors auch Tutorials sowie den Marketplace. Man startet den UE4 Editor und klickt auf "New Project" (Abbildung 4.3). Danach hat man die Auswahl zwischen einem Blueprint- und einem C++ Projekt. Bei der Blueprint Variante steht das Visual Scripting im Vordergrund, während man bei der anderen Variante eher mit C++ und Visual Studio arbeitet. Für beide Varianten stehen bereits einige Templates zur Verfügung, die einen kleinen Einblick in Spieleansätze geben. Weiters hat man die Möglichkeit, Projekt-Voreinstellungen vorzunehmen, die sich allerdings jederzeit im Editor ändern lassen. Zu diesen Voreinstellungen zählt das Deployment-Ziel des Projektes, d.h. ob für Desktop/Konsole entwickelt wird oder für Mobile Plattformen. Weiters lässt sich die angestrebte grafische Qualität sowie die Option, Starter Content zu verwenden, einstellen. Der Starter Content beinhaltet ua. einige vorgefertigte Assets, welche bei einem neuen Projekt hilfreich sein können. In dieser Demonstration wird ein Blank Blueprint Projekt ohne Starter Content erstellt. Nach dem Erstellen des Projektes befindet man sich im Unreal Editor (Abbildung 4.4). Dort besteht der erste Schritt darin, die Szene bzw. Map zu speichern. Dazu geht man über das File Menü auf Save und speichert die derzeitige Map unter einem beliebigen Namen ab.

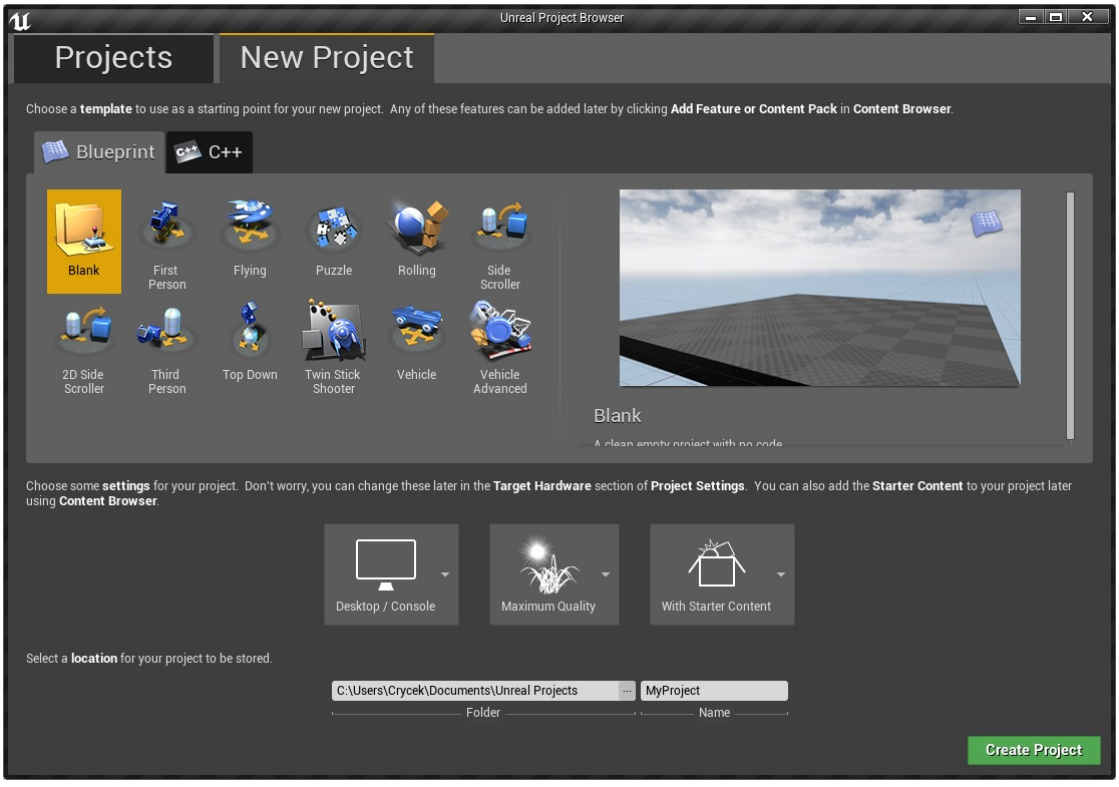

Abbildung 4.3: (Unreal) Neues Projekt

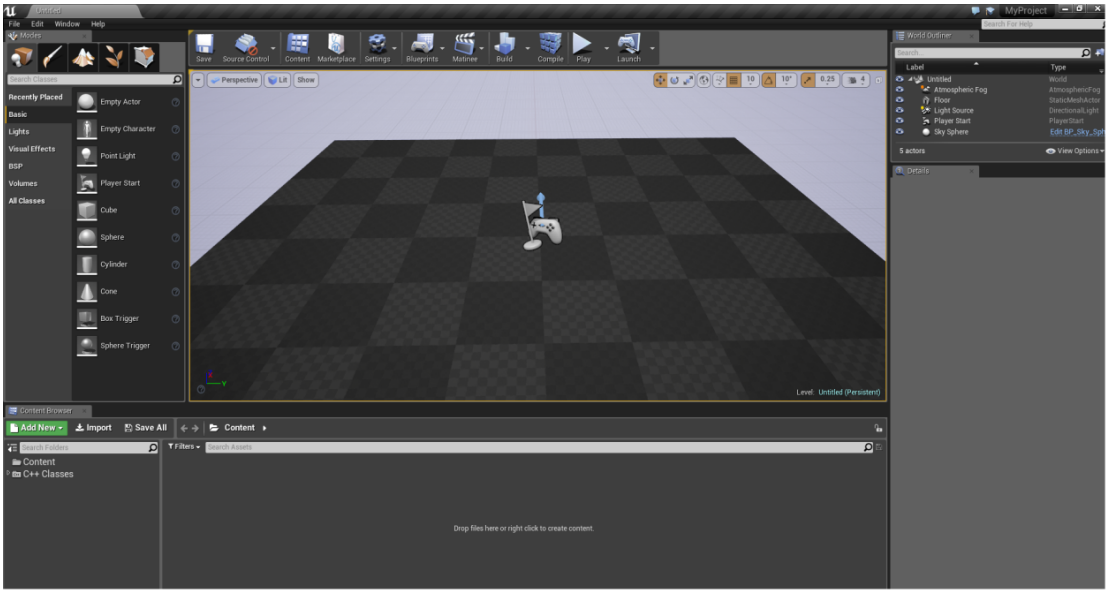

Abbildung 4.4: (Unreal) Editor

4.1.3 Fazit

Das Erstellen eines neuen Projektes erweist sich in beiden Engines als sehr unkompliziert. Auf beiden Seiten werden uns vorgefertigte Assets zur Verfügung gestellt.

4.2 Level Design

Für unser Beispiel werden wir eine einfache rechteckige Arena erstellen. Dazu wird eine Oberfläche benötigt, auf der sich der Spieler bewegen kann, sowie eine Umrandung, welche ihn daran hindern soll, die Arena zu verlassen. Um die Beleuchtung des Levels müssen wir uns nicht kümmern, da die Engines automatisch ein Directional Light in die Szene einbauen.

4.2.1 Unity 5

In Unity 5 können wir über "GameObject/3D Object" in der Menüleiste oben eine Plane erstellen. Alternativ kann man auch im Hierarchy-Fenster mit Rechtsklick in dasselbe Create-Menü gelangen. Als nächstes wählen wir die Plane aus und vergrößern sie, indem wir im Inspector-Fenster den Scale-Faktor erhöhen (Abbildung 4.5). Für die Umrandung werden wir vier Wände brauchen, welche am Rand unserer Plane angebracht sind, sodass unsere Kugel später nicht aus dem Level fällt. Dafür erstellen wir vier Cubes über "GameObject/3D Object" oder das Create-Menü im "Hierarchy"-Fenster. Diese wählen wir nun einzeln aus und verschieben sie jeweils an die vier Ränder unserer Plane indem wir ihre x,z Position Werte entsprechend setzen. Als nächstes passen wir mit dem

Scale-Faktor ihre Größe an, sodass sich die Cubes über die jeweilige Seite ausdehnen und eine gewisse Höhe haben. Zum Schluss ändern wir noch die y-Position, damit die Wände genau auf unsere Plane liegen (Abbildung 4.6).

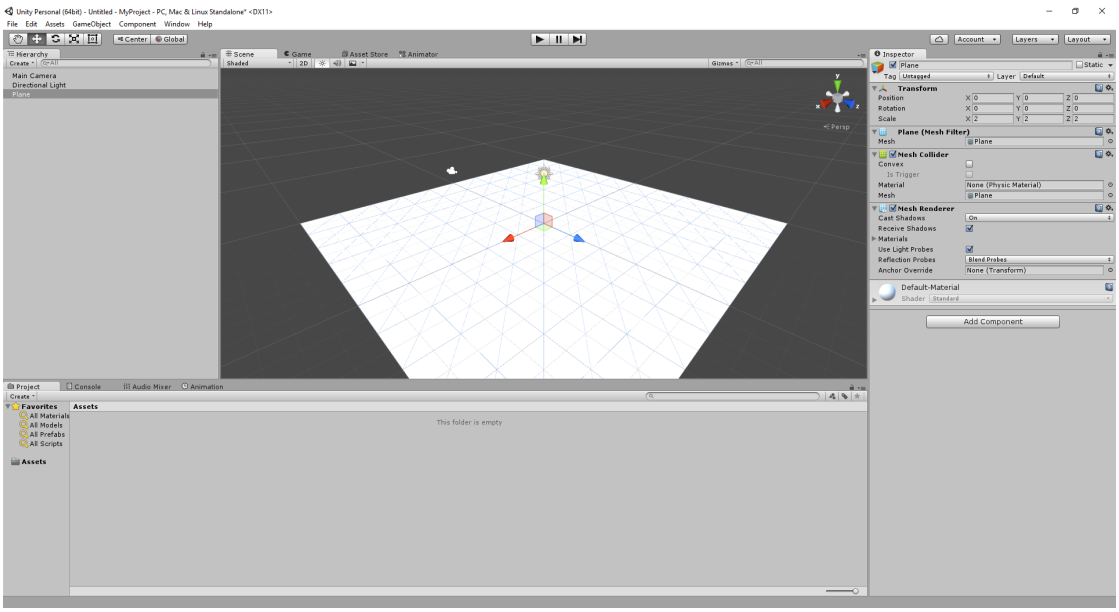

Abbildung 4.5: (Unity) Bodenfläche

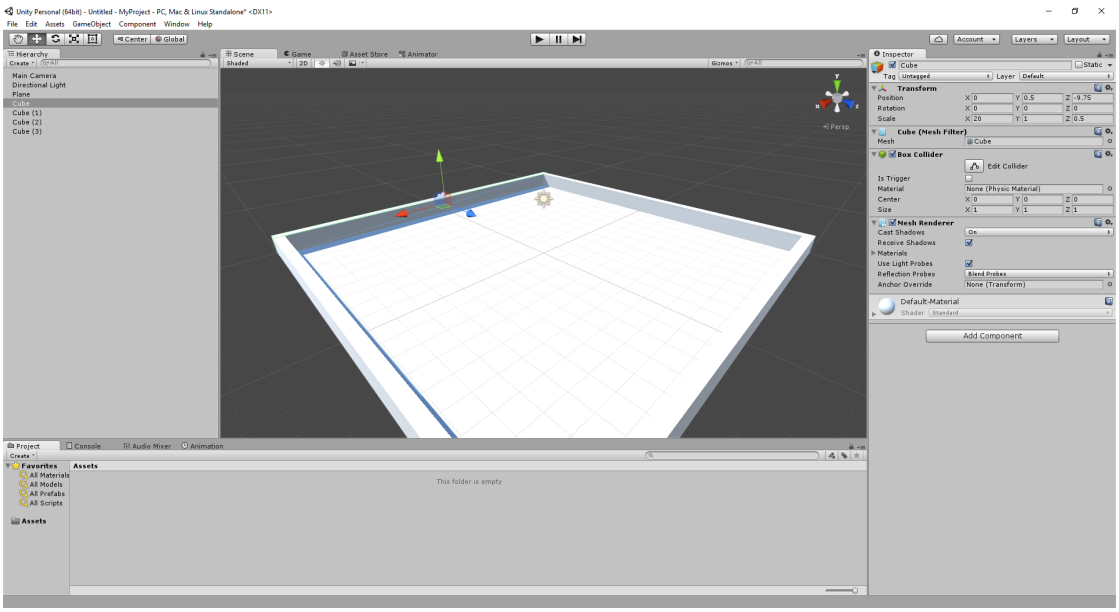

Abbildung 4.6: (Unity) Boden mit Wänden

4.2.2 Unreal Engine 4

Die benötigte Plane wird bei einem leeren Projekt in der Unreal Engine automatisch erstellt. Damit man später nicht von der Plane fällt werden wir noch vier Wände um die Plane herum aufstellen. Wir ziehen dazu aus dem Modes Fenster einen Würfel in die Szene. Wir wählen den Würfel aus und stellen im Details Fenster auf der rechten Seite den x-Scalefaktor so ein, dass die Wand dieselbe Länge hat wie unsere Plane, und den y-Scalefaktor so ein, dass dabei eine Wand entsteht (Abbildung 4.7). Nun verschieben wir die Wand an den Rand der Plane. Um die Gegenüberliegende Wand zu erstellen muss nur "Alt" gedrückt gehalten werden während man mit dem Translate Object Tool das Objekt in die gewünschte Richtung zieht. Dadurch wird eine Kopie des Objekts erstellt. Nun zieht man die Kopie an die gewünschte Position. Um die anderen beiden Wände zu erstellen gehen wir analog zur letzten Wand vor, nur dass die Wand nach dem Kopieren auch noch rotiert wird. Um das zu erreichen stellen wir den Yaw Value der gewünschten Wand auf 90 und ziehen die Wand anschließend in die richtige Position. Dasselbe Ergebnis kann auch durch Vertauschen der x- und y-Skalierung erreicht werden. Die letzte Wand wird einfach kopiert und in die richtige Position gebracht (Abbildung 4.8).

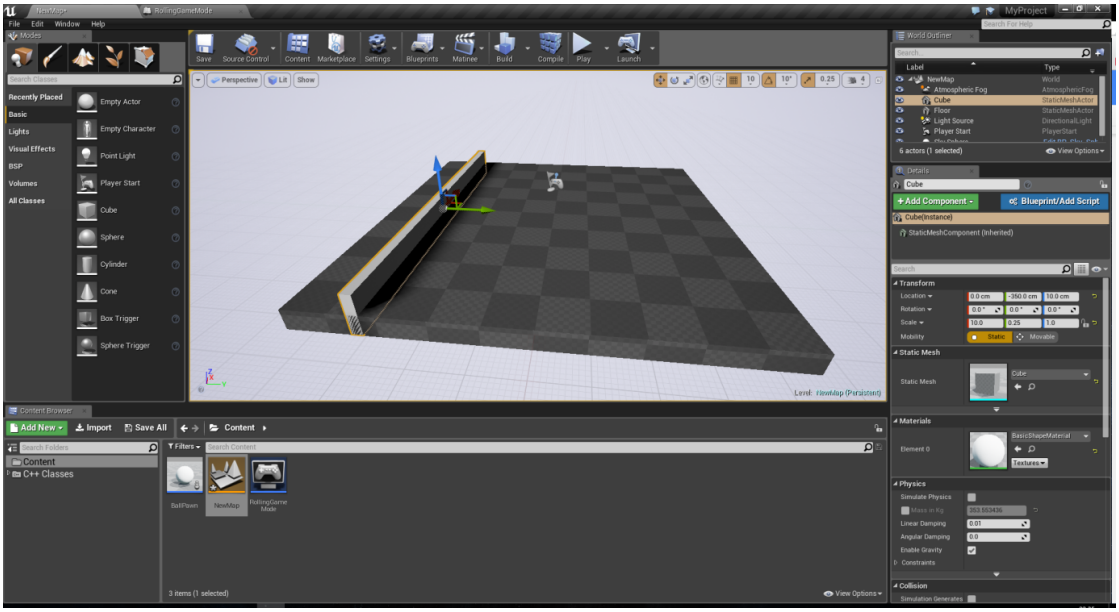

Abbildung 4.7: (Unreal) Bodenfläche

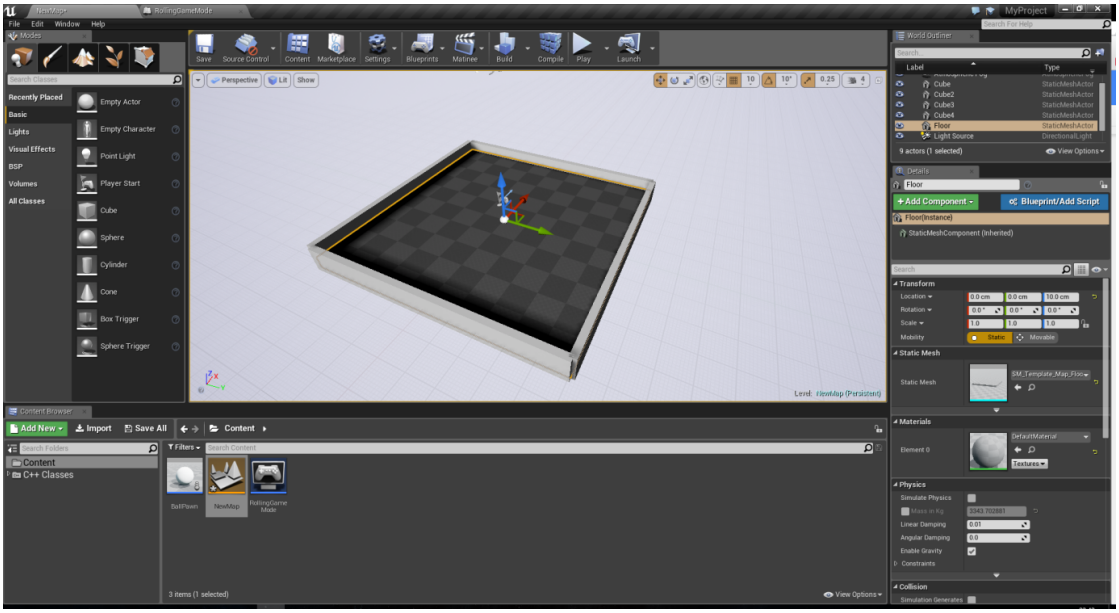

Abbildung 4.8: (Unreal) Boden mit Wänden

4.2.3 Fazit

In beiden Engines lässt sich das Level Design der Arena recht einfach umsetzen. Die UE 4 greift uns zusätzlich mit der Schnell-Funktion, ausgewählte Objekte mit gedrückter Alt-Taste zu kopieren und zu verschieben, unter die Arme.

4.3 Spieler Charakter

In unserem Beispiel soll der Spieler eine Kugel steuern und im späteren Verlauf mit anderen Objekten kollidieren können. Außerdem soll die Spieleransicht schräg von oben auf die Kugel herabsehen und an diese gebunden sein.

4.3.1 Unity 5

In Unity 5 erstellen wir zunächst ein Sphere-Objekt und verschieben es auf der y-Achse, sodass es auf unserer Plane liegt. Nun wählen wir die Kugel aus und fügen ihr einen Rigidbody als Komponente hinzu. Dieser ist für physikalisches Verhalten zuständig (Abbildung 4.9). Wir wollen, dass der Spieler unsere Kugel steuern kann. Dafür müssen wir ein Script schreiben, welches dieses Verhalten modelliert. Über "Add Component/New Script" erstellen wir ein neues C#-Script, welches anschließend über das Project-Fenster mit Doppelklick in einem externen Editor wie VisualStudio oder MonoDevelop-Unity (falls man diesen bei der Installation eingestellt hat) geöffnet werden kann. Hier fügen wir unseren Code für die Kugel-Bewegung ein. 4.1

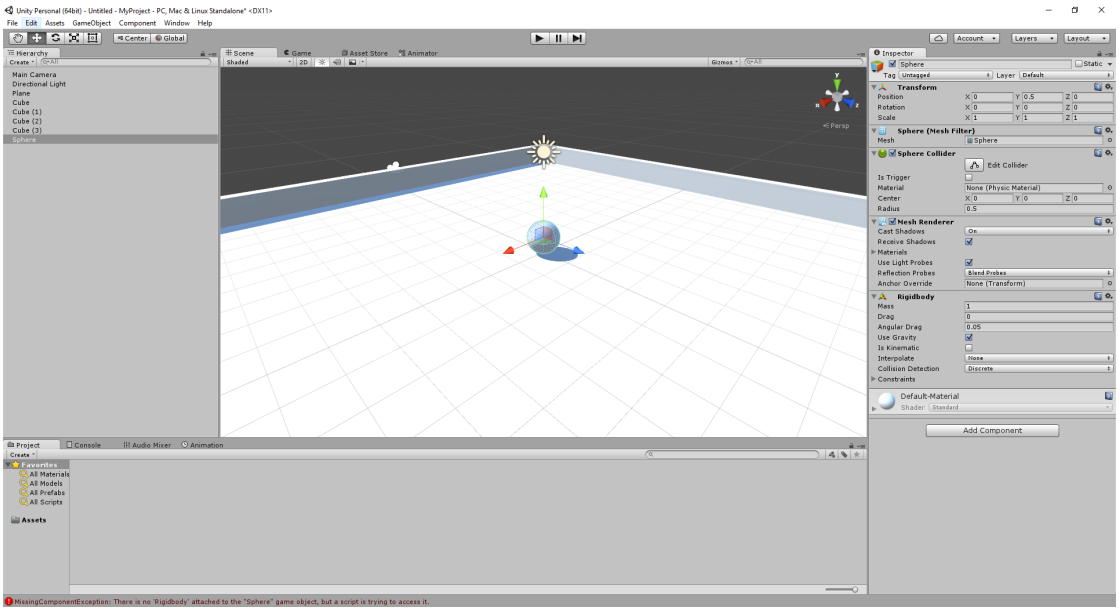

Abbildung 4.9: (Unity) Kugel hinzugefügt

```
using UnityEngine;
using System.Collections;

public class playerController : MonoBehaviour {

public float speed;
private Rigidbody rb;

// Use this for initialization
void Start () {
rb = GetComponent<Rigidbody>();
}

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

void FixedUpdate ()
{
float moveHorizontal = Input.GetAxis("Horizontal");
float moveVertical = Input.GetAxis("Vertical");

Vector3 movement = new Vector3(moveHorizontal, 0.0f, moveVertical);

rb.AddForce(movement * speed);
}
}
```

Listing 4.1: Kugel-Bewegung

Zuerst deklarieren wir oben zwei Variablen:
`public float speed:` diese Variable bestimmt die Geschwindigkeit mit der unsere Kugel rollen wird. Sie ist auf public gestellt, damit wir sie zu einem späteren Zeitpunkt direkt im Editor verändern können.
`private Rigidbody rb:` eine Referenz auf unsere Rigidbody Komponente welche wir vorhin im Editor erstellt haben.
In der `Start()`-Methode wird diese Variable mittels `GetComponent<Rigidbody>()` mit der entsprechenden Komponente belegt. Dies geschieht einmalig beim erstellen unserer Kugel. Nun folgt der Movement-Teil, welchen wir in der `FixedUpdate()`-Methode definieren. Diese wird im Gegensatz zur normalen `Update()`-Methode in regelmäßigen Intervallen aufgerufen und sollte für physikalische Veränderungen oder Einwirkungen auf Objekte verwendet werden. Hier deklarieren wir zwei floats, welchen wir die Player Input-Werte der jeweiligen Achsen zuweisen. Mit Hilfe dieser beiden Werte erstellen wir nun einen Movement-Vektor, den wir anschließen unserem Rigidbody als Impuls hinzufügen. Nun können wir das Script speichern und zum Unity Editor wechseln. Bei der Script Komponente unserer Kugel erscheint jetzt der Wert "Speed", welchen wir beliebig ändern können. Die Bewegung der Kugel mit "WASD" ist fertig und kann über "Play" getestet werden. Allerdings bewegt sich unsere Kamera nicht mit dem Ball mit. Dafür müssen wir ein weiteres Script erstellen, diesmal für das MainCamera-Objekt. 4.2

```
using UnityEngine;
using System.Collections;

public class cameraController : MonoBehaviour {

public GameObject player;
private Vector3 offset;

// Use this for initialization
void Start () {
offset = transform.position - player.transform.position;
}

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

void LateUpdate () {
transform.position = player.transform.position + offset;
}
}
```

Listing 4.2: Kamera-Bewegung

Hierzu brauchen wir eine Referenz auf die Kugel und einen Offset-Vektor. In der `Start()`-Methode bestimmen wir unseren Offset-Vektor, welchen wir in `Update()` zusammen mit der Kugelposition verwenden, um die richtige Position der Kamera zu bestimmen. Nachdem wir mit unserem Script fertig sind, wechseln wir in den Unity Editor und wählen unser Kamera-Objekt aus. Bei der Script Komponente ist nun ein neues Feld aufgetaucht in das wir unser Sphere-Objekt ziehen müssen. Bei dem Feld

handelt es sich um die `public GameObject` Variable, welche wir im Script deklariert haben. Nun ist die Kamera an die Kugel gebunden und bewegt sich mit dieser mit, jedoch entspricht die Positionierung noch nicht ganz einer 3rd-Person Perspektive. Wir wählen unsere Kamera aus und verändern ihre Position und Rotation, sodass sie von schräg oben auf die Kugel herabsieht (Abbildung 4.10). Da die Beleuchtung der Szene etwas ungünstig für unser Beispiel ist, ändern wir die Rotation des Directional Lights, welches am Anfang automatisch von der Engine erstellt wurde (Abbildung 4.11).

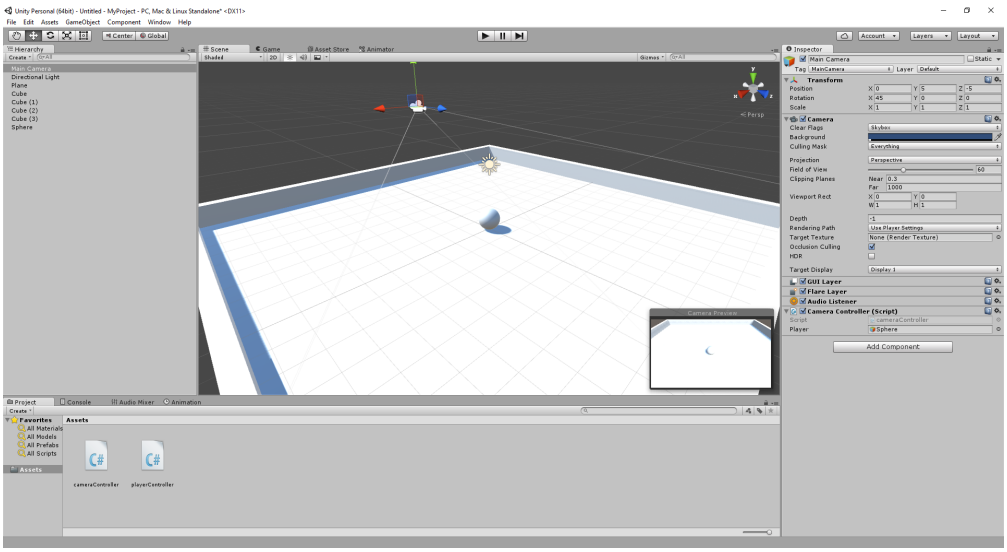

Abbildung 4.10: (Unity) Kamera Position im Editor

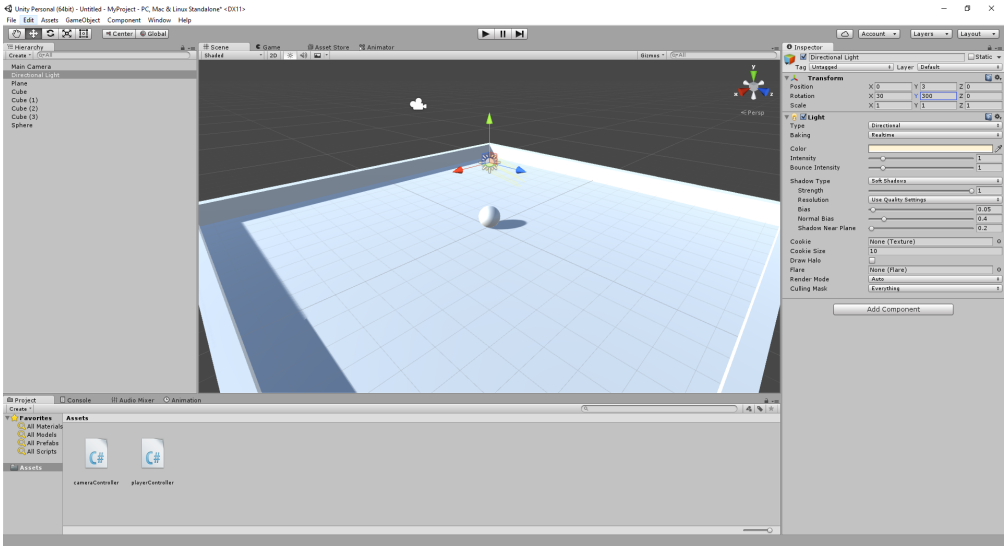

Abbildung 4.11: (Unity) Licht angepasst

4.3.2 Unreal Engine 4

Der erste Schritt besteht darin, die Input Mappings festzulegen. Die Input Mappings findet man im Settings Menü, das sich über dem Viewport befindet. Dort wählt man die Project Settings aus und findet unter Engine den Eintrag Input. Hier lassen sich alle Input Mappings für das Projekt festlegen. Um die Kugel bewegen zu können brauchen wir zwei Methoden: `"MoveRight"`, für die Links und Rechtsbewegung sowie `"MoveForward"`, für Vorwärts und Rückwärts. Die Mappings werden über das Plus Symbol bei den Axis Mappings hinzugefügt. Pro Methode brauchen wir zwei Buttons, die bei den Methoden über den Plus Button hinzugefügt werden. Für `"MoveRight"`, nehmen wir "A" und "D", für `"MoveForward"` "W" und "S". Als letztes muss noch der Scale-Faktor der einzelnen Tastenbelegungen angepasst werden. Damit die Rückwärtsbewegung über die "S" Taste funktioniert muss die Scale dafür auf -1 gesetzt werden, dasselbe muss auch für die Linksbewegung mit "A" eingestellt werden (Abbildung 4.12).

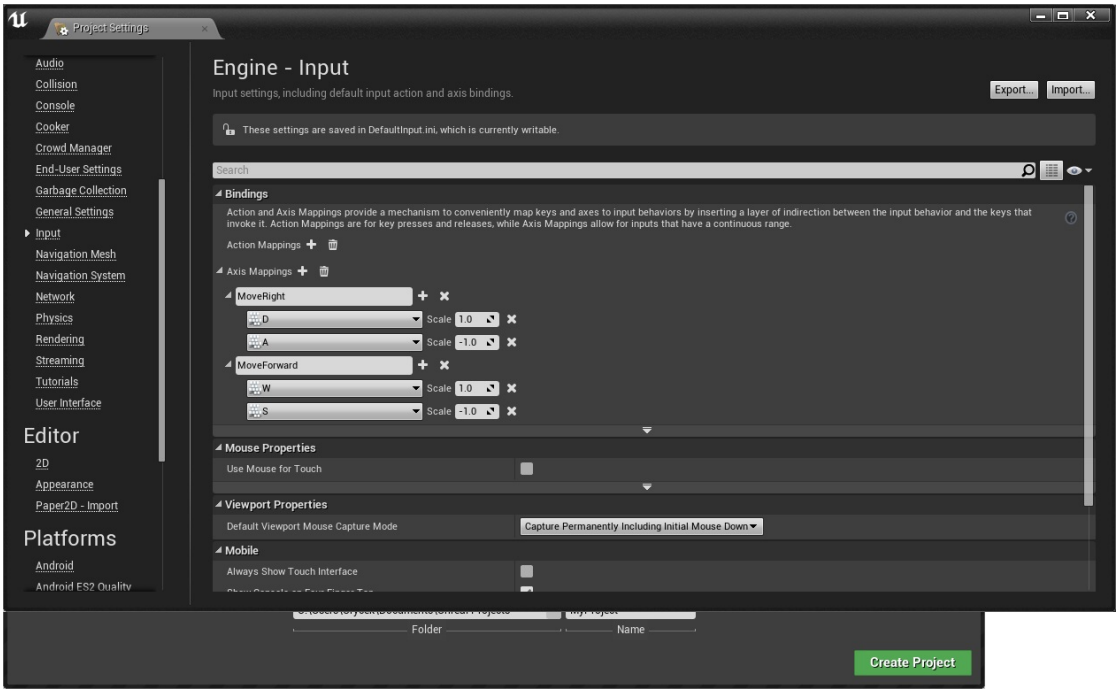

Abbildung 4.12: (Unreal) Eingabe-Einstellugen

Als nächstes müssen wir einen Blueprint für den Game Mode erstellen, der uns erlaubt, Spielregeln zu definieren. Zu diesem Zweck macht man einen Rechtsklick im Content Fenster unter dem Viewport und wählt unter "Create Basic Asset" die Blueprint Class aus. Danach öffnet sich ein PopUp mit der Auswahl der Parent Class (Abbildung 4.13). Hier wählt man "Game Mode" aus woraufhin der Blueprint erstellt wird und man ihn benennen kann. Mit einem Doppelklick auf den Game Mode kann man sich dessen Einstellungen ansehen und verändern. In unserem Fall werden wir hier keine Änderungen vornehmen und den default erstellten Game Mode verwenden. Als nächstes muss man

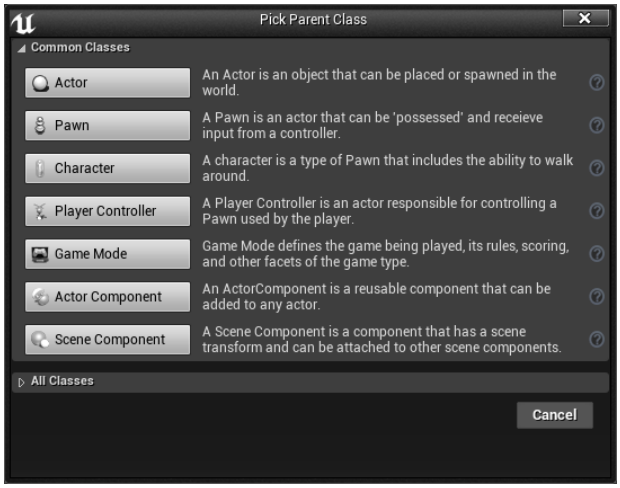

Abbildung 4.13: (Unreal) Neuen Blueprint erstellen

einen weiteren Blueprint erstellen, welcher den Spieler-Input verwendet und ihn in die Bewegung der Kugel umwandelt. Diesmal nimmt man als Parent Class für den Blueprint nicht den Game Mode, sondern Pawn. Ein Pawn in der UE4 ist ein Actor, der vom Spieler übernommen werden kann bzw. ein AI Spieler. Mit einem Doppelklick auf den neu erstellten Pawn kommt man in den Blueprint Editor. Links im Components Fenster sieht man den Eintrag "DefaultSceneRoot". Hier erstellt man den Actor für die Demonstration. In unserem Fall ist es eine Sphere. Dafür wählt man über "Add Component" unter Common eine Sphere aus, klickt danach auf den erstellten Eintrag und zieht ihn auf den "DefaultSceneRoot", da wir wollen, dass die Sphere der Root für dieses Blueprint ist.
Um den Actor zu vervollständigen brauchen wir noch einen Spring Arm und eine Kamera. Auch diese beiden Objekte fügen wir über "Add Component" hinzu; zuerst den Spring Arm, den wir auf die Sphere ziehen und danach die Kamera, die wir auf den Spring Arm ziehen (Abbildung 4.14).

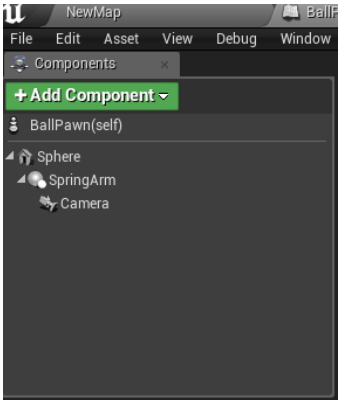

Abbildung 4.14: (Unreal) Blueprint Komponenten

Im Blueprint Editor wechseln wir vom Viewport in den Event Graph, die grafische Programmieroberfläche der UE4. Hier findet man drei vordefinierte Nodes, die man für diese Demonstration nicht braucht. Daher zieht man einen Rahmen über die drei Nodes und entfernt sie. Als erstes wird die links/rechts Bewegung erstellt. Dazu macht man einen Rechtsklick in den Event Graph und tippt die bei den Inputmappings erstellte Methode "`MoveRight`" ein. Die Methode wird gefunden und mit Enter oder einem Klick auf diese als Node in den Event Graph eingefügt. Um den Ball zu bewegen brauchen wir eine Drehbewegung für das Objekt. Die UE4 bietet zu diesem Zweck die Methode "`Add Torque`" an. Um unseren Input mit der Methode zu verbinden, zieht man vom Pfeilsymbol der "`MoveRight`"-Methode eine Verbindung heraus. Nach dem Loslassen der Maustaste hat man die Möglichkeit, einen weiteren Node zu erstellen. Über die Suche findet man "`Add Torque`", das nach dem Hinzufügen auch gleich unsere Sphere als Target eingetragen hat. "`Add Torque`" hat eine Property "Torque", welche einen Vektor verlangt. Diesen Vektor bauen wir uns aus dem Axis Value des Inputs. Allerdings ist ein Wert von 1 bzw. -1 viel zu gering um die Kugel damit sinnvoll bewegen zu können, daher müssen wir den Axis Value vor dem umwandeln in einen Vektor noch mit einem Faktor multiplizieren. Dazu erstellen wir eine neue Variable. Über das Plus links bei Variables kann man eine neue Variable erstellen. Nach einem Klick auf den Eintrag sehen wir rechts das Details Fenster der Variable, in dem wir den Typ auf float umstellen und das Häkchen bei "Editable" setzen können. Als Default Value trägt man 50000000 ein. Nach demselben Schema wie oben erstellt man die übrigen Nodes (Abbildung 4.15). Die Vorwärts/Rückwärtsbewegung wird analog dazu erstellt nur dass man die "`MoveForward`" Axis als Input nimmt (Abbildung 4.16).

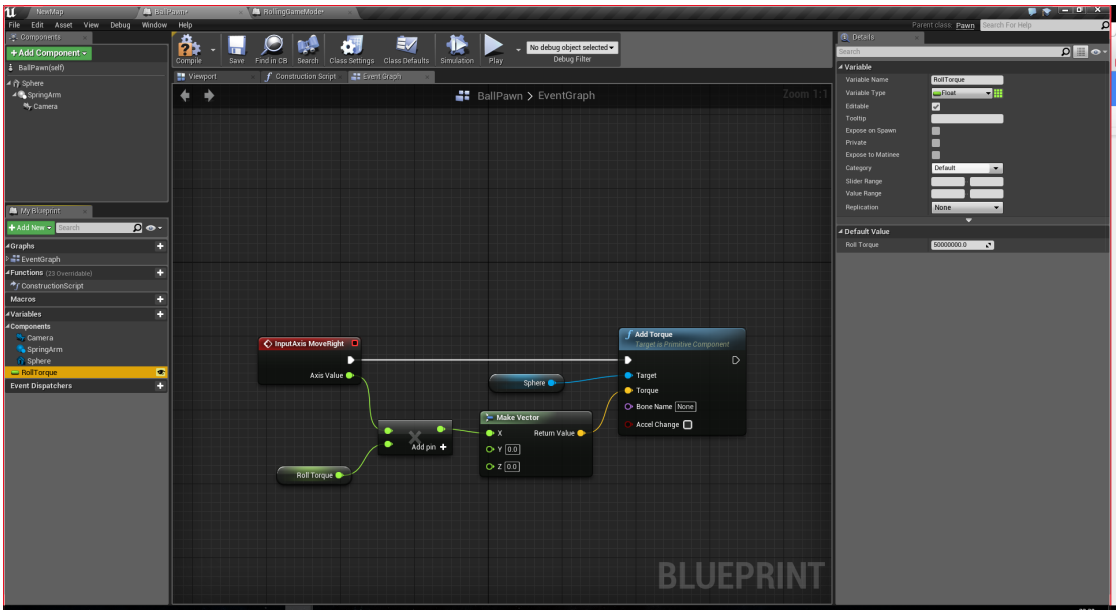

Abbildung 4.15: (Unreal) Kugelsteuerung 1

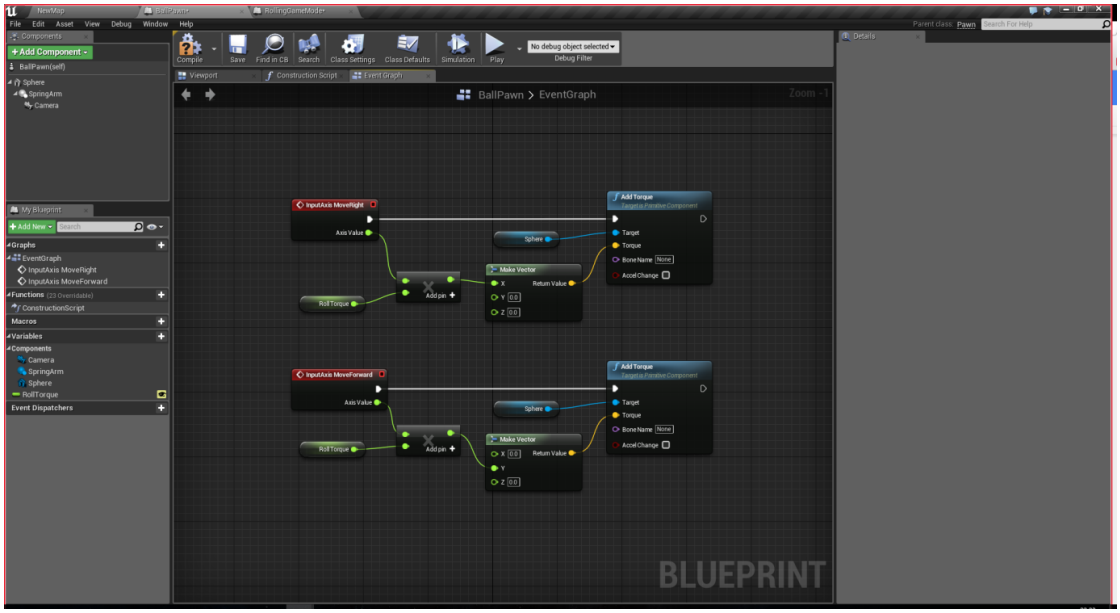

Abbildung 4.16: (Unreal) Kugelsteuerung 2

Als Nächstes stellen wir die Kamera auf einen etwas sinnvolleren Winkel ein. Dazu wechseln wir in den ViewPort Tab und klicken links auf den Spring Arm. Im Details Fenster auf der rechten Seite stellen wir die Rotation der Y-Achse auf -45°. Damit sich die Kamera nicht mit der Kugel dreht, muss man bei den Kamera Settings Inherit Pitch, Yaw und Roll abdrehen (Abbildung 4.17).
Als nächsten Schritt aktivieren wir die Physik bei der Sphere sodass eine Drehbewegung auch wirklich zu einer Bewegung der Kugel umgesetzt wird. Dazu klicken wir links auf den Sphere Eintrag und aktivieren bei Physics "Simulate Physics". Zusätzlich wird die Masse in Kg auf 5Kg gesetzt und Linear sowie "Angular Dampening" auf 1 gesetzt (Abbildung 4.18). Anschließend wechselt man auf die Map, zieht den gerade erstellten Blueprint in die Szene und wählt ihn aus. Im Details Panel unter Pawn setzt man "Auto Possess Player" auf "Player 0". Nach einem Klick auf den Play Button lässt sich die Kugel nun auf der Plane bewegen.

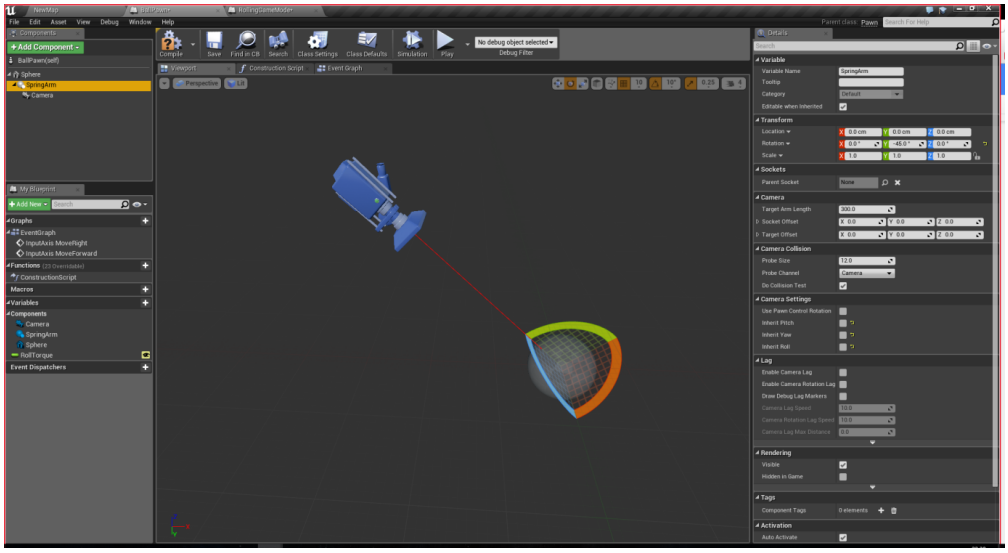

Abbildung 4.17: (Unreal) Kamera Einstellungen 1

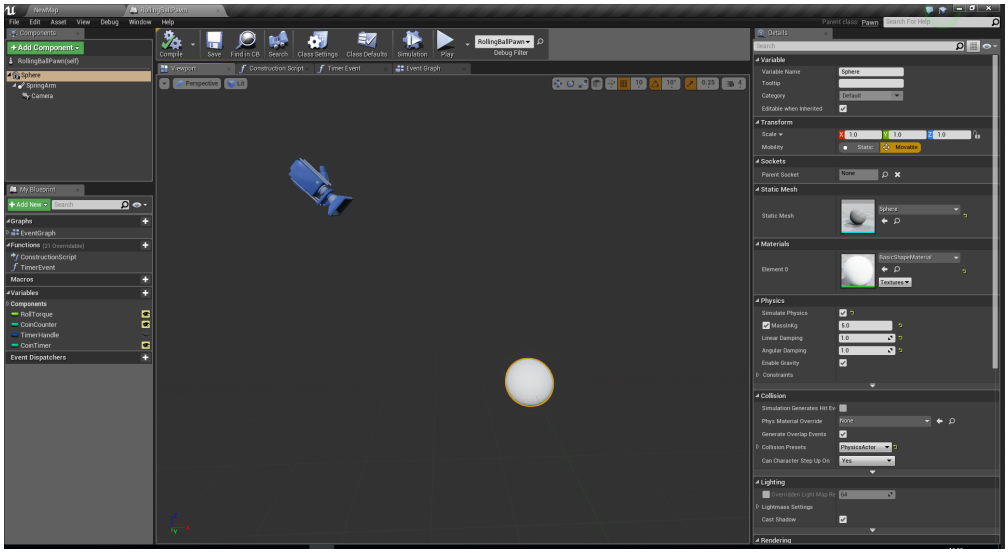

Abbildung 4.18: (Unreal) Kamera Einstellungen 2

4.3.3 Fazit

Da wir uns bei UE4 für die Blueprint-Variante des Projektes entschieden haben, wirkt dieser Schritt auf den ersten Blick etwas komplizierter als das Skripten in Unity 5, obwohl der Aufwand auf beiden Seiten in etwa gleich ausfällt. Im Wesentlichen wurden in beiden Projekten unterschiedliche Herangehensweisen angewandt.

4.4 Hindernisse

In unserem Game wollen wir abgesehen von den Wänden ein paar Hindernisse einbauen, mit denen unsere Kugel kollidieren kann.

4.4.1 Unity 5

In Unity ist dieser Vorgang sehr einfach, da man im Prinzip nur wie gewohnt 3D-Objekte erstellt und ihnen eine Rigidbody-Komponente hinzufügt. Auf diese Weise erstellen wir drei Würfel und vier Zylinder und verteilen sie auf der Grundoberfläche. Werte wie "mass", "drag" und "angular drag" müssen eventuell noch angepasst werden, je nachdem wie sehr sich Kollision auf Hindernisse oder die Spieler-Kugel auswirken soll (Abbildung 4.19). Damit sich nicht alle Hindernisse gleich verhalten, werden wir die "friction" und "bounciness" unserer Würfel etwas modifizieren. Dazu benötigen wir ein Physic Material, welches über "Assets/Create/Physic Material" erstellt werden kann (Abbildung 4.20).

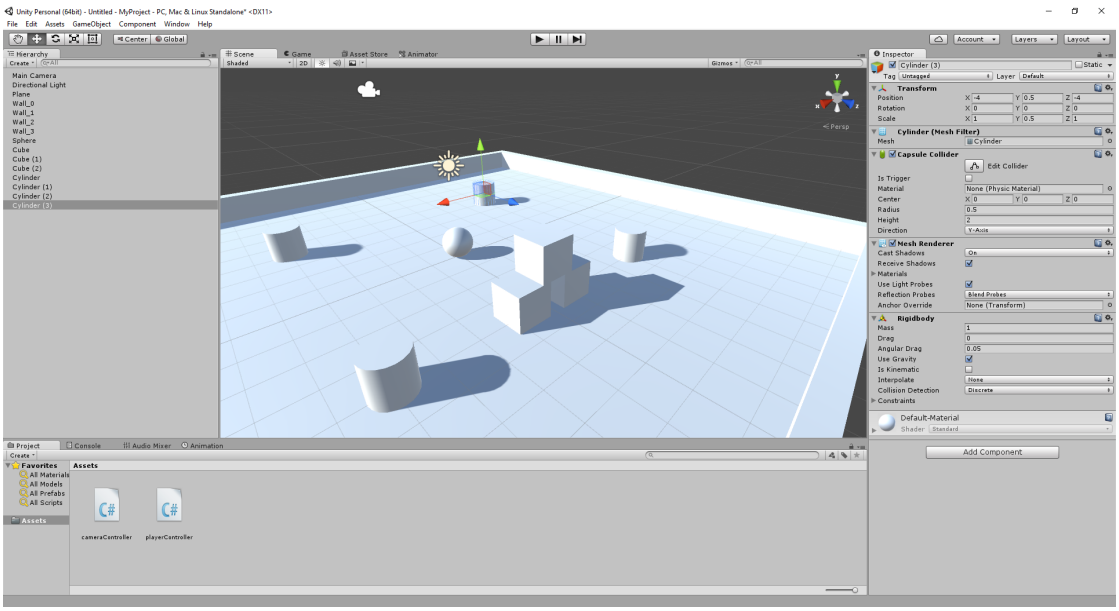

Abbildung 4.19: (Unity) Hindernisse hinzugefügt

Sobald die Werte entsprechend gesetzt wurden, muss die Datei in ein Material-Feld gezogen werden. In unserem Fall befindet sich dieses in der Collider-Komponente unserer Cube-Objekte (Abbildung 4.21). Die Zylinder sollen in unserem Beispiel als feste Säulen dienen und daher physikalisch nicht beeinflusst werden. Das lässt sich ganz einfach bewerkstelligen, indem man entweder in der Rigidbody-Komponente den Punkt "Is Kinematic" einschaltet oder die gesamte Komponente entfernt (Abbildung 4.22).

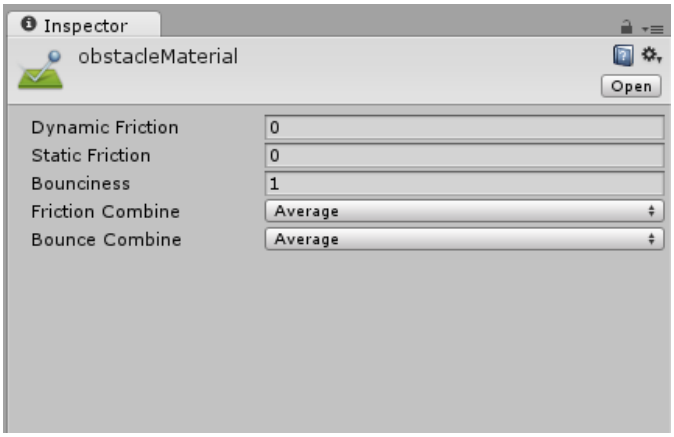

Abbildung 4.20: (Unity) Neues Physics Material

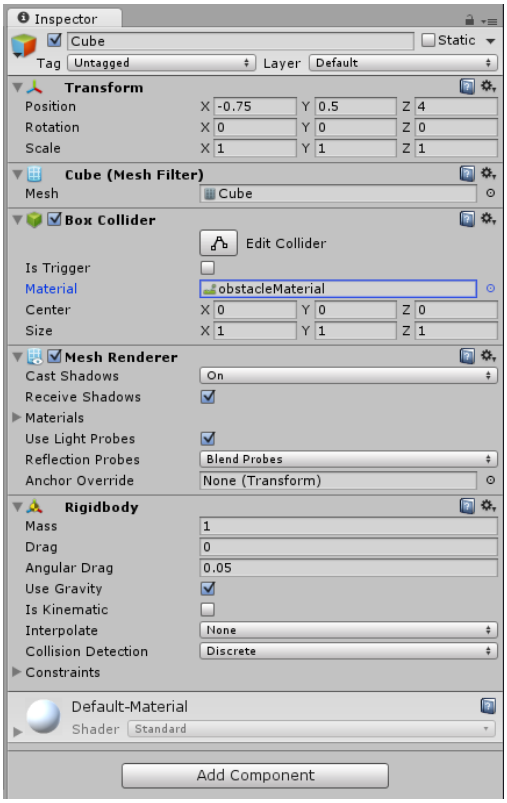

Abbildung 4.21: (Unity) Hindernis Eigenschaften 1

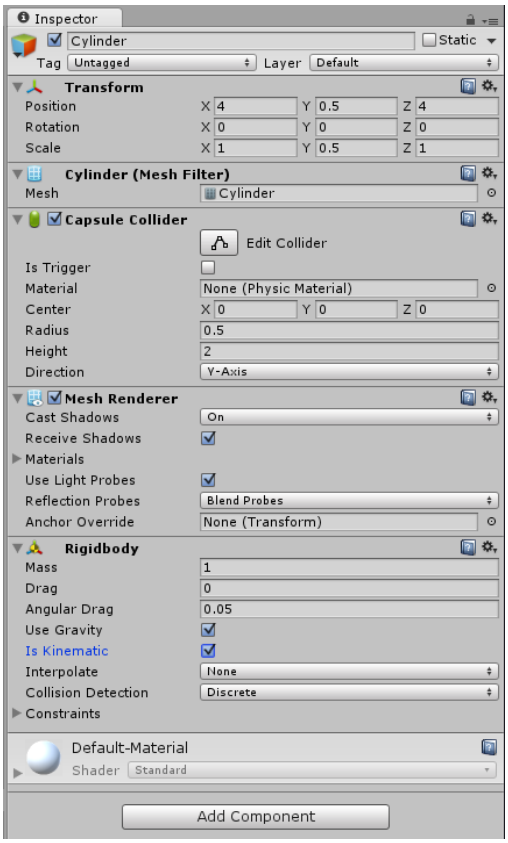

Abbildung 4.22: (Unity) Hindernis Eigenschaften 2

4.4.2 Unreal Engine 4

Wie auch in der Unity Engine ist hier dieser Vorgang sehr einfach. Man zieht zu diesem Zweck einfach ein Cube-Element aus dem Modes Panel in die Map. Wie auch beim Unity Beispiel werden wir eine Pyramide aus drei Würfeln bauen. Weiters ziehen wir vier Säulen auf die Map und ordnen sie in einem Quadrat auf der Map an (Abbildung 4.23).

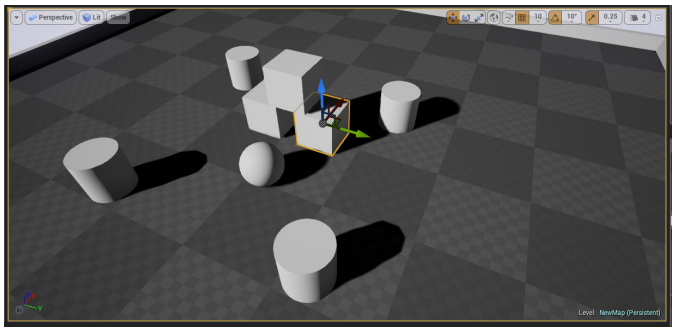

Abbildung 4.23: (Unreal) Hindernisse hinzugefügt

Sollen auch die Würfel von Kollisionen beeinflusst werden, muss man den Würfel auswählen und anschließend das Häkchen im Details Panel bei Simulate Physics setzen und das Gewicht des Würfels anpassen. Weiters muss man noch Physical Materials erstellen und den jeweiligen Meshes zuorden. Das Physical Material lässt sich unter Collision im Details Panel auswählen. Da kein Starter Content ausgewählt wurde ist die Liste noch leer. Jedoch kann man im offenen Auswahlfenster gleich ein neues Material erstellen, welches sofort dem ausgewählten Mesh zugeordnet wird. Alternativ kann man das Material einfach erstellen, indem man im Contentbrowser rechtsklickt und unter Physics das Physical Material auswählt. Anschließend muss das Material noch dem gewünschten Mesh zugeordnet werden. Nach einem Doppelklick auf das Physical Material öffnet sich ein Fenster mit den einstellbaren Parametern (Abbildung 4.24). Wir wollen, wie auch beim Unity Beispiel, die "Bounciness", die sich hier "Restitution" nennt, erhöhen sowie die "Friction" etwas heruntersetzen. Für die "Friction" nehmen wir einen Wert von 0,4 und für "Restitution" einen Wert von 1. Wenn man nun die Würfel auswählt und ihnen das erstellte Physical Material zuordnet, lassen sie sich schön über die Plane schieben.

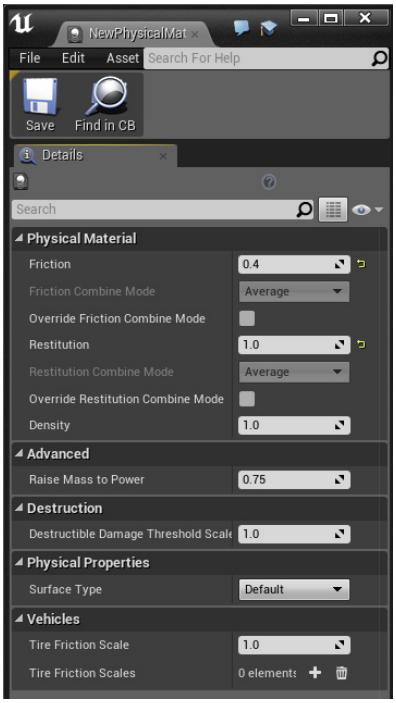

Abbildung 4.24: (Unreal) Neues Physics Material

4.4.3 Fazit

Wie bereits beim Level Design ist auch hier das Hinzufügen von neuen Objekten in der Szene sowohl in Unity als auch Unreal sehr simple. Zusätzlich kommt ein Physik-Material zum Einsatz, das in beiden Engines ähnlich gehandhabt wird.

4.5 User Interface

Im folgenden Abschnitt wird ein einfaches Menü in das Spiel eingebaut, welches die Möglichkeit bieten soll, das Spiel (neu) zu starten, zu beenden, anzuhalten und fortzusetzen.

4.5.1 Unity 5

Für die Darstellung eines Menüs brauchen wir zunächst ein Canvas, auf dem es abgebildet werden soll. Dieses lässt sich einfach über "GameObject/UI/Canvas" erstellen und erscheint automatisch in unserem Hierarchy-Fenster. Auf dieselbe Weise lassen sich weitere UI-Elemente hinzufügen. Über "GameObject/UI/Button" erstellen wir nun 3 Schaltflächen, welche wir vertikal anordnen und eventuell in Höhe und Breite anpassen. Diese Elemente sollen unsere Play-, Restart- und Quit-Buttons repräsentieren und brauchen die entsprechenden Beschriftungen. Dazu wählen wir im Hierarchie-Fenster jeden Button einzeln aus und fügen ein Text-Element über "GameObject/UI/Text" hinzu. Anschließend tragen wir die passende Beschriftung in das Text-Feld im Inspector ein (Abbildung 4.25).

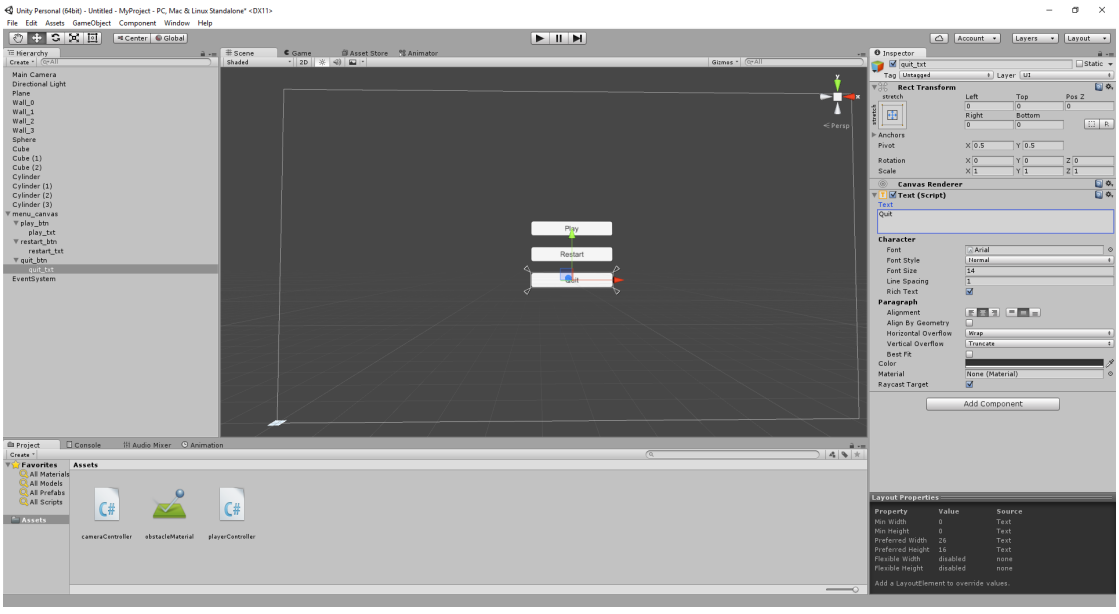

Abbildung 4.25: (Unity) UI Canvas

Im nächsten Schritt werden wir ein Script erstellen, welches die Logik hinter unserem Menü steuern soll. Dazu öffnen wir im Project-Browser per Rechtsklick das PopUp-Menü und wählen den Punkt "Create/C# Script" aus. Nachdem der Name der Datei entsprechend gesetzt wurde, öffnen wir diese per Doppelklick in einem separaten Editor. Wie üblich wird hier unser Code für die Menü-Steuerung eingefügt. 4.3

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;
using UnityEngine.SceneManagement;

public class menuScript : MonoBehaviour {
public Canvas mainMenu;
public Button btnPlay;
public Text txtPlay;
public Button btnRestart;
public Button btnQuit;
// Use this for initialization
void Start () {

mainMenu = mainMenu.GetComponent<Canvas>();
btnPlay = btnPlay.GetComponent<Button>();
txtPlay = txtPlay.GetComponent<Text>();
btnRestart = btnRestart.GetComponent<Button>();
btnQuit = btnQuit.GetComponent<Button>();

mainMenu.enabled = true;
btnPlay.gameObject.SetActive(true);
btnRestart.gameObject.SetActive(false);
Time.timeScale = 0;
txtPlay.text = "Play";
}

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

if (Input.GetKeyDown("p")&&!mainMenu.enabled)
{
mainMenu.enabled = true;
txtPlay.text = "Resume";
btnRestart.gameObject.SetActive(true);
Time.timeScale = 0;
}
}

public void pressPlay()
{
mainMenu.enabled = false;
startGame();
}

public void pressRestart()
{
```

```
restartGame();
}

public void pressQuit()
{
quitGame();
}

public void startGame()
{
Time.timeScale = 1;
}

public void restartGame()
{
SceneManager.LoadScene("level0");
}

public void quitGame()
{
Application.Quit();
}
}
```

Listing 4.3: Menü-Steuerung

Zuerst deklarieren wir oben alle Elemente, die wir verwenden möchten; ein canvas, drei buttons, ein text. In der `Start()`-Methode werden diese mit den entsprechenden Komponenten initialisiert. Das Enabled Attribut unseres Canvas setzen wir auf "true" damit unser Menü direkt nach dem Start angezeigt wird. Allerdings gilt das nicht für unseren Restart-Button. Dieser soll erst beim Pausieren des Games sichtbar sein und wird somit in der `Start()`-Methode deaktiviert. Des Weiteren verwenden wir `Time.timeScale = 0`, um das Spielgeschehen anzuhalten während unser Menü angezeigt wird. Die Referenz zum Text-Element wird in unseren Script verwendet, um den Text des Play-Buttons zu verändern; "Resume" beim Pausieren und "Play" beim Start oder Neustart.

In der `Update()`-Methode wollen wir überprüfen, ob die Pause-Taste betätigt wurde, um das Spiel anzuhalten und das Menü anzuzeigen. Dabei wird, wie bereits oben erwähnt, der Text des Play-Buttons in "Resume" umgeändert. Außerdem aktivieren wir unseren Restart-Button, damit der Spieler die Möglichkeit bekommt das Game, neu zu starten.

Im folgenden Schritt werden wir die `OnClick()`-Methoden für unsere Buttons beschreiben. Diese sind recht einfach und selbst erklärend. Wir haben eine `pressPlay()`-Methode, welche beim betätigen des Play-Buttons ausgeführt werden soll. Diese stellt die Sichtbarkeit des Menüs auf "false" und führt anschließend `startGame()` aus, welches unser Spiel startet bzw. fortsetzt. Die `pressRestart()`-Methode zusammen mit `restartGame()` dienen dazu, das Level bzw. die Szenen neu zu laden. Mit der `pressQuit()`- und `quitGame()`-Methode wird die Spiel-Anwendung geschlossen. Dies lässt sich allerdings nicht in Unity selbst testen, da dieses Verfahren den gesamten Editor beenden und somit nicht gespeicherte Änderungen verloren gehen würden.

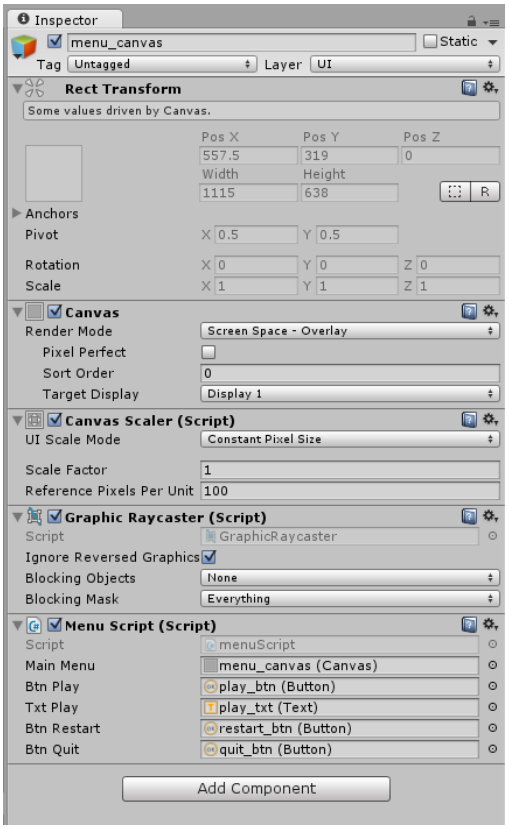

Abbildung 4.26: (Unity) Canvas Skript hinzugefügt

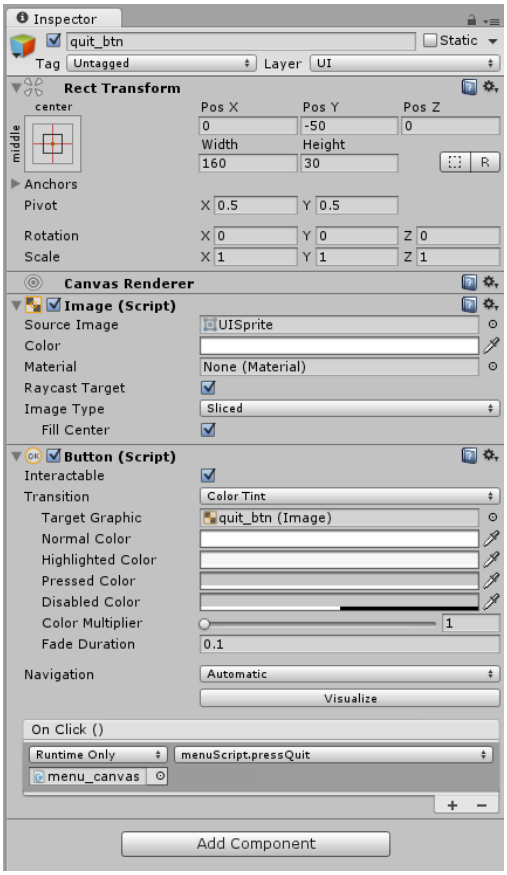

Abbildung 4.27: (Unity) onClick Event hinzugefügt

Nachdem wir unser fertiges Script abgespeichert haben, wechseln wir zu Unity und ziehen die Datei in unser Canvas-Objekt, damit diese als Script-Komponente hinzugefügt wird. Anschließend müssen unsere 3 Buttons sowie das Text-Objekt in die entsprechenden Variabel-Felder gezogen werden, damit eine Referenz aufgebaut werden kann (Abbildung 4.26). Zu guter Letzt bleiben noch die Button-Objekte, denen wir gewisse Funktionalitäten aus unserem Skript zuordnen müssen. Dazu wählen wir den jeweiligen Button aus und klicken im Inspector unter dem Punkt `OnClick` auf das Plus-Symbol, um ein neues `OnClick`-Event zu erstellen. In das Objekt-Feld mit dem Platzhalter "None (Object)" muss das Canvas-Element gezogen werden. Danach erhält man über das DropDown-Menü Zugriff auf unser Skript und kann die passende Methode auswählen (Abbildung 4.27). Zum Schluss müssen wir unsere Szene unter dem Namen, den wir im Skript in der `Restart`-Methode verwendet haben ("level0"), abspeichern und das Lighting von der Szene unter "Window/Lighting" einmal builden (Abbildung 4.28, 4.29).

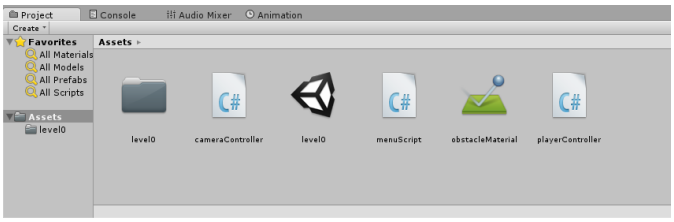

Abbildung 4.28: (Unity) Level abgespeichert

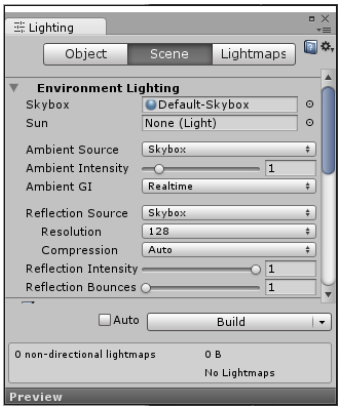

Abbildung 4.29: (Unity) Beleuchtung

4.5.2 Unreal Engine 4

Um ein Menü zu erstellen braucht man ein Widget Blueprint, das man über den Content Browser hinzufügen kann. Man rechtsklickt dazu in den Content Browser und wählt unter User Interface den Widget Blueprint aus. In diesem Beispiel heißt das Widget Main_Menu. Über einem Doppelklick auf das Main_Menu-Widget gelangt man in den UMG UI Designer (Abbildung 4.30).

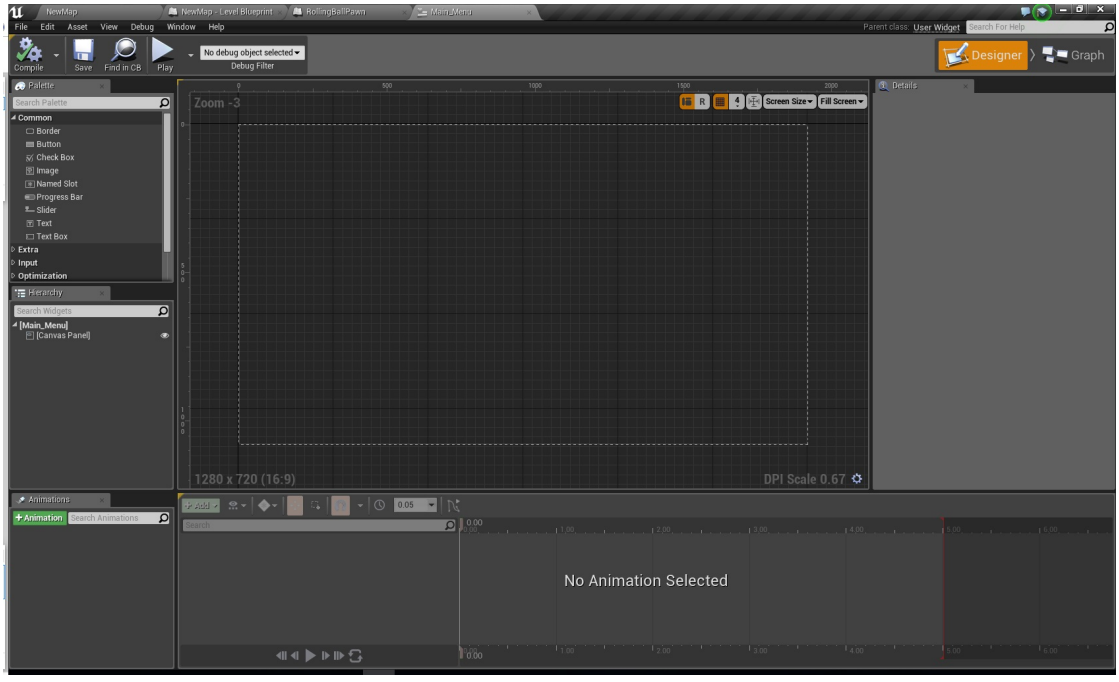

Abbildung 4.30: (Unreal) UI-Canvas

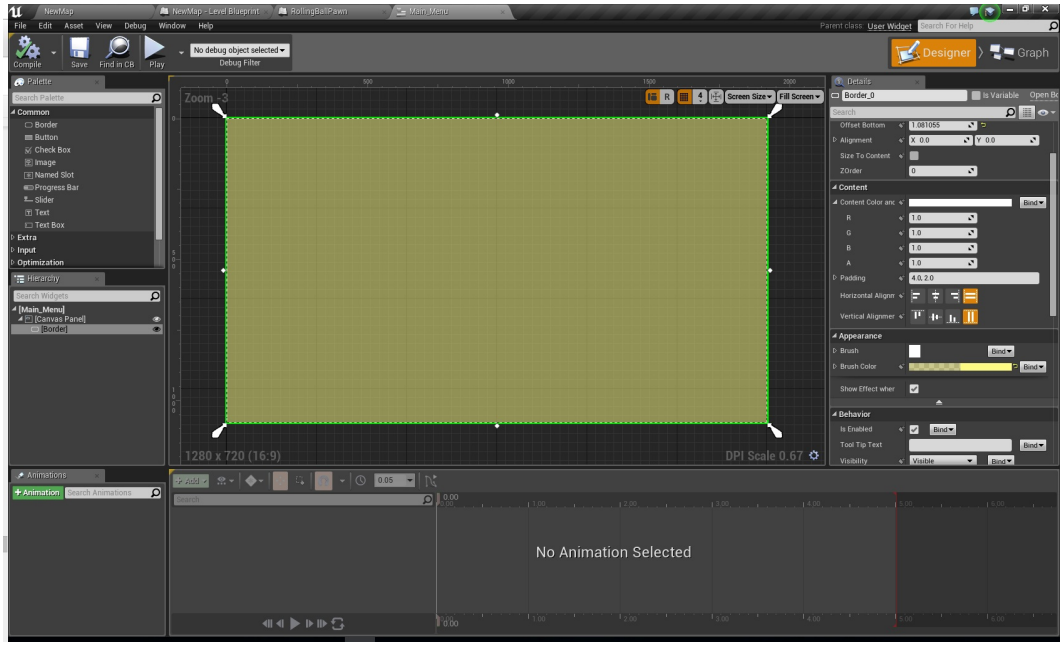

Abbildung 4.31: (Unreal) Canvas Rahmen

Hier fügen wir zur Hierarchie einige Elemente hinzu, um das Menü aufzubauen. Als erstes fügen wir dem Canvas Panel in der Hierarchie einen Border hinzu. Ein Border ist ein Container, der diverse Einstellungen erlaubt und als Basis für das Menü dienen wird. Wir werden den Border jetzt etwas an unsere Bedürfnisse anpassen. Wir wollen, dass der Border die gesamte Bildfläche bedeckt. Dazu ziehen wir den Border soweit auf, dass er das gesamte Canvas Panel bedeckt. Alternativ kann man auch im Details Panel unter Slot die Größe des Borders händisch eingeben, in unserem Fall 1980 x 1080. Als nächstes stellen wir ein, wo der Border im Bild verankert werden soll. Dazu klickt man im Details Panel auf Anchors und sucht sich eine passende Verankerung aus, in unserem Fall das ganze Bild, also das Element ganz rechts unten. Wir geben dem Menü jetzt ein bisschen Farbe, dazu klickt man unter Appearance auf die Brush Color und wählt eine beliebige Farbe aus (Abbildung 4.31). Damit man, wenn man das Menü aufruft, das Spiel noch sieht, stellt man das Alpha auf 0.5. Als nächstes fügen wir dem Border in der Hierarchie eine Vertical Box hinzu und zentrieren sie im Border. Zentrieren kann man die Box indem man sie in der Hierarchie auswählt und im Details Panel unter Slot sowohl das Horizontale als auch das vertikale Alignement auf Center stellt (Abbildung 4.32). Der Vertical Box fügen wir nun drei Buttons hinzu und jeder Button bekommt als Child ein Text-Element (Abbildung 4.33). Jedem Button werden wir nun einen Sinnvollen Namen geben. Dazu wählt man einen Button aus und trägt im obersten Feld im Details Panel einen Namen ein. Bei uns sind das `PlayBtn`, `RestartBtn` und `QuitBtn`. Auch dem Text-Element des `PlayBtn` geben wir einen Namen. Wir nennen das Element `PlayText` (Abbildung 4.34).

Abbildung 4.32: (Unreal) Menü Ausrichtung

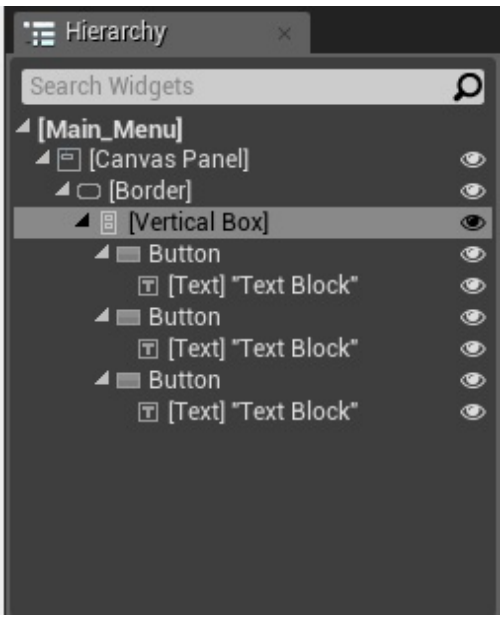

Abbildung 4.33: (Unreal) Menü Aufbau

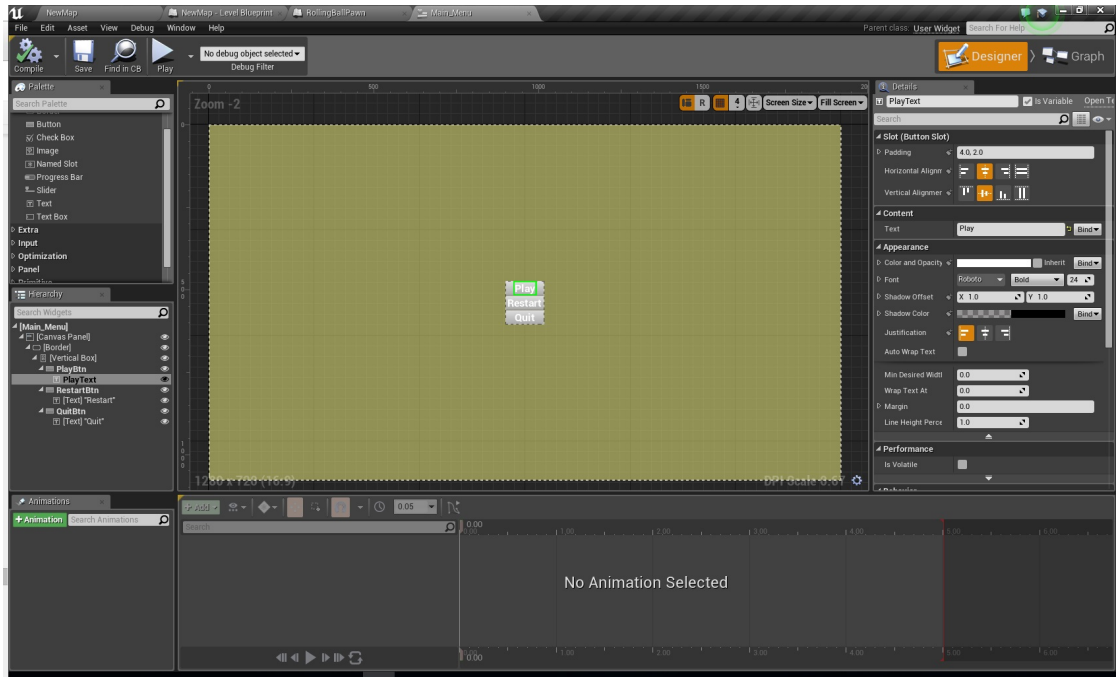

Abbildung 4.34: (Unreal) Menü Schaltflächen

Später werden wir den Text des `PlayBtn` manipulieren, daher ist es ratsam dem Text-Element einen Namen zu geben, um es leichter zu finden. Nun stellen wir noch den Text in den Text-Elementen ein. Dazu wählt man die Text-Elemente aus und gibt ihnen einen passenden Namen.

Es ist auch noch wichtig einzustellen, welche Elemente als Variable zur Verfügung stehen sollen, da man sonst nicht auf die Elemente zugreifen kann, was wir benötigen werden. Die Checkbox, mit der ein Element als Variable deklariert werden kann, findet man im Details Panel rechts neben dem Namen des Elements. Wir deklarieren alle Buttons und den PlayText als Variablen. Weiters stellen wir den `RestartBtn` standardmäßig auf Invisible, da wir ihn später selber auf Visible setzen werden. Dazu setzen wir beim `RestartBtn` im Details Panel unter Behavior die Visibility auf Hidden. Nun wechseln wir in den Visual Programming Bereich des UMG UI Designer. Dazu klicken wir rechts oben in der Ecke einfach auf Graph. Wie auch schon in den anderen Teilen dieses Beispielprojektes in der Unreal Engine fügen wir dem Graphen die nötigen Funktionen hinzu (Abbildung 4.35).

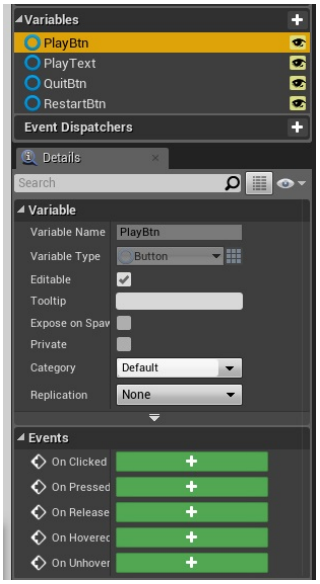

Abbildung 4.35: (Unreal) Schaltflächen Eigenschaften

Nach einem Klick auf die `PlayBtn` Variable findet man im Details Panel eine Liste von Events, die durch Interaktion mit dem Button ausgeführt werden können. Unter anderem auch ein `OnClicked`-Event. Nach einem Klick auf das Plus Symbol erscheint das Event im Graphen. Wir fügen auch bei den anderen Buttons die `OnClick`-Events zum Graphen hinzu. Nun erstellen wir die Logik. Wir wollen, dass nach einem Klick auf den `PlayBtn` das Menü verschwindet und, dass aus dem Pausenmodus in den Spielmodus gewechselt wird. Daher ziehen wir eine Linie vom `OnClicked(PlayBtn)` Event und erstellen einen `"Remove from Parent"`-Knoten. Von diesem ziehen wir eine Linie und erstellen einen `"Set Game Paused"`-Knoten, welcher die Checkbox "Paused" auf "false" gesetzt hat. Wenn wir auf den "QuitBtn" klicken soll das Spiel auch wirklich beenden, daher erstellen wir vom `OnClicked(QuitBtn)` Event einen `"Execute Console Command"`-Knoten, der den Command-Parameter "quit" bekommt. Der `RestartBtn` soll das Level neu laden. Dazu erstellen wir vom `OnClicked(RestartBtn)` Event einen `"Open Level"`-Knoten und tragen als "Level Name" den Namen unserer Map ein (Abbildung 4.36).

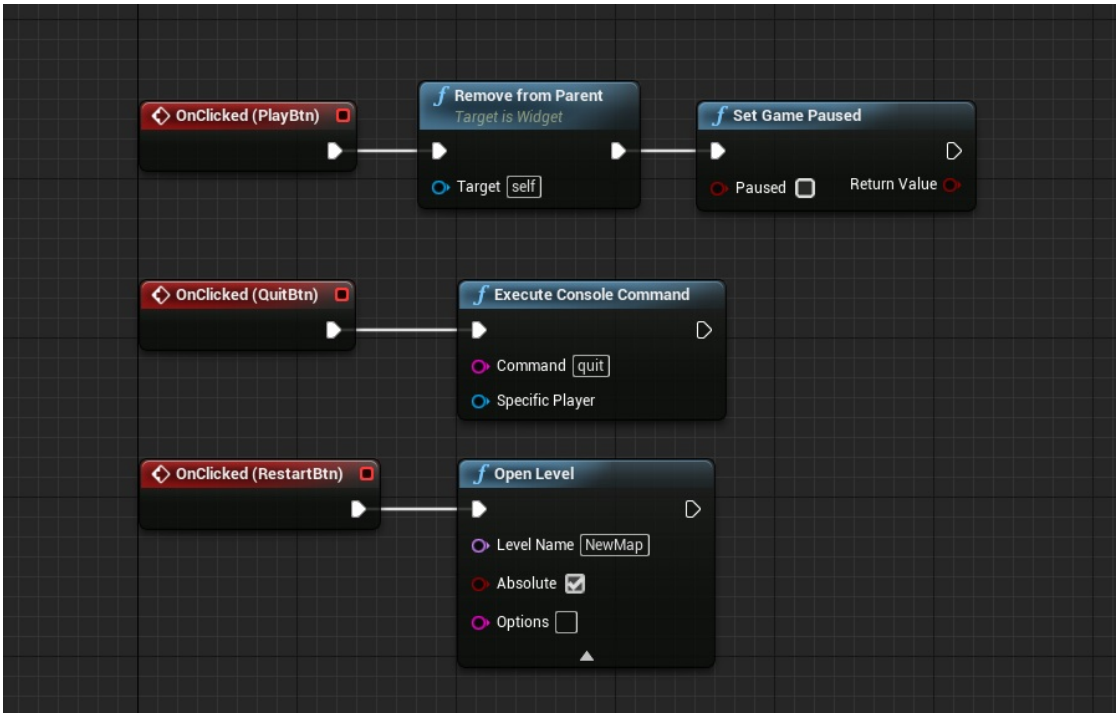

Abbildung 4.36: (Unreal) Schaltflächen Events

Damit wir das Spiel auch wirklich pausieren können und es im pausierten Modus startet, müssen wir noch ein bisschen Logik zu unserem Level hinzufügen. Um den Blueprint des Levels zu bearbeiten, wechseln wir in das Hauptfenster des Unreal Engine Editors und öffnen den Blueprint über das Blueprint Menü, das sich über dem Viewport befindet. Wir wollen das Menü an das Level anhängen, außerdem soll das Level nach dem Laden im Pausenmodus starten. Weiters wollen wir eine Taste definieren, mit der sich das laufende Spiel pausieren lässt. Vom Event BeginPlay aus erstellen wir den Knoten `"Create Menu"` und wählen als Class das Main Menu aus. Das erstellte Widget wollen wir nun zum Viewport hinzufügen. Dazu erstellen wir vom Create-Knoten aus einen `"Add to Viewport"`-Knoten und verbinden noch den Return Value des Widgets mit dem `"Add to Viewport"`-Target. Wir wollen das Menü mit der Maus bedienen können, und zwar mit unserem Player Controller, daher brauchen wir zunächst den `"Get Player Controller"`-Knoten. Vom Player Controller Return Value aus erstellen wir einen `"Set Show Mouse Cursor"`-Knoten und stellen sicher, dass die Checkbox aktiviert ist. Der `"Add to Viewport"`-Knoten wird daraufhin noch mit dem `"Set"`-Knoten verknüpft.
Als letztes wollen wir das Spiel im pausierten Modus starten, daher brauchen wir vom Set-Knoten aus noch einen `"Set Game Paused"`-Knoten in dem die Paused-Checkbox aktiviert ist. Um das Spiel mit einer Taste Pausieren zu können fügen wir dem Graphen ein Tasten- Event hinzu. Man kann sich jede beliebige Taste aussuchen, in unserem

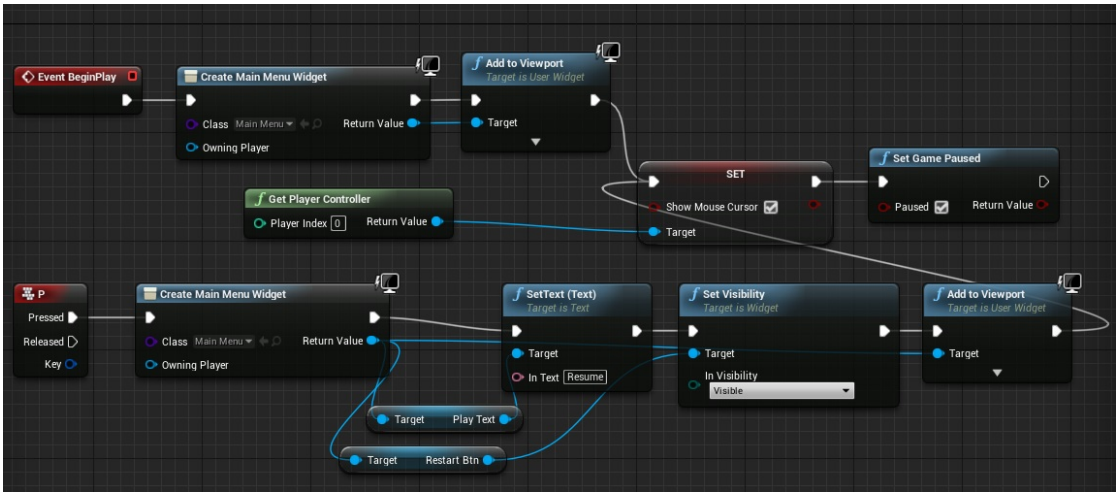

Abbildung 4.37: (Unreal) Logik fürs Pausieren

Beispiel verwenden wir die P Taste. Nach einem Rechtsklick in den Graphen, suchen wir über das Popup-Menü nach "p". Uns wird sofort ein Tasten Event für die Taste P vorgeschlagen. Von den "Pressed" Attribut aus erstellen wir ein `"Create Widget"`, das wir, wie schon oben beschrieben mit der Main Menu Klasse versehen. Als nächstes sorgen wir dafür, dass der `PlayBtn` nach drücken unserer Pause Taste den Text "Resume" trägt. Zu diesem Zweck erstellen wir von Return Value des Create-Knotens einen `"Get PlayText"`-Knoten, von dem aus wir einen `"Set Text"`-Knoten erstellen, dem wir als Parameter Text "Resume" angeben. Der Create Knoten wird mit dem `"Set Text"`-Knoten verbunden. Weiters wollen wir, dass der `RestartBtn` angezeigt wird. Daher erstellen wir vom Return Value des Create-Knotens einen `"Get RestartBtn"`-Knoten, von dem aus wir einen `"Set Visibility"`-Knoten erstellen, in dem wir die Visibility auf "Visible" setzen. Nun wird noch der `"Set Text"`-Knoten mit dem Set Visibility-Knoten verbunden. Wir wollen auch, dass das Menü angezeigt wird, daher brauchen wir noch einen `"Add To Viewport"`-Knoten, den wir mit dem Visibility-Knoten verbinden und der als Target mit dem Return Value des Create-Knotens verbunden wird. Das letzte was wir noch verbinden müssen, um die Pause Taste fertigzustellen, ist den gerade erstellen `"Add To Viewport"`-Knoten mit dem `"Set Show Mouse Cursor"`-Knoten zu verbinden (Abbildung 4.37). Nach einem Klick auf die Play Taste kann man das Ergebnis sehen und ausprobieren.

4.5.3 Fazit

Der Vorgang fürs Erstellen des Menüs ist in beiden Engines ziemlich gleich, mit dem Unterschied, dass die Logik im Hintergrund in Unity mit einem C#-Script und in Unreal durch Visual Scripting innerhalb des Blueprints umgesetzt wurde.

4.6 Collectibles

In diesem Abschnitt werden wir ein einsammelbares Objekt in Form eines Karos in unser Spiel einbauen. Dieses soll sich um die vertikale Achse drehen und bei einer Kollision die Position auf der Map zufällig ändern.

4.6.1 Unity 5

In Unity erstellen wir zunächst einen neuen Würfel, den wir anschließend so anpassen, dass er die Form eines flachen, über dem Boden schwebenden Karos annimmt. Dazu wird seine Skalierung, Rotation und Position entsprechend im Inspector angepasst (Abbildung 4.38).

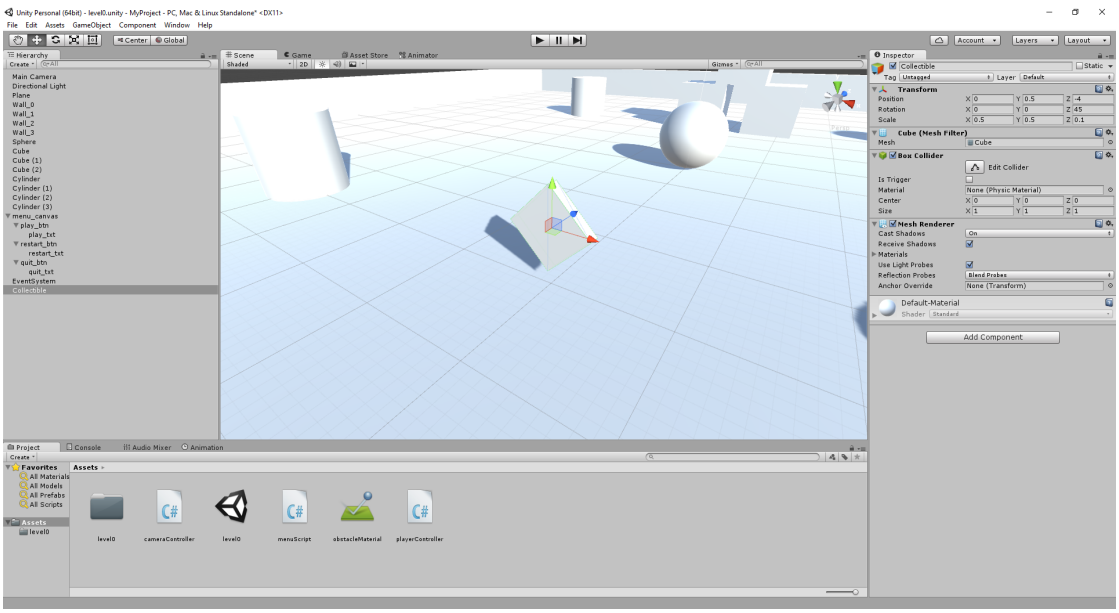

Abbildung 4.38: (Unity) Collectible hinzugefügt

Als nächstes benötigen wir ein Script, welches das Verhalten des Karos beschreibt. Dieses soll sich laut unserer Angabe um die eigene vertikale Achse Y drehen und bei einer Kollision mit dem Spieler die Position zufällig wechseln. Wir erstellen wie gewohnt ein neues C#-Script, welches wir unserem Karo-Objekt als Komponente hinzufügen. 4.4

```
using UnityEngine;
using System.Collections;

public class collectibleController : MonoBehaviour {

// Use this for initialization
void Start () { }

// Update is called once per frame
void Update () {
transform.Rotate(new Vector3(0.0f, 45.0f, 0.0f) * Time.deltaTime, Space.World);
}

void OnTriggerEnter(Collider other) {
transform.position = new Vector3(Random.Range(-9.0f, 9.0f),
                                 0.5f, Random.Range(-9.0f, 9.0f));
}
}
```

Listing 4.4: Karo-Objekte hinzufügen

In der `Update()`-Methode führen wir die Rotation des Objekts durch, indem wir "transform.Rotate" mit den passenden Werten einsetzen. Da unser Würfel bereits in der Szene um 45° rotiert wurde, müssen wir zusätzlich als letzten Paramter `Space.World` übergeben, damit die Rotation um die globale Achse erfolgt. Für die Neupositionierung des Karos verwenden wir eine `Event`-Methode von Unity namens `OnTriggerEnter`. Diese wird ausgelöst, sobald es zu einer Kollision von zwei Objekten kommt. In der Methode generieren wir zufällige X- und Z-Koordinaten, welche die neue Position unseres Collectibles bestimmen. Der Umfang dieser beiden Werte wird so gewählt, dass das Endresultat innerhalb unserer Map bleibt. Zum Schluss müssen wir in Unity unser Karo-Objekt auswählen und im Inspector den Punkt "Is Trigger" einschalten, damit unsere Methode auch tatsächlich ausgelöst werden kann (Abbildung 4.39).

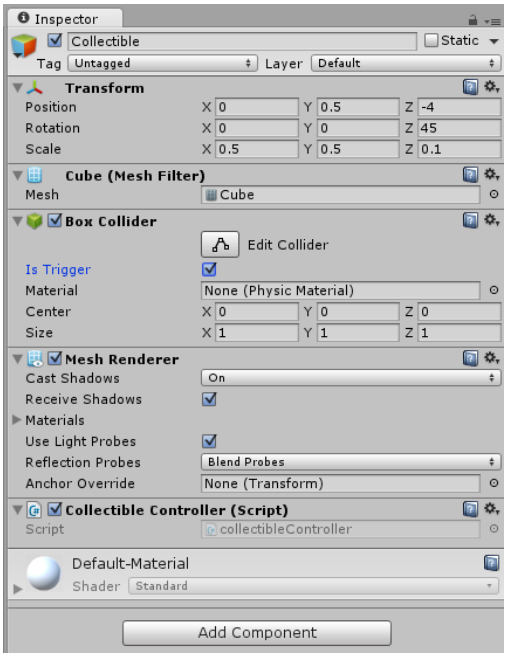

Abbildung 4.39: (Unity) Collectible Eigenschaften

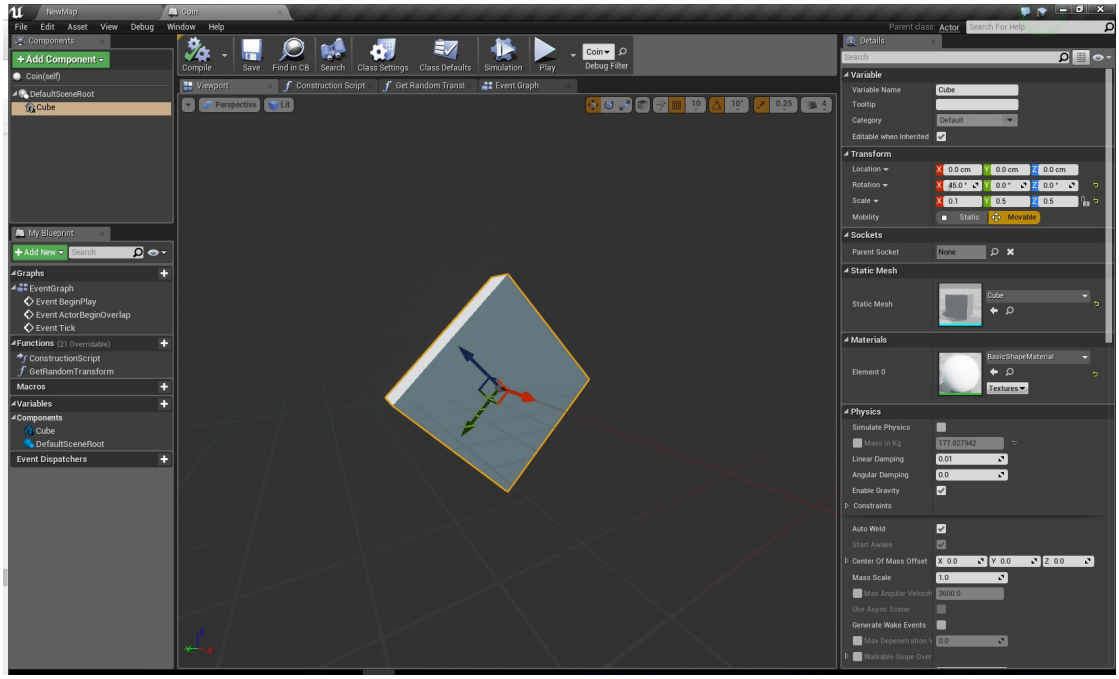

Abbildung 4.40: (Unreal) Collectible Blueprint

4.6.2 Unreal Engine 4

Für die Collectibles in UE 4 erstellen wir ein Actor-Blueprint über einen Rechtsklick im Content Browser und nennen ihn Coin. Nach einem Doppelklick auf den neu erstellen Blueprint gelangen wir in den Blueprint Editor, wo wir dem "DefaultSceneRoot" im Components Panel einen Cube hinzufügen, den wir wie auch schon im Unity Beispiel zurechtbiegen. Weiters setzen wir die Rotation der X-Achse auf 45° (Abbildung 4.40). Damit man die Coin auch einsammeln kann, muss im Details Panel unter Collision das Collision Preset "Overlap All" und das Häkchen bei "Generate Overlap Events" gesetzt werden.

Damit sind wir fertig mit der Erstellung des Meshes für unsere Coin. Jetzt müssen wir die Coin nur noch dazu bringen sich zu drehen. Dazu brauchen wir die passende Logik im Event Graph. Ausgehend vom Event Tick erstellen wir einen "`AddRelativeRotation(Cube)`"-Knoten. Als Target sollte automatisch der Cube aus den Components referenziert werden. Als Delta Rotation stellen wir für die Z-Achse einen Wert von 2 ein (Abbildung 4.41). Weiters müssen wir auch noch die Logik für die Kollision mit der Coin erstellen und dafür sorgen, dass sie bei einer Kollision die Location in unserem Spielfeld wechselt. Zuerst erstellen wir dafür eine Funktion. Über das Plus unter MyBlueprint beim Reiter Functions wird eine neue Funktion erstellt. Wir nennen sie "GetRandomLocation". In der Funktion ändern wir im Details Panel den Output Value auf Vector, und nennen den Parameter "Location". Jetzt erstellen wir zwei "`Random Float in Range`"-Knoten, denen wir

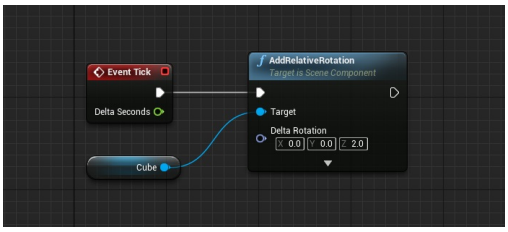

Abbildung 4.41: (Unreal) Collectible Logik 1

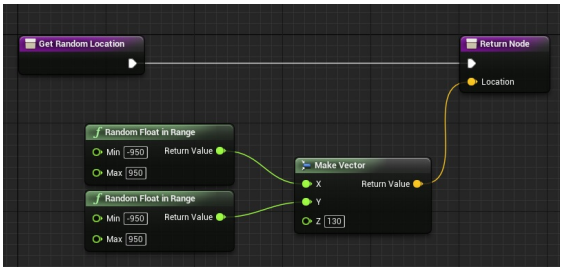

Abbildung 4.42: (Unreal) Collectible Logik 2

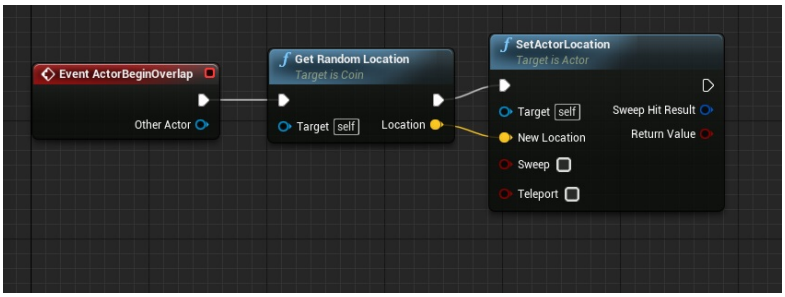

Abbildung 4.43: (Unreal) Collectible Logik 3

eine etwas kleinere Fläche als unseren Floor in unserer Map angeben, damit die Coin innerhalb unseres Spielfeldes bleibt. Jetzt erstellen wir einen `"Make Vector"`-Knoten bei dem wir jeweils einen der `"Random Float in Range"`-Konten als X bzw. Y Value angeben. Als Z Value Tragen wir 130 ein damit die Coin ein bisschen über dem Boden schwebt. Den Return Value verbinden wir mit der Return Node. Das Ergebnis ist in Abbildung 4.42 zu sehen. Wir wechseln zurück in den Event Graphen und erstellen die Kollisions Logik. Vom Event `"ActorBeginOverlap"` erstellen wir einen Knoten mit unserer gerade zuvor erstellten Funktion. Vom `"GetRandomLocation"`-Knoten erstellen wir einen `"SetActorLocation"`-Knoten, der als Input die Location der Funktion bekommt (Abbildung 4.43). Damit die Coins nicht in die Säulen verschoben werden, muss bei jeder der Säulen im Details Panel unter Collision noch das Häkchen bei "Generate OverLap Events" gesetzt werden.

4.6.3 Fazit

Auch in diesem Schritt liegt der wesentliche Unterschied zwischen den beiden Engines darin, dass auf der einen Seite mit C#-Scripts gearbeitet wird und auf der anderen erneut das Visual Scripting in den Blueprints zum Einsatz kommt.

4.7 Score und Timer

Um den Spielererfolg zu messen brauchen wir eine Punkteanzeige und eine Lose-Bedingung. Ziel soll es sein, so viele Objekte wie möglich aus dem vorherigen Abschnitt einzusammeln, wobei jedes eingesammelte Objekt den Timer um fünf Sekunden und die Punkte um Eins anhebt. Erreicht der Timer Null ist das Spiel vorbei.

4.7.1 Unity 5

Für unsere Anzeigen benötigen wir in Unity ein neues Canvas, das wir über "GameObject/ UI/Canvas" erstellen. Diesem fügen wir drei Text-Elemente hinzu, welche die Timer-, Punkte- und GameOver-Anzeige darstellen sollen. Position, Größe und andere Eigenschaften werden so angepasst wie sie in Abbildung 4.44 zu sehen sind. Oben links befindet sich unsere Punkte-Anzeige, oben mittig ist der Timer abgebildet. Über dem Hauptmenü wird der GameOver-Text eingeblendet. Nun erstellen wir ein neues C#-Script, welches unsere neuen Text-Elemente steuern soll. 4.5

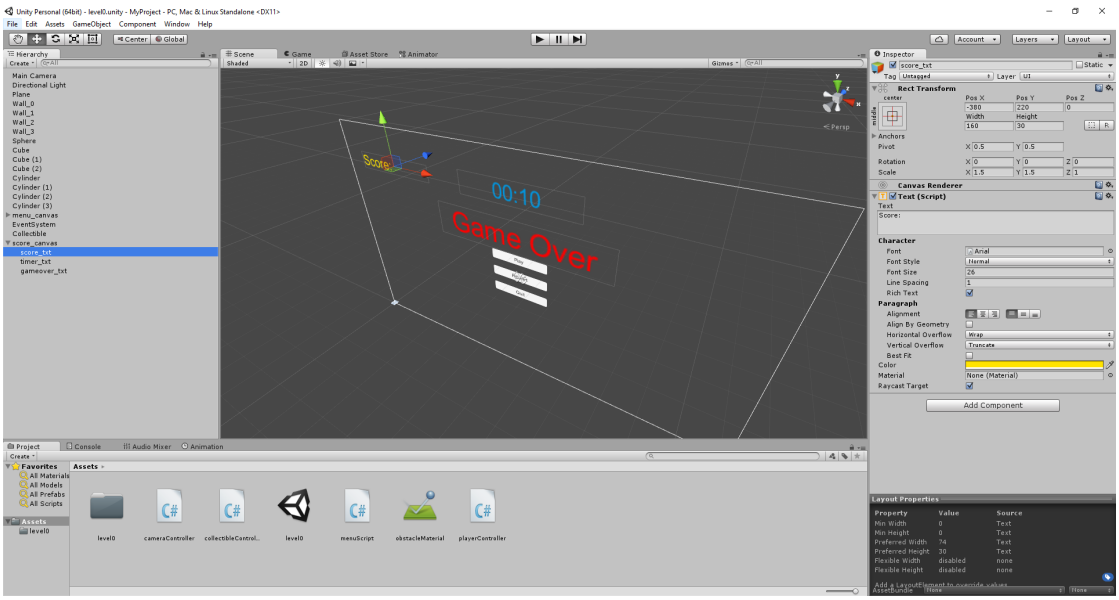

Abbildung 4.44: (Unity) Neues Canvas

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;

public class scoreScript : MonoBehaviour {

public Canvas scoreCanvas;
public Text txtScore;
public Text txtTimer;
public Text txtGameOver;
public static int score = 0;
public static float timeLeft = 10.0f;
// Use this for initialization
void Start () {
scoreCanvas = scoreCanvas.GetComponent<Canvas>();
txtScore = txtScore.GetComponent<Text>();
txtTimer = txtTimer.GetComponent<Text>();
txtGameOver = txtGameOver.GetComponent<Text>();
txtGameOver.enabled = false;
}

// Update is called once per frame
void Update () {
txtScore.text = "Score: " + score;
timeLeft -= Time.deltaTime * Time.timeScale;
int minutes = (int)(timeLeft / 60.0f);
int seconds = Mathf.CeilToInt(timeLeft % 60.0f);
txtTimer.text = string.Format("{0:00}:{1:00}", minutes, seconds);
if (timeLeft <= 0.0f) txtGameOver.enabled = true;
}
}
```

Listing 4.5: Score und Timer

Am Anfang deklarieren wir wie üblich unsere UI-Elemente (1x Canvas & 3x Text). Außerdem erstellen wir zwei globale Variablen für unseren Punktestand und die übergebliebene Zeit. In der `Start()`-Methode initialisieren wir unsere Elemente und blenden den GameOver-Text ein. Die `Update()`-Methode enthält den Algorithmus, durch welchen die Zeit- und Punkte-Anzeige aktualisiert werden. Außerdem wird im Falle, dass die übrig geblieben Zeit 0 erreicht, der GameOver-Text eingeblendet. Nachdem das Skript fertiggestellt wurde, muss es nur noch in Unity dem passenden Canvas als Komponente hinzugefügt und die Variablen-Felder entsprechend gesetzt werden. Damit wären wir hier fertig, müssen allerdings noch ein paar Änderungen in anderen Skripten vornehmen. 4.6

```
using UnityEngine;
using System.Collections;

public class collectibleController : MonoBehaviour {

// Use this for initialization
void Start () { }

// Update is called once per frame
void Update () {
transform.Rotate(new Vector3(0.0f, 45.0f, 0.0f) * Time.deltaTime, Space.World);
}

void OnTriggerEnter(Collider other) {
if (other.gameObject.CompareTag("Player"))
{
scoreScript.score += 1;
scoreScript.timeLeft += 5.0f;
}
transform.position = new Vector3(Random.Range(-9.0f, 9.0f), 0.5f, Random.Range(-9.0f, 9.0f));
}
}
```

Listing 4.6: Karo-Code anpassen

In unserem Skript für das Karo fügen wir in der `OnTriggerEnter()`-Methode eine zusätzliche `if`-Abfrage hinzu, welche überprüft, ob es sich beim kollidierenden Objekt um den Spieler handelt und die Punktezahl erhöht wird. 4.7

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;
using UnityEngine.SceneManagement;

public class menuScript : MonoBehaviour {
public Canvas mainMenu;
public Button btnPlay;
public Text txtPlay;
public Button btnRestart;
public Button btnQuit;
// Use this for initialization
void Start () {

mainMenu = mainMenu.GetComponent<Canvas>();
btnPlay = btnPlay.GetComponent<Button>();
txtPlay = txtPlay.GetComponent<Text>();
btnRestart = btnRestart.GetComponent<Button>();
btnQuit = btnQuit.GetComponent<Button>();

mainMenu.enabled = true;
btnPlay.gameObject.SetActive(true);
btnRestart.gameObject.SetActive(false);
Time.timeScale = 0;
txtPlay.text = "Play";
```

```
scoreScript.score = 0;
scoreScript.timeLeft = 10.0f;
}

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

if (Input.GetKeyDown("p")&&!mainMenu.enabled)
{
mainMenu.enabled = true;
txtPlay.text = "Resume";
btnRestart.gameObject.SetActive(true);
Time.timeScale = 0;
}
if (scoreScript.timeLeft <= 0.0f)
{
mainMenu.enabled = true;
btnPlay.gameObject.SetActive(false);
btnRestart.gameObject.SetActive(true);
Time.timeScale = 0;
}
}

public void pressPlay() {
mainMenu.enabled = false;
startGame();
}

public void pressRestart() {
restartGame();
}

public void pressQuit() {
quitGame();
}

public void startGame() {
Time.timeScale = 1;
}

public void restartGame() {
SceneManager.LoadScene("level0");
}

public void quitGame() {
Application.Quit();
}
}
```

Listing 4.7: Menü-Code anpassen

In unserem Hauptmenü-Skript werden ebenfalls ein paar Zeilen hinzugefügt. In der `Start()`-Methode möchten wir, dass sowohl die Zeit als auch die Punkte bei einem Start

oder Neustart zurückgesetzt werden. In der `Update()`-Methode bauen wir eine zweite `if`-Abfrage ein, welche das Spiel beim Ablauf der Zeit anhält und die entsprechenden Menü-Elemente einblendet. Nachdem wir alle Skript-Änderungen durchgeführt haben, kehren wir zum Unity-Editor zurück und ziehen unser neu erstelltes Skript ("scoreScript") in das "score_canvas"-Element damit es als Komponente hinzugefügt wird. Anschließend müssen die Variablen- Felder der Skript-Komponente entsprechend belegt werden (Abbildung 4.45). Zu Abschluss wählen wir unsere Spielerkugel aus und setzen im Inspector oben den Tag von "Untagged" auf "Player" damit unsere Kugel beim Kollisions-Event des Karo-Skripts erkannt wird und sowohl Punkteanzahl als auch Timer erhöht werden können (Abbildung 4.46).

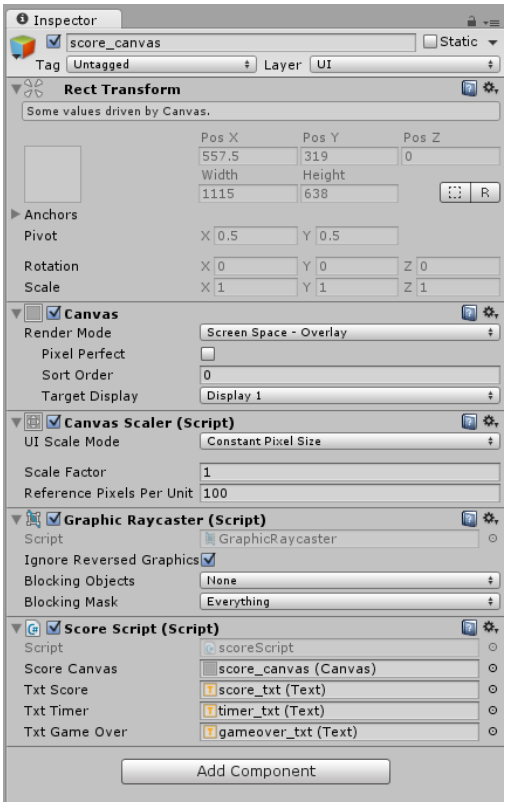

Abbildung 4.45: (Unity) Canvas Eigenschaften

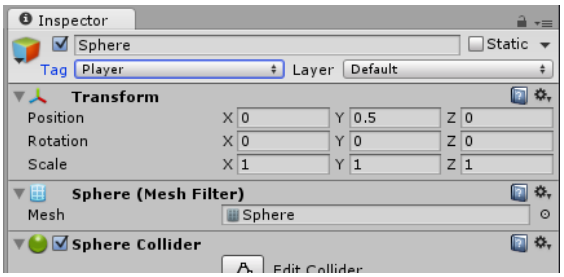

Abbildung 4.46: (Unity) Kugel getaggt

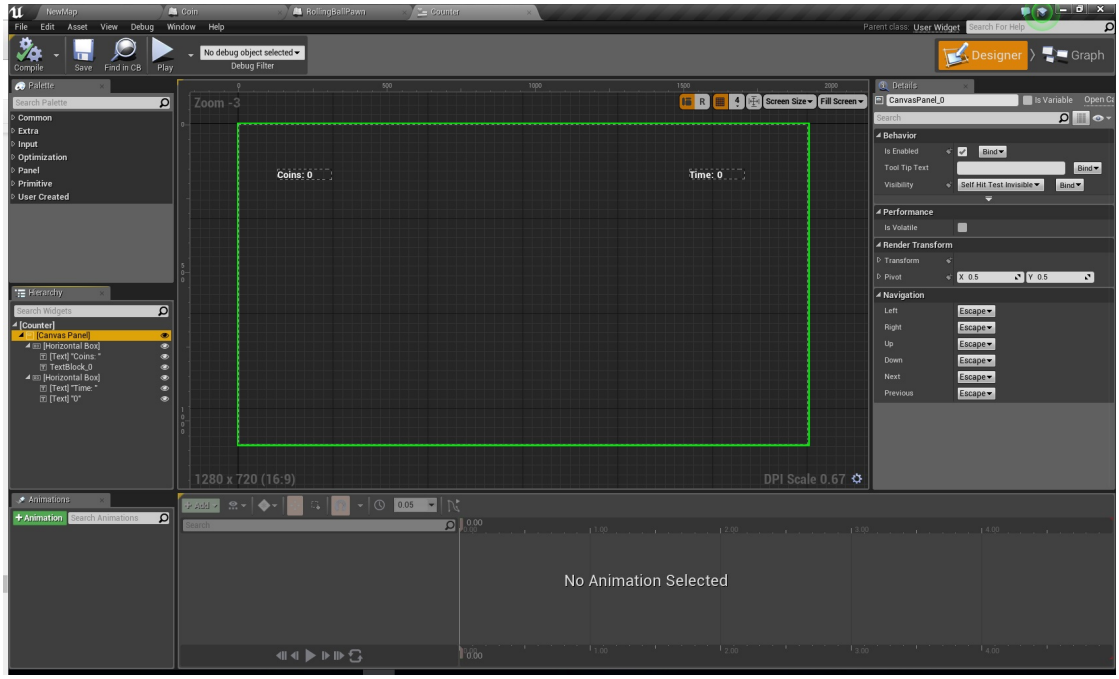

Abbildung 4.47: (Unreal) Neues Canvas

4.7.2 Unreal Engine 4

Wir beginnen damit im Content Browser ein neues Widget Blueprint zu erstellen. Wir nennen es Counter. Nach einem Doppelklick auf den Blueprint gelangen wir wie üblich in den Editor. Hier fügen wir dem Canvas Panel in der Hierarchie zwei horizontale Boxen hinzu. Jede Box bekommt anschließend noch zwei Text-Elemente. Die erste horizontale Box wird über die Anchors Funktion des Details Panels links oben verankert und die Box wird in die Nähe der linken oberen Ecke verschoben. Dem ersten Textfeld der ersten horizontalen Box geben wir den Text "Coins: " und dem zweiten Textfeld den Text "0". Bei der zweiten horizontalen Box gehen wir analog vor, nur dass wir den Anker, sowie die Box rechts Oben anordnen, und dem ersten Textfeld den Text "Time" geben (Abbildung 4.47). Wir brauchen auch noch einen "Game Over" Schriftzug, wenn dem Spieler die Zeit ausgeht. Diesen werden wir, wie auch schon in Unity, ans Hauptmenü anhängen. Nach einem Doppelklick auf den Main_Menu Blueprint gelangen wir in den Editor. Auch hier fügen wir dem Canvas Panel zwei horizontale Boxen hinzu. Um sie später leichter ansprechen zu können bekommt die Erste den Namen "GameOverBox" und die Zweite den Namen "CounterBox". Beide Boxen sollen als Variablen dienen, dazu muss das passende Häkchen im Details Panel gesetzt werden. Weiters sollen beide Boxen standardmäßig "Hidden" sein. Dazu wählt man im Details Panel bei beiden Boxen die passende Visibility aus. Weiters verankern wir beide Boxen mittig-oben, und verschieben die Boxen auch so, dass sie etwas über den Buttons liegen. Der "GameOverBox" wird ein Text-Element hinzugefügt, das den Text "GameOver!" bekommt und der "CounterBox" werden zwei

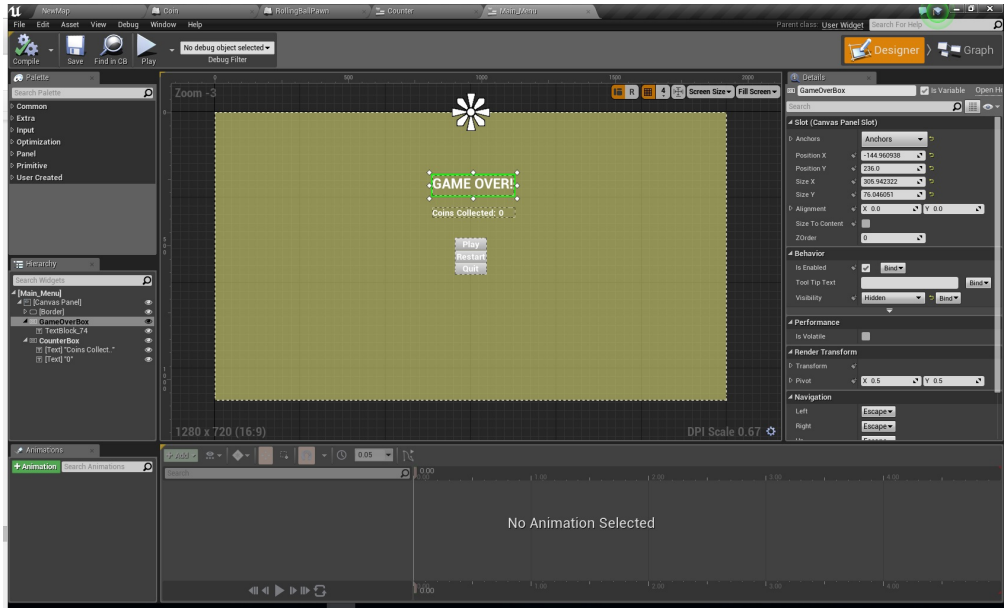

Abbildung 4.48: (Unreal) UI-Anzeige

Text-Elemente hinzugefügt. Das erste bekommt den Text "Coins Collected: " und das zweite den Text "0" (Abbildung 4.48). Nun müssen wir auch in unserem RollingBallPawn die KollisionsMechanik erstellen. Dazu öffnen wir den entsprechenden Blueprint und fügen als erstes bei den Variablen zwei Integer Variablen und eine TimerHandle Variable hinzu. Die Erste Int nennen wir "CoinCounter" und die Zweite nennen wir "CoinTimer", der TimerHandle heißt einfach TimerHandle. Nach einem Klick auf Compile oben in der Menüleiste können wir den Variablen auch Standartwerte geben, der CoinCounter bekommt den Wert Null und der Timer den Wert Zehn. Als nächstes erstellen wir einen "ActorBeginOverlap"-Knoten, von dem aus wir einen "Branch"-Knoten erstellen, der als Condition eine Überprüfung enthält, ob das überlappende Objekt eine Coin ist. Vom Branch-Knoten aus erstellen wir einen "Set Coin Timer"-Knoten, dem wir bei jedem Aufruf einen Wert von Fünf dazu addieren. Von dem Set-Knoten ausgehend erstellen wir einen IncrementInt-Knoten der als Parameter den CoinCounter bekommt (Abbildung 4.49).

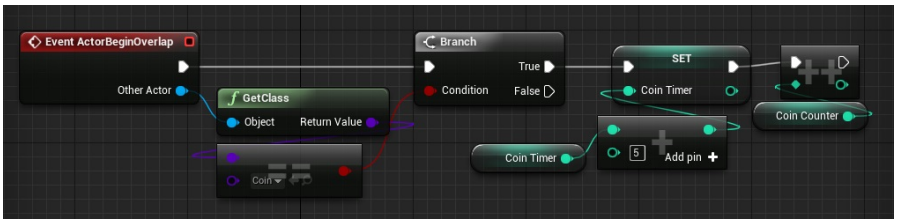

Abbildung 4.49: (Unreal) Punkte Logik 1

Weiters brauchen wir noch einen Timer, der das Spiel beendet, wenn dem Spieler die Zeit ausgeht. Dazu erstellen wir uns eine Funktion, die wir "TimerEvent" nennen. In der TimerEvent Funktion fügen wir einen `"DecrementInt"`-Knoten ausgehend vom `"TimerEvent"`-Knoten hinzu. Der `"DecrementInt"`-Knoten bekommt als Parameter den "CoinTimer". Als nächstes folgt eine Überprüfung, ob der "CoinTimer" bei Null angelangt ist. Falls die Überprüfung "true" zurückliefert, soll das Main_Menu Widget erstellt werden, wobei die "CounterBox", die "GameOverBox" und der "RestartBtn" auf Visible gesetzt wird und der "PlayBtn" auf Invisible gesetzt werden soll. Danach fügen wir das Widget dem Viewport hinzu, setzen das Spiel auf Paused und Clearen den Timer Handle (Abbildung 4.50).

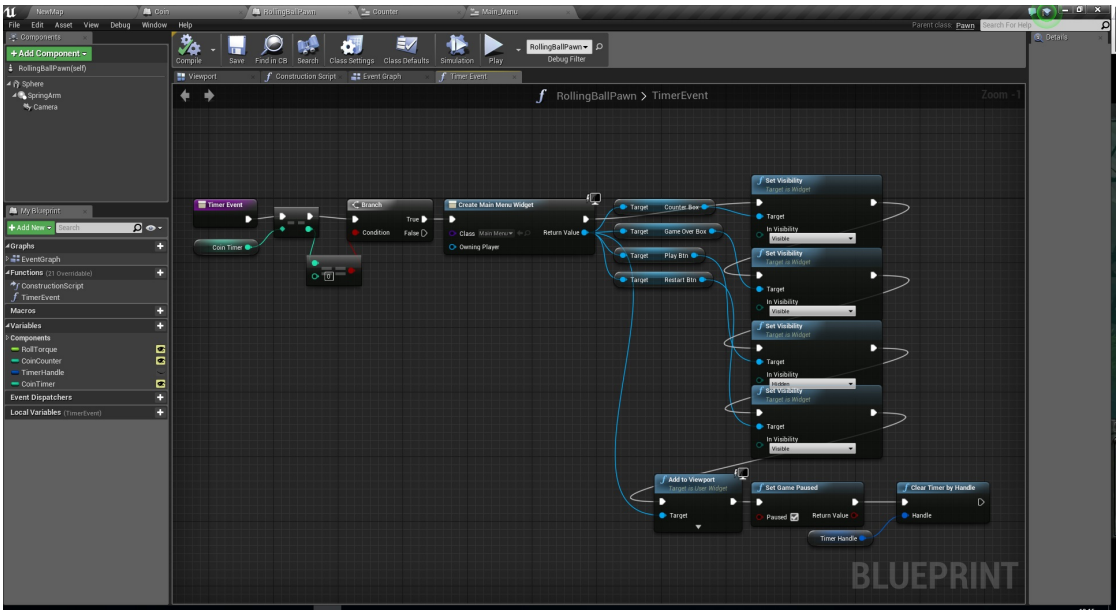

Abbildung 4.50: (Unreal) Punkte Logik 2

Damit wir dieses TimerEvent sinnvoll einsetzen können brauchen wir auch noch einen Timer, der bei Spielbeginn startet. Dazu erstellen wir im Event Graph "BeginPlay Event", von dem aus wir einen `"Set Timer by Function Name"`-Knoten erstellen. Diesem Knoten geben wir als Function Name "TimerEvent" an, die Time setzen wir auf Eins und wir setzen das Häkchen bei Looping. Den Return Value dieses Knotens übergeben wir dem TimerHandle (Abbildung 4.51). Nun kann man das ganze über den Playbutton ausprobieren.

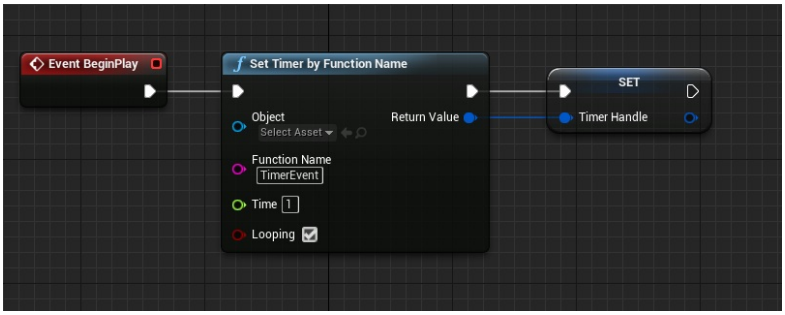

Abbildung 4.51: (Unreal) Punkte Logik 3

4.7.3 Fazit

Dieser Abschnitt ähnelt dem des Erstellens vom User Interface. Das Verhalten des Timers und Punktezählers sowie der "Game Over" Anzeige wird in Unity über C#-Scripts geregelt und in Unreal mit Visual Scripting. Für das HUD wird auf beiden Seiten ein Canvas verwendet, auf dem die UI-Elemente abgebildet werden.

4.8 Zusammengefasst

Das Erstellen und Hinzufügen neuer Objekte in der Szene wird in beiden Engines gleich gehandhabt. Dies betrifft auch UI-Elemente mit der Ausnahme, dass Unity die Vorschau-Anzeige in der Szene abbildet während Unreal eines separates Editor-Fenster besitzt. Des Weiteren liegt der wohl größte Unterschied im Programmieren der Game-Logik. In Unity wird diese mit C#-Scripts (oder auch JavaScripts) gelöst, welche als Komponenten zu gewissen Objekten hinzugefügt werden. Dabei ist es wichtig zu wissen, dass alle Skripts, mit Ausnahme von statischen Methoden und Variablen, nur zum Einsatz kommen, wenn sie in der Szene vorhanden oder als Objekt-Komponenten existieren. In Unreal hat man bei jedem Blueprint Zugriff auf den Abschnitt "EventGraph", wo das Visual Scripting zum Einsatz kommt.

CHAPTER 5

Effekte

Im diesem Abschnitt werden wir einen Blick auf die Effekte aus der Liste der Lehrveranstaltung "Computergraphik Übung" werfen und beschreiben, ob und wie diese Effekte in der jeweiligen Engine vertreten sind. Die Liste kann man unter folgendem Link finden: `https://lva.cg.tuwien.ac.at/cgue/wiki/doku.php?id=students:effect_list`

5.1 Lighting / Shading

5.1.1 Light Mapping

Unity 5	Unreal Engine 4
Unity verfügt über ein integriertes Tool "Beast" welches das Lightmapping übernimmt.	Das Erstellen und Baken der Lightmaps kann direkt von der Unreal Engine übernommen werden, wenn Lightmaps nicht schon über ein externes Tool in das Mesh gebaked wurden.

5.1.2 Deferred Shading

Unity 5	Unreal Engine 4
Deferred Shading wird von Unity unterstützt und kann als Rendering Path ausgewählt werden.	Deferred Shading ist der Standard Rendering Path der Unreal Engine 4.

5.1.3 Shadow Maps

Unity 5	Unreal Engine 4
Unity stellt mehrere Licht-Objekte zur Verfügung, welche Schatten werfen können. Darunter befinden sich sowohl eindirektionale und auch omni-direktionale Shadow-Mappings. Zusätzlich gibt es Einstellungen, mit denen sich die Art und Qualität der Schatten anpassen lässt.	Auch die Unreal Engine bietet mehrere Licht-Objekte, die Schatten werfen können. Darunter befinden sich sowohl eindirektionale und auch omni-direktionale Shadow-Mappings. Zusätzlich gibt es Einstellungen mit denen sich die Art und Qualität der Schatten anpassen lässt.

5.1.4 Shadow Volumes

Unity 5	Unreal Engine 4
Standardmäßig nicht vorhanden. Unity Versionen vor 5.0 hatten Toolkits im Assetstore für Shadow-Volumes.	Standardmäßig nicht vorhanden. Muss in C++ selber umgesetzt werden, da über den Engine Code auf den stencil Buffer zugegriffen werden kann.

5.1.5 Spotlights

Unity 5	Unreal Engine 4
Vorhanden.	Vorhanden.

5.1.6 Projected Textures

Unity 5	Unreal Engine 4
Unter den Komponenten gibt es einen Projector zur Auswahl, mit welchem man ein Matrial auf eine oder mehrere Oberflächen projizieren kann.	Wird in der Unreal Engine Decal genannt. Decals können auf sämtliche Meshes projiziert werden.

5.2 Miscellaneous

5.2.1 Particle System

Unity 5	Unreal Engine 4
Kann als Effekt-Komponente zu einem Objekt hinzugefügt werden und bietet eine Vielzahl an Einstellungsmöglichkeiten.	Über das Cascade Particle System lassen sich CPU/GPU Particle Emitter verwenden.

5.2.2 Water

Unity 5	Unreal Engine 4
Die einfachste Variante wäre es, eines der Standard-Assets für Wasser zu verwenden. Alternativ kann man mit einem horizontal orientierten Mesh, einem Script und dem passenden Material ein eigenes Objekt zur Repräsentation einer Wasseroberfläche erstellen. Beim Material lässt sich oben der Shader auswählen. In dem Fall sollte man einen, der unter "FX/" zur Verfügung gestellten Water-Shadern, verwenden.	Über den Material Editor umsetzbar. Kann mit mit Hilfe von passende Normal Maps und Panner Nodes animiert werden. Allerdings wäre die einfachste Variante, sich einen fertigen Water Blueprint aus den Examples zu verwenden.

5.2.3 Water Reflection

Unity 5	Unreal Engine 4
Lässt sich in allen vier Water-Shadern einstellen.	Das Reflection Environment und Screen Space Reflection sind dafür zuständig, Reflexionen darzustellen.

5.2.4 Water Refraction

Unity 5	Unreal Engine 4
Lässt sich im einem Water-Shader einstellen.	Über den Material Editor umsetzbar.

5.2.5 Water Mesh

Unity 5	Unreal Engine 4
Muss eigenständig implementiert werden. Es gibt dafür keine vorgefertigte Option.	Über den Material Editor umsetzbar.

5.2.6 GPU Vertex-Skinning

Unity 5	Unreal Engine 4
Diese Funktion wird unterstützt, da man die Möglichkeit hat Skeletal-Meshes zu importieren.	Diese Funktion wird unterstützt, da man die Möglichkeit hat Skeletal-Meshes zu importieren.

5.3 Surface Effects

5.3.1 Environment Mapping

Unity 5	Unreal Engine 4
Der Textur-Typ "Cubemap" kümmert sich um das Environment Mapping.	Das Reflection Environment kümmert sich um das Environment Mapping.

5.3.2 Normal Mapping

Unity 5	Unreal Engine 4
Der Textur-Typ "Normalmap" kümmert sich um das Normal-Mapping.	Über den Material Editor kann Normal Mapping verwendet werden.

5.3.3 Parallax Occlusion Mapping

Unity 5	Unreal Engine 4
In einem Material, welches den Standard Shader verwendet, lässt sich eine Heightmap einbauen, welche das Parallax Mapping anwendet.	Über den Material Editor kann Parallax Occlusion Mapping verwendet werden.

5.3.4 Displacement Mapping (Surface Tessellation)

Unity 5	Unreal Engine 4
Für diese Funktion muss ein eigener Tessellation Shader geschrieben werden.	Über den Material Editor kann Displacement Mapping verwendet werden.

5.3.5 Animated Textures (Video Stream)

Unity 5	Unreal Engine 4
Wenn man ein Video in Untiy importiert, kann man dieses als MovieTexture einsetzen. Dazu muss allerdings QuickTime installiert werden.	Über den Material Editor und das Media Framework können Video Texturen umgesetzt werden. Für einfache animierte Texturen reicht der Material Editor.

5.4 Visibility / Level of Detail

5.4.1 Static/Tessellation LOD

Unity 5	Unreal Engine 4
Static LOD Lässt sich mit der LOD-Group Komponente einstellen. Für Tessellation muss der entsprechende Shader (aus dem Displacement Mapping Abschnitt) angepasst werden.	Diverse Einstellungen für statisches LOD als auch für Tessellation LOD.

5.4.2 Protal Rendering

Unity 5	Unreal Engine 4
Muss eigenständig mit Scripts umgesetzt werden.	Muss selber über Blueprints bzw. Code umgesetzt werden.

5.5 Screen Space Effects

5.5.1 Lens Flare

Unity 5	Unreal Engine 4
Kann als Effekt-Komponente zu einen Objekt hinzugefügt werden.	Kann in Post Process Volumes eingestellt werden.

5.5.2 Bloom

Unity 5	Unreal Engine 4
Ein Script ist in den Standard-Assets enthalten. Dieses kann als Komponente zu einer Camera hinzufügt werden.	Kann in Post Process Volumes eingestellt werden.

5.5.3 Motion Blur

Unity 5	Unreal Engine 4
Ein Script ist in den Standard-Assets enthalten. Dieses kann als Komponente zu einer Camera hinzufügt werden.	Kann in Post Process Volumes eingestellt werden.

5.5.4 Depth of Field

Unity 5	Unreal Engine 4
Ein Script ist in den Standard-Assets enthalten. Dieses kann als Komponente zu einer Kamera hinzufügt werden.	Kann in Post Process Volumes eingestellt werden.

5.5.5 Lightshafts

Unity 5	Unreal Engine 4
Ein Script ist in den Standard-Assets enthalten. Dieses kann als Komponente zu einer Kamera hinzufügt werden.	Kann in Post Process Volumes eingestellt werden.

5.5.6 HDR

Unity 5	Unreal Engine 4
HDR ist ein Attribut eines Kamera-Objects und kann ein- oder ausgeschalten werden.	Kann in Post Process Volumes eingestellt werden.

5.5.7 Screen Space Ambient Occlusion (SSAO)

Unity 5	Unreal Engine 4
Ein Script ist in den Standard-Assets enthalten. Dieses kann als Komponente zu einer Kamera hinzufügt werden.	Kann in Post Process Volumes eingestellt werden.

5.5.8 Horizon-based Ambient Occlusion (HBAO)

Unity 5	Unreal Engine 4
Diese Funktion ist standardmäßig nicht vorhanden.	Diese Funktion ist standardmäßig nicht vorhanden.

5.5.9 Cel shading

Unity 5	Unreal Engine 4
In den Standard-Assets befindet sich ein Toon-Shader.	Man kann Cel Shading über Post Process Materials erreichen, aber für schönes Cel Shading muss man sich Materials und Blueprints erstellen.

5.5.10 Contours

Unity 5	Unreal Engine 4
Ein Script ist in den Standard-Assets enthalten. Dieses kann als Komponente zu einer Kamera hinzufügt werden.	Kann in Post Process Volumes eingestellt werden.

CHAPTER 6

Rechtliches

6.1 Unity 5

Unity bietet zwei verschiedene Versionen ihres Produktes an. Die kostenpflichtige Unity Pro und die kostenlose Unity Personal. In der Hinsicht gibt es bei der Lizenzvereinbarung einige Punkte, die besonders beachtet werden sollten. Unity Personal darf nur unter der Bedingung, dass das Einkommen bzw. Budget des vorherigen Jahres den Betrag von 100.000 $ nicht überschreitet, verwendet werden. Andernfalls ist man dazu verpflichtet, Unity Pro zu kaufen. Inhalte, die mit Unity Personal erstellt wurden, dürfen nicht mit von Unity Pro erzeugten Inhalten in irgendeiner Art und Weise kombiniert bzw. vereint werden. Man kann jedoch mit der Personal Version anfangen, auf Pro upgraden und die Arbeit anschließend fortsetzen. Man darf Inhalte der Unity Personal Add-on für ein Betriebssystem nicht mit Inhalten der Unity Pro Add-on für dasselbe Betriebssystem kombinieren. Unity Free darf nur auf einem Rechner installiert werden, während man bei der Pro Version einen sekundären Rechner auswählen kann. Trotzdem ist es nicht erlaubt, mehrere Kopien gleichzeitig laufen zu haben. Um erstellte Inhalte öffentlich über Stream, Broadcast zu übertagen, wird eine zusätzliche Lizenz von Unity benötigt. Ebenso dürfen keine Glückspiel-Inhalte ohne einer separaten Lizenz enthalten sein. Falls die erstellte Software auf elektronischen Geräten installiert oder darin eingebettet ist und die Hauptfunktionen bzw. das Userinterfaces des Geräts beinhaltet, darf die Anzahl dieser Geräte höchstens 1000 Stück betragen. Andernfalls braucht man eine zusätzliche Lizenz von Unity.[uni16]

6.2 Unreal Engine 4

Im Allgemeinen verlangt Epic Games 5% Lizenzgebühr von jedem Produktverkauf. Davon ausgenommen sind die ersten 3000 $ für jedes Produkt pro Kalenderquartal. Für den edukativen Gebrauch der Unreal Engine gelten einige Sonderregelungen, die

die Verwendung für akademische Institute vereinfachen. Im Normalfall gilt eine Lizenz für einen einzelnen User, der die Engine auf allen seinen Rechnern installieren, die Technologie jedoch nicht mit anderen Leuten teilen darf. Wenn man eine akademische Institution ist, ist es jedoch erlaubt, die Engine auf allen Computern des Instituts zu installieren und jeder User dieser Computer hat die Erlaubnis, die Engine zu benutzen, natürlich unter Einhaltung der Lizenz. Weiters ist auch das Streamen von Unreal Engine 4 Videos ausdrücklich erlaubt. Mit der Lizenz bekommt ein User auch kompletten Zugriff auf den C++ Engine Code, der viele interessante Code Segmente beinhaltet. Dabei muss man bedenken, dass nicht mehr als 30 zusammenhängende Zeilen Code online veröffentlicht werden dürfen. Solange Projekte kein Geld bzw. weniger als 3000 $ pro Kalenderquartal einbringen, können sie ohne weiteres veröffentlicht werden. Daher steht einer Veröffentlichung im Rahmen einer universitätsinternen Hall of Fame nichts im Wege.[unr16]

6.3 Fazit

Bei der Unreal Engine gibt es eine einzige Lizenz-Variante, welche dem Benutzer einen uneingeschränkten kostenlosen Zugang zu allen Features verschafft. Eine ähnliche kostenfreie Lizenz wird auch von Unity angeboten, bei der allerdings einige Quality-of-Life Funktionen fehlen, welche nur in der kostenpflichtigen Pro Version enthalten sind. Ein weiterer wichtiger Punkt ist, dass man bei der Unreal Engine auf den Source-Code zugreifen kann, was bei Unity weder in der Personal noch in der Pro Version enthalten ist. Dieser Zugang muss separat erworben werden, indem das zuständige Unity-Personal in dem jeweiligen Land kontaktiert wird. Ein Vergleich der Angebote beider Engines lässt ohne Weiteres den Schluss zu, dass die UE4 auch für Lehrzwecke kostentechnisch eher geeignet ist. Vorausgesetzt man möchte den vollen Funktionsumfang nutzen, ist die Unreal Engine die bessere Wahl, da während der Entwicklung des Projektes keine Kosten anfallen, sondern erst nachdem das Produkt Geld einnimmt. Bei der Verwendung von Unity Pro fallen fixe Monatsgebühren von 75 $ an. Dies erweist sich jedoch bei größeren und erfolgreichen Projekten als vorteilhaft, da die Kosten im Gegensatz zu Unreal Engine Projekten nicht abhängig von den Einnahmen ansteigen.

CHAPTER 7

Zusammenfassung

Die Wahl der passenden Engine für ein Projekt hängt von mehreren Faktoren ab. Auf der einen Seite haben wir die visuelle Qualität, welche vor allem bei AAA Titeln eine große Rolle spielt. Wenn man Ambitionen hat, ein ähnliches grafisches Niveau zu erreichen, sollte man zur Unreal Engine 4 greifen. Diese bringt zusätzlich zu ihrer State-of-the-Art Technologie auch noch den Vorteil einer aktiven Community, welche die Engine ständig weiterentwickelt. Auf der anderen Seite können Einsteiger schnell von der Feature Vielfalt überfordert werden, während die Unity Engine durch ihren einfachen Aufbau anfängerfreundlicher wirkt.

Ein weiterer Faktor ist das Scripten innerhalb der Engines, welches bei Unity dem gewohnten Ablauf folgt. Während es in Unreal auch möglich ist in C++ zu programmieren, wird mit dem Visual Scripting ein innovativer Weg eingeschlagen, der unseres Erachtens ein Killer Feature ist.

Bei der Content-Pipeline haben wir festgestellt, dass die Unity Engine weitaus mehrere Formate beim Import von Audio und 3D-Assets unterstützt, wobei man auf einige durchaus verzichten könnte, da sie keinen qualitativen Vorteil bringen.

Der letzte wichtige Punkt sind die Kosten. Bei finanziell erfolgreichen Projekten müssen auf beiden Seiten Abgaben geleistet werden, wobei sich diese bei Unity auf die Abo-Gebühren der Pro Version beschränken und bei Unreal proportional zum Erfolg des Projektes ansteigen. Daher muss auf Projekt zu Projekt Basis entschieden werden, welche Engine die wenigsten Kosten verursacht und ob man dafür Qualitätseinbußen hinnehmen will.

Abschließend können wir sagen, dass uns die Unreal Engine mehr beeindruckt hat und das Umsetzen des Beispielprojektes nach der Einarbeitungszeit intuitiver und angenehmer von statten ging.

Bibliography

[uni] Unity 5 Documentation. http://docs.unity3d.com/Manual/.

[uni16] Unity Pro and Unity Personal Software License Agreement 5.x. https://unity3d.com/legal/eula, 2016.

[unr] Unreal Engine 4 Documentation. https://docs.unrealengine.com/latest/INT/index.html.

[unr16] Unreal® Engine End User License Agreement . https://www.unrealengine.com/eula, 2016.