Diplomarbeit

Höhere Technische Bundeslehranstalt Leonding
Abteilung für Informatik

Ludimus - eine mobile Spieleplattform

Eingereicht von: **David Matousch, 5AHIF**
Eric Stock, 5AHIF
Datum: **April 9, 2019**
Betreuer: **Rupert Obermüller**

Declaration of Academic Honesty

Hereby, I declare that I have composed the presented paper independently on my own and without any other resources than the ones indicated. All thoughts taken directly or indirectly from external sources are properly denoted as such.

This paper has neither been previously submitted to another authority nor has it been published yet.

Leonding, April 9, 2019

David Matousch, Eric Stock

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorgelegte Diplomarbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Gedanken, die aus fremden Quellen direkt oder indirekt übernommen wurden, sind als solche gekennzeichnet.

Die Arbeit wurde bisher in gleicher oder ähnlicher Weise keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Leonding, am 9. April 2019

David Matousch, Eric Stock

Abstract

Ludimus is a mobile gaming platform for all sorts of card, board and arcade games designed for the modern family to play in their holidays. The goal is to provide an alternative to playing games with your family at home and to take all your favourite games with you on your holidays. The latter heavily influenced our decisions on how we build our platform. Ludimus is not based on web technologies and remote servers because a good internet connection is rare in these occasions. When users have an internet connection they have the ability to download new games from our Server. We used Firebase for this, because we didn't need any backend logic just storage and a database and that's exactly what Firebase delivers on. We developed Ludimus with the game engine Unity, which provided the next to perfect cross platform capabilities we needed for our Windows, Android and iOS clients. We wanted to make playing in remote areas as easy as possible without ditching constructs that traditional games established. We think the reason why families play these games is because they involve local social activities, which get partially lost when playing over the internet and there is no central point of sight which players share. We tried to keep these basic principles but incorporated technology into the games as well. To accomplish this we decided to display the board, or the table in general, on a large screen, like a TV or a tablet and the cards, resources, controls or money of each player is displayed on their phone. We wanted to minimize the eye contact needed to control the phone, without abandoning it completely. Some people like the physical activity when rolling the dice or moving the figures on the board. That's why we developed systems to simulate these gestures. Although we attended many public events to show and promote Ludimus and took part in multiple competitions to win prices, we finally decided to cancel the project.

Zusammenfassung

Ludimus ist eine mobile Spieleplattform für Karten-, Brett- und Arcadespielen, die für die moderne Familie in den Ferien konzipiert wurde. Das Ziel ist eine Alternative zum Spielen mit der Familie zu Hause anzubieten und zu ermöglichen alle Lieblingsspiele in den Urlaub mitzunehmen. Letzteres hat unsere Entscheidungen über den Aufbau unserer Plattform stark beeinflusst. Ludimus basiert nicht auf Webtechnologien und Remote-Servern, da im Urlaub eine gute Internetverbindung eine Seltenheit ist. Wenn Benutzer jedoch über eine Internetverbindung verfügen, können sie neue Spiele von unserem Server herunterladen. Hierfür haben wir Firebase verwendet, weil wir keine Backend-Logik brauchten, nur einen Dateispeicherort und eine Datenbank, und genau das ist, was Firebase liefert. Wir haben Ludimus mit der Game-Engine Unity entwickelt, da diese perfekte plattformübergreifende Funktionen bietet, die wir für unsere Windows-, Android- und iOS-Clients benötigten. Wir wollten das Spielen in abgelegenen Gegenden so einfach wie möglich machen, ohne Konstrukte zu verwerfen, die traditionelle Spiele etabliert haben. Wir denken, der Grund, warum Familien diese Spiele spielen, liegt darin, dass sie lokale soziale Aktivitäten beinhalten, die beim Spielen über das Internet teilweise verloren gehen und es keinen zentralen Punkt gibt, den die Spieler teilen. Wir haben versucht, diese Grundprinzipien beizubehalten, bauten jedoch auch Technologie in die Spiele ein. Um dies zu erreichen, haben wir uns entschieden, das Brett oder den Tisch im Allgemeinen auf einem großen Bildschirm wie einem Fernseher oder einem Tablet anzuzeigen, und die Karten, Ressourcen, Steuerelemente oder das Geld jedes Spielers werden auf dem Handy angezeigt. Wir wollten den Augenkontakt minimieren, der zur Steuerung des Smartphones erforderlich ist, ohne es vollständig unbrauchbar zu machen. Manche Leute mögen die körperliche Aktivität beim Würfeln oder beim Bewegen der Figuren auf dem Brett. Deshalb haben wir Systeme entwickelt, um diese Gesten zu simulieren. Obwohl wir bei vielen öffentlichen Veranstaltungen vertreten waren, um Ludimus zu zeigen und zu promoten, und an mehreren Wettbewerben teilgenommen haben, um Preise zu gewinnen, haben wir uns schließlich entschlossen, das Projekt abzubrechen.

Danksagung

Wir möchten uns bei der Htl Leonding und im speziellen bei Dipl.-Ing. Peter Bauer und Dipl.-Ing. Wolfgang Holzer für ihre Vision und ihr Engagement für die "Think Your Product"-Initiative bedanken. Bei unserem Betreuungslehrer Dipl.-Ing. Rupert Obermüller, der uns auf unserem Weg begleitet hat, möchten wir uns selbstverständlich auch bedanken, ebenso wie bei den Mitgliedern der Factory 300 und des Edison Preises für ihr Feedback und ihre Unterstützung.

Inhaltsverzeichnis

1 Einleitung **7**
- 1.1 Ausgangssituation . . . 7
- 1.2 Ziele . . . 7
- 1.3 Marke . . . 8
- 1.4 Ähnliche Arbeiten und Projekte . . . 8
- 1.5 Struktur der Arbeit . . . 8
  - 1.5.1 Verwendete Technologien . . . 8
  - 1.5.2 How to use . . . 9
  - 1.5.3 Ausgewählte Aspekte der Systemerstellung . . . 9
  - 1.5.4 Spiele . . . 9
  - 1.5.5 Ars Electronica Festival . . . 9

2 Zusammenfassung **10**

3 Verwendete Technologien **11**
- 3.1 Unity (DM) . . . 12
  - 3.1.1 Unity Multiplayer . . . 12
  - 3.1.2 Remote Settings . . . 12
  - 3.1.3 Unity Asset Store . . . 13
  - 3.1.4 Unity Collaborate . . . 13
  - 3.1.5 Unity Preispolitik . . . 14
  - 3.1.6 Plattformen . . . 14
  - 3.1.7 Mobile Entwicklung . . . 15
  - 3.1.8 Community . . . 15
  - 3.1.9 Integration . . . 15
    - 3.1.9.1 Unity und Blender . . . 15
    - 3.1.9.2 Unity und Photoshop . . . 15
- 3.2 Entwicklung mit Unity (DM) . . . 16
  - 3.2.1 Editor . . . 16
    - 3.2.1.1 Hierarchy . . . 16
    - 3.2.1.2 Scene . . . 17
    - 3.2.1.3 Project . . . 17
    - 3.2.1.4 Console . . . 17

3.2.1.5 Animation . . . 17
3.2.1.6 Game . . . 17
3.2.2 Komponenten . . . 17
3.2.2.1 Mesh . . . 18
3.2.2.2 Effects . . . 18
3.2.2.3 Physics . . . 18
3.2.2.3.1 2D Physics . . . 18
3.2.2.3.2 3D Physics . . . 19
3.2.2.4 Navigation . . . 19
3.2.2.5 Audio . . . 19
3.2.2.6 Video . . . 19
3.2.2.7 Rendering . . . 19
3.2.2.7.1 Camera . . . 19
3.2.2.7.2 Skybox . . . 20
3.2.2.7.3 Light . . . 20
3.2.2.7.4 Sprite Renderer . . . 20
3.2.2.8 UI . . . 20
3.2.2.9 AR . . . 20
3.2.3 Scripts . . . 21
3.2.3.1 Lifecycle Methoden . . . 21
3.2.3.1.1 Awake . . . 21
3.2.3.1.2 OnEnable . . . 21
3.2.3.1.3 Start . . . 21
3.2.3.1.4 FixedUpdate . . . 21
3.2.3.1.5 Update . . . 22
3.2.3.1.6 LateUpdate . . . 22
3.2.3.1.7 OnDisable . . . 22
3.2.3.1.8 OnDestroy . . . 22
3.2.3.2 Kollisionen . . . 22
3.2.3.2.1 OnCollisionEnter . . . 22
3.2.3.2.2 OnCollisionEnter2D . . . 22
3.2.3.2.3 OnCollisionStay/OnCollisionStay2D . . . 22
3.2.3.2.4 OnCollisionExit/OnCollisionExt2D . . . 22
3.2.3.3 Trigger . . . 22
3.2.3.4 Objekte erzeugen . . . 23
3.2.3.4.1 Resources . . . 23
3.2.3.4.2 Klonen . . . 23
3.2.3.4.3 Referenzen . . . 23
3.2.3.5 Öffentliche Variable und Verlinkung zwischen Komponenten . . . 23
3.2.4 Alternativen . . . 24
3.3 Firebase (DM) . . . 26
3.3.1 Datenbank . . . 26

3.3.1.1 Cloud Firestore . . . 26
3.3.1.2 Realtime Database . . . 27
3.3.2 Storage . . . 27
3.3.3 Hosting . . . 27
3.3.4 Authentifizierung . . . 28
3.3.5 Preispolitik . . . 28
3.4 Netzwerkverbindungen (ES) . . . 29
3.4.1 TCP . . . 29
3.4.1.1 Definition . . . 29
3.4.1.2 Verifizierung . . . 29
3.4.2 TCP Client . . . 30
3.4.2.1 Übersicht . . . 30
3.4.2.2 Konstruktoren . . . 30
3.4.2.3 Methoden . . . 31
3.4.2.4 Verwendung in Ludimus . . . 32
3.4.3 TCP Listener . . . 32
3.4.3.1 Übersicht . . . 32
3.4.3.2 Konstruktoren . . . 32
3.4.3.3 Methoden . . . 33
3.4.3.4 Implementierung in Ludimus . . . 33
3.4.4 Network Streams . . . 34
3.4.4.1 Übersicht . . . 34
3.4.4.2 Methoden . . . 34
3.5 Sonstige Technologien . . . 36
3.5.1 Figma (DM) . . . 36
3.5.2 Blender . . . 36
3.5.3 Vectr (DM) . . . 36
3.5.4 Illustrator (DM) . . . 36
3.5.5 Delegates (ES) . . . 36
3.5.5.1 Übersicht . . . 36
3.5.5.2 Anwendung . . . 36
3.5.5.3 Implementierung in Ludimus . . . 37
3.5.6 ZXing.Net (ES) . . . 37
3.5.6.1 Übersicht . . . 37
3.5.6.2 Einbindung . . . 37
3.5.6.3 Warum ZXing.Net . . . 38
3.5.6.4 Usage . . . 38
3.5.6.4.1 Erzeugung . . . 38
3.5.6.4.2 Entschlüsselung . . . 38
3.5.6.5 Implementierung in Ludimus . . . 38

4 How to use (ES) 40
4.1 Vor dem Start . . . 41
4.2 Computer-App . . . 41
4.3 Android/IOS - App . . . 41
4.3.1 Vor dem Einloggen . . . 41
4.3.2 Nach dem Einloggen . . . 41

5 Ausgewählte Aspekte der Systemerstellung 43
5.1 Designentscheidungen (DM) . . . 44
5.1.1 Generelle Designentscheidungen . . . 44
5.1.2 Spezielle Designentscheidungen . . . 48
5.1.2.1 Shop . . . 48
5.2 Spielstart (DM) . . . 50
5.2.1 Probleme . . . 50
5.2.2 Optimierungsansätze . . . 52
5.3 Debug Modus (DM) . . . 53
5.4 Aufbau von Ludimus (ES) . . . 54
5.4.1 Übersicht . . . 54
5.4.2 Ablaufdiagram Ludimus . . . 55
5.4.2.1 Erklärung des Ablaufdiagrams(5.10) . . . 55
5.4.2.1.1 Schritt 1 . . . 55
5.4.2.1.2 Schritt 2 . . . 55
5.4.2.1.3 Schritt 3 . . . 55
5.4.2.1.4 Schritt 4 . . . 57
5.4.2.1.5 Schritt 5 . . . 57
5.5 Lobbycodes (ES) . . . 58
5.5.1 Verbindungsgeschichte . . . 58
5.5.2 Erzeugung des Lobbycodes . . . 58
5.6 Verbindung (ES) . . . 59
5.7 Aufbau der Verbindung (ES) . . . 60
5.7.1 PC/Laptop/Tablet . . . 60
5.7.2 Adroid/IOS . . . 60
5.8 Für Verbindung wichtige Klassen und Methoden / Übergang Schnittstellen (ES) . . . 61
5.8.1 PlayerManager . . . 61
5.8.1.1 Übersicht . . . 61
5.8.1.2 Wichtige Fields/Properties . . . 61
5.8.1.3 Wichtige Methoden . . . 61
5.8.2 ClientController . . . 62
5.8.2.1 Übersicht . . . 62
5.8.2.2 Wichtige Fields/Properties . . . 62
5.8.2.3 Wichtige Methoden . . . 62

6 Spiele (ES) 63
6.1 Spieleentwicklung für Ludimus . . . 64
6.1.1 Grundgedanke . . . 64
6.1.2 Motivation . . . 64
6.1.3 Wie programmiere ich Spiele für Ludimus . . . 64
6.1.4 Testen . . . 65
6.2 Vorgang Spieleauswahl . . . 65
6.3 Probleme bei der Spiele Entwicklungen . . . 65
6.4 Spiele . . . 67
6.4.1 Poker . . . 67
6.4.1.1 Erklärung . . . 67
6.4.1.2 Geschichte . . . 67
6.4.1.3 Aufbau . . . 68
6.4.1.4 Schwierigkeiten . . . 68
6.4.2 Jumpy . . . 68
6.4.2.1 Erklärung . . . 68
6.4.2.2 Geschichte . . . 69
6.4.2.3 Aufbau . . . 69
6.4.2.4 Schwierigkeiten . . . 70
6.4.3 Knightslider . . . 70
6.4.3.1 Erklärung . . . 70
6.4.3.2 Geschichte . . . 70
6.4.3.3 Aufbau . . . 70
6.4.3.4 Schwierigkeiten . . . 71
6.4.4 Billard . . . 71
6.4.4.1 Erklärung . . . 71
6.4.4.2 Geschichte . . . 71
6.4.4.3 Aufbau . . . 71
6.4.4.4 Schwierigkeiten . . . 71
6.4.5 Squarerunner . . . 72
6.4.5.1 Erklärung . . . 72
6.4.5.2 Geschichte . . . 72
6.4.5.3 Aufbau . . . 72

7 Ars Electronica Festival (ES) 73
7.1 Übersicht . . . 74
7.2 Ludimus Testlauf . . . 74
7.2.1 Wie es dazu kam . . . 74
7.2.2 Idee . . . 74
7.2.3 Realität . . . 75
7.3 Ergebnis . . . 75
7.4 Probleme . . . 75

Kapitel 1

Einleitung

1.1 Ausgangssituation

Klassische Brettspiele benötigen viel Platz zum Verstauen. Platz, den eine Familie eventuell nicht einmal daheim hat, aber schon gar nicht wenn sie in die Ferien fahren. Brettspiele benötigen viel zu viel Platz im Koffer und man muss sich, wenn man schon welche mitnimmt, für ein paar wenige entscheiden. Und wenn dem Kind dann zufällig die Spiele gerade nicht passen hat man ein Problem. Bei Ludimus hat man die komplette Spielesammlung in der Cloud und kann sie immer und überall mitnehmen ohne viel Platz zu brauchen.
Auch werden Kinder immer abgeneigter gegenüber klassischen Brettspielen und wollen lieber was am Handy spielen. Mithilfe von Ludimus können Eltern das Spielen mit ihren Kindern mit dem, lieber am Handy spielen ihrer Kinder verbinden, da die Plattform komplett über das Smartphone gesteuert wird.

1.2 Ziele

Das große Ziel von Ludimus ist es, ein Produkt auf den Markt zu bringen, das die Art, wie Familien mit ihren Kindern Zeit verbringen, für immer ändert. Brettspielabende werden immer seltener, weil Kinder an ihre technischen Geräte heutzutage so sehr gewohnt sind, dass Brettspiele einfach ein zu großer Rückschritt sind. Ludimus soll einerseits diese Abende durch Technologie wieder regelmäßiger machen, andererseits aber auch Familien ermöglichen Spiele in den Urlaub mitzunehmen. Ludimus basiert auf mehreren Smartphones und einem Tablet, Laptop oder Fernseher. Mit Ausnahme des Fernsehers, sind das tragbare Geräte, die sowohl wenig Platz brauchen, als auch wenig Gewicht haben und somit selbst im Flieger mitgenommen werden können. Das Ziel ist somit im Urlaub für Familien unersetzbar zu sein um dadurch zu Hause eine Alternative gegenüber traditionellen Spielen zu sein.

Abbildung 1.1: Ludimus Logo

1.3 Marke

Ludimus war von Beginn an als Produkt für Kunden gedacht, weshalb wir viel Zeit in den Namen und das Logo steckten. Während der Prototypphase nannten wir das Projekt ConnecTable, da wir den Tisch mit Handys verbinden. Zusätzlich bedeutet das englische Wort „connectable“ aber auch verbindbar, wodurch das Prinzip hinter der Idee verdeutlicht wird. In der Aussprache würde die Schreibweise jedoch verloren gehen, weshalb wir uns für einen neuen Namen entschieden. Ludimus war das Produkt dieser neuen Überlegungen und ist Latein für „wir spielen“. Ludimus sollte eine Plattform für Familien sein und das „wir“ in den Mittelpunkt stellen. Zusätzlich klingt Ludimus melodisch und ist einfach auszusprechen. In Verbindung mit dem Namen entwarfen wir unser Logo. Es soll sowohl einen Spielwürfel, als auch eine Sprechblase darstellen, da Spiele, die Unterhaltungen und Zusammenhalt fördern, unser Ziel waren.

1.4 Ähnliche Arbeiten und Projekte

Ludimus ist nicht die einzige Spieleplattform. Neben traditionellen Spielekonsolen, wie der Xbox oder der Playstation, betreten immer mehr Konkurrenten, wie Google, Amazon, Microsoft, uvm. mit Gamestreaming Plattformen den Markt für mobiles Spielen. Gegenüber diesen Alternativen grenzen wir uns klar mit der Auswahl unserer Spiele ab, denn wir bieten Spiele, die speziell für die Familie und das spielen mit Freunden entwickelt wurden. Ein weiterer Unterscheidungspunkt ist unser möglicher Einsatz im Urlaub, da wir keinen externen Server zum Spielen verwenden, sondern ein lokales Gerät zum Server machen.

1.5 Struktur der Arbeit

1.5.1 Verwendete Technologien

Hier werden alle von Ludimus verwendeten Technologien kurz erklärt und deren Einsatz im Projekt gezeigt.

1.5.2 How to use

In wenigen Sätzen wird in diesem Kapitel beschrieben unter welchen Voraussetzungen eine Lobby zustande kommen kann, und welche Schritte der User durchführen muss um sich dorthin zu verbinden.

1.5.3 Ausgewählte Aspekte der Systemerstellung

Dieses Kapitel behandelt besonders wichtige oder besonders schwierige Aspekte während der Entwicklung von Ludimus.

1.5.4 Spiele

In diesem Abschnitt werden alle begonnenen Spiele von Ludimus aufgezählt und erklärt, warum wir nicht jedes Spiel in unsere Spielebibliothek aufnahmen.

1.5.5 Ars Electronica Festival

Von unseren Vorabüberlegungen, Vorbereitungen und Testresultaten wird in diesem Kapitel beschrieben.

Kapitel 2

Zusammenfassung

Here you give a summary of your results and experiences. You can add also some design alternatives you considered, but kicked out later. Furthermore you might have some ideas how to drive the work you accomplished in further directions.

Kapitel 3

Verwendete Technologien

3.1 Unity (DM)

Unity ist eine Entwicklungsumgebung hauptsächlich gedacht für die Spieleentwicklung. Eigentümer des Produkts ist Unity Technologies, die ihren Sitz in San Francisco haben. Das Unternehmen ist seit 2004 in Betrieb und arbeitet ausschließlich an Unity und dessen Services. Die wichtigsten Services hierbei sind Unity Multiplayer, Remote Settings, der Unity Asset Store und Unity Collaborate.

3.1.1 Unity Multiplayer

Unity Multiplayer soll es Entwicklern ermöglichen leicht und schnell, auf Mehrspieler basierende, Spiele zu entwickeln. Der Service bietet Support für die Synchronisierung von Objektpositionen zwischen den Geräten, Serverseitige Commands, die Cheating deutlich erschweren, eine Unterscheidung zwischen Client und Server im Code, sodass Entwickler auch für Einzelspieler gedachte Projekte leicht umbauen können, und vieles mehr. Der wohl größte Vorteil ist die Inkludierung eines Lobbysystems. Spieler können Lobbys erstellen und diese für andere Spieler freischalten, sodass diese der Gruppe beitreten können. Immer mehr Mehrspielerspiele, wie „Battlefield“, „Counterstrike“ oder „Destiny“, benutzen Matchmaking um Spieler, die in derselben Region spielen, in Lobbys zusammenzuschließen, um sie zu füllen. Die Spieleindustrie geht immer mehr in die Richtung, der Servicespiele. Der Grund ist, dass Spiele immer teurer werden und große Publisher nicht mehr mit den 60 Euro pro Spiel auskommen. Sie versuchen die Spiele weiter zu monetarisieren, indem sie Spielinhalte ohne zusätzliche Kosten erst spät oder gar nicht erreichbar machen. Je langlebiger die Spiele sind, desto eher ist die Wahrscheinlichkeit, dass Spieler bereit sind mehr dafür zu zahlen. Erreicht wird diese Langlebigkeit durch periodische Updates und nicht zu Letzt durch Mehrspielermodi. Spiele wie „The Division“, „Destiny“ oder „Anthem“ werden in vergleichbar schlechten Zustand veröffentlicht, mit einer großen Karte zum Erkunden und einem Mehrspielermodus. Während der nächsten Jahre werden Bugs behoben und Inhalte hinzugefügt um Spieler zurück zu bringen. Dieser Trend ist in der jetzigen Form zwar nicht wünschenswert, es ist jedoch trotzdem wichtig, dass Entwickler die Möglichkeit dazu haben.

3.1.2 Remote Settings

Viele Mehrspielerspiele, wie „First-Person-Shooter“, MOBAs oder MMOs, benötigen regelmäßige Patches, um die Spielbalance zu verändern. Dieses Thema ist enorm komplex, da viele Räder ineinander greifen. „League of Legends“ ist seit über 10 Jahren, mit

über 200 Millionen registrierten Nutzern, aktiv. Unabhängig davon wird alle 14 Tage ein Patch ausgerollt, der bestimmte Charaktere stärker und andere wieder schwächer macht. Diese Patches will der Service „Remote Settings“ ablösen, indem er Entwicklern die Möglichkeit gibt, bestimmte Werte mit der Cloud zu synchronisieren, sodass diese jederzeit über ein Webinterface, ohne zusätzliche Patches, geändert werden können. Patches sind somit nur mehr für Fehlerbehebungen und zusätzliche Inhalte nötig.

3.1.3 Unity Asset Store

Der Asset Store ist direkt in die Engine eingebaut und ist somit ein zentraler Aspekt von Unity. Er ist vor allem für Prototypen enorm wichtig und hilft Entwicklern schnell ihre Ideen zu testen und verbessern. Im Store können Nutzer Scripts, Modelle, Animationen, Tools und vieles mehr hochladen und auch herunterladen. Eine simple, aber effektive Review-Implementation ist vorhanden um hochwertige Assets zu präferieren. Inhalte sind für fast jedes Szenario und jede Idee vorhanden und deren Preise reichen von kostenlos bis zu mehr als 100 Euro. Es sind viele Modellpakete vorhanden, die oft gratis sind und für die Prototypenphase, als Platzhalter, verwendet werden. Viele Entwickler laden auch selbst entwickelte Tools, für die Kameraführung oder die Terraingestaltung, hoch. 2017 sparten Entwickler über 1 Milliarde Dollar durch die Benutzung von Assets aus dem Store. Der Store ist aber auch zum Teil Schuld an der schlechten Reputation, die Unity, seit einiger Zeit, plagt. Unzählige Spiele werden täglich in die mobilen App Stores hochgeladen, die nur aus Asset Store Inhalten bestehen, um billige Kopien bekannter Spiele zu machen. Auch ein Grund ist, die Verwendung der Marke Unity selbst. Unity bietet eine Gratisversion, dessen größte Unterscheidung die Inkludierung des Unity Logos, am Start des Spiels, ist. Bei Bezahlversionen hat man die Wahl und da die Erscheinung des Logos meist für billige Spiele steht, entschließen sich die meisten dagegen. Die Zeit des Hochfahrens wird bei Spielen meist für die zur Schau Stellung der Technologien verwendet und die Gameengines in Verwendung entwerfen oft angepasste Logos für wichtige Spieleveröffentlichungen. Unity ist die meist verwendetste Engine am Markt und auch die erste Wahl für viele Indie Entwickler. Spiele, wie „Ori and the Blind“, „Cuphead“, „Superhot“, „Hollow Knight“, „Subnautica“ oder „Overcooked“, sind eine der größten Indie Erfolgsgeschichten in den letzten Jahren aber auch Smartphone Klassiker, wie „Angry Birds“, „Temple Run“, „Crossy Road“ oder „Pokemon Go“ sind mit Unity entwickelt worden. Jedoch erscheint das Unity Logo kaum bei Spielstart, weswegen man diese Spiele nicht sofort damit in Verbindung bringt.

3.1.4 Unity Collaborate

Unity Collaborate ist Unitys in die Engine eingebaute Git Alternative, die ein einfaches Zusammenarbeiten ermöglichen soll. Es gibt gewisse Einschränkungen, die unter gewissen Bedingungen KO-Kriterien sein können. Für Teams bis vier Personen ist der Dienst zwar gratis, werden Accounts jedoch auch mehreren Geräten genutzt, kann es zu Fehlern kommen, die behoben werden können, sofern man sich für eine der Bezahloptionen entscheidet. Unity Collaborate ist ein simples Tool, jedoch ist das neben der größten Stärke

auch eine nicht zu vernachlässigende Schwäche. Ohne Erweiterungen kann der Dienst keine Dateiinhalte lesen und so Dateien zusammenfügen, wie es Git zum Beispiel kann. Es kann nur Unterscheiden in geändert oder eben nicht. Je nach Workflow kann das zum Problem werden. Ist eine strikte Arbeitsteilung vorhanden, so reicht diese Implementierung aus und macht den Dienst, so zur idealen Wahl. Arbeiten oft mehrere Leute an der selben Datei, so kann man Erweiterungen einbauen, die zusätzliche Funktionalität bieten können. Leider bietet Unity keine solchen Erweiterungen und man ist auf Dritte angewiesen.

3.1.5 Unity Preispolitik

Unity bietet hier drei Optionen, die sich abhängig von der Teamgröße, im Preis ändern können. Für Einsteiger und Solo Entwickler ist die Gratis Version von Unity ausreichend. Man ist in keiner Weise in den Plattformen, für die Spiele entwickelt werden können, eingeschränkt und auch der Großteil der Services ist gratis verfügbar. Support wird von Unity nicht bereit gestellt. Um diesen in eingeschränkter Form genießen zu können, wird die, 25 Dollar pro Monat kostende, Plus Version benötigt. Zusätzlicher Cloud Speicherplatz und Zugang zu einem Übungskurs sind zusätzliche Boni. Um sowohl einen „Dark Mode“, als auch ein T-Shirt zu bekommen, benötigt man die, 125 Dollar kostende, Pro Version, die alle Features, wie 20 Prozent Rabatt auf Premium Assets, mehr Cloud Speicherplatz, Zugang zu Entwicklerkonferenzen und vieles mehr freischaltet. Für größere Teams bietet Unity individuelle Lösungen. Selbst wenn man die zusätzlichen Features der Bezahlversionen nicht benötigt, ist man ab einem gewissen Umsatz dazu verplichtet.

3.1.6 Plattformen

Unity unterstützt seit vielen Jahren alle neuen Plattformen, selbst wenn diese selbst noch in Entwicklung sind. Neben den Standardplattformen, wie X-Box, Playstation, Nintendo Switch, Android, Ios, Windows, Linux und Mac werden auch Fernseherbetriebssysteme, wie Samsungs Tizen OS, Apples tvOS oder Googles Android TV unterstützt. Plattformen der Zukunft wie Microsoft Mixed Reality, Gear VR, Google Cardboard, AR-Core und viele mehr, die sich auf Augmented Reality oder Virtual Reality fokussieren, werden zahlreich unterstützt, weshalb Unity hier eine Marktdominanz zeigt. Spiele werden meist für mehrere Plattformen veröffentlicht und ein Vorteil, den Unity bietet, ist es, die selbe Codebasis für alle Plattformen zu verwenden. Ludimus hat sowohl einen Android, als auch einen Windows Client, dennoch können beide Plattformen mit dem selben Code funktionieren. Muss oder soll es jedoch Unterscheidungen zwischen Plattformen geben, so bietet Unity auch hier eine Lösung. Plattformspezifischer Code wird für alle nicht relevanten Plattformen, aus dem Code heraus geschnitten. Häufige Anwendungen hierfür sind Filesysteme und deren verschiedene Pfade.

3.1.7 Mobile Entwicklung

Neben der Virtual und der Augmented Reality, ist Unitys Hauptverwendungszweck die mobile Entwicklung für Ios und Android. Der Vorteil der gleichen Codebasis ist hier enorm wichtig, da diese zwei Systeme sehr unterschiedliche Entwicklungsworkflows haben und es nicht effizient ist, für mehrere Plattformen einzeln zu entwickeln. Selbst große Publisher wie Square Enix, greifen bei ihrem „Tomb Raider“ Smartphone Ableger auf Unity zurück, da sie laut eigener Angabe schnell einen vorzeigbaren Prototypen bauen konnten. Auch Blizzard benutzte Unity zur Entwicklung von ihrem Kartenspiel „Hearthstone“, welches sowohl einen Android und Ios, als auch einen Windows Client, besitzt und auch sie erwähnen die schnelle Entwicklungszeit. Es sind Erfolgsgeschichten, wie diese weswegen Unity unter Entwicklern einen sehr guten Ruf genießt, auch wenn die Öffentlichkeit davon nichts mitbekommt.

3.1.8 Community

Durch den großen Erfolg bildete sich schon zu Beginn eine riesige Community, die durch Tutorials und Livestreams von Unity selbst inspiriert wurde. Hunderte hochqualifizierte Entwickler teilen deren Wissen mit Anfängern und helfen ihnen auf ihrem Weg zum ersten Erfolg. Die Tutorials reichen von Programmierkursen, über die Modellierung, bis hin zur Integration anderer Dienste in Unity. Die Gemeinschaft ist groß genug, sodass jedes gewünschte Verhalten dokumentiert worden ist, oder jeder aufgetretene Fehler behoben worden ist.

3.1.9 Integration

Unity bietet Integrationen mit vielen Modellierungs-, Zeichen- und Audiotools. Die zwei am meisten vorkommenden Kombinationen sind wohl: Unity und Blender und Unity und Photoshop.

3.1.9.1 Unity und Blender

Blender ist eine gratis Open Source Modellierungssoftware, die in viele Formate exportieren kann und auch eine ähnliche Community, wie Unity, bietet. In vielen Tutorials werden Unity und Blender gemeinsam verwendet, da beide Programme zwar Einsteigerfreundlich sind, jedoch auch enormes Potenzial haben, sofern gebraucht.

3.1.9.2 Unity und Photoshop

Für die 2D Spieleentwicklung sind Sprites, also Bilder, enorm wichtig. Viele Guides empfehlen hier meist Photoshop, den sowohl verschiedene Layer als auch Animationen werden perfekt unterstützt, um einen reibungslosen Workflow zu garantieren.

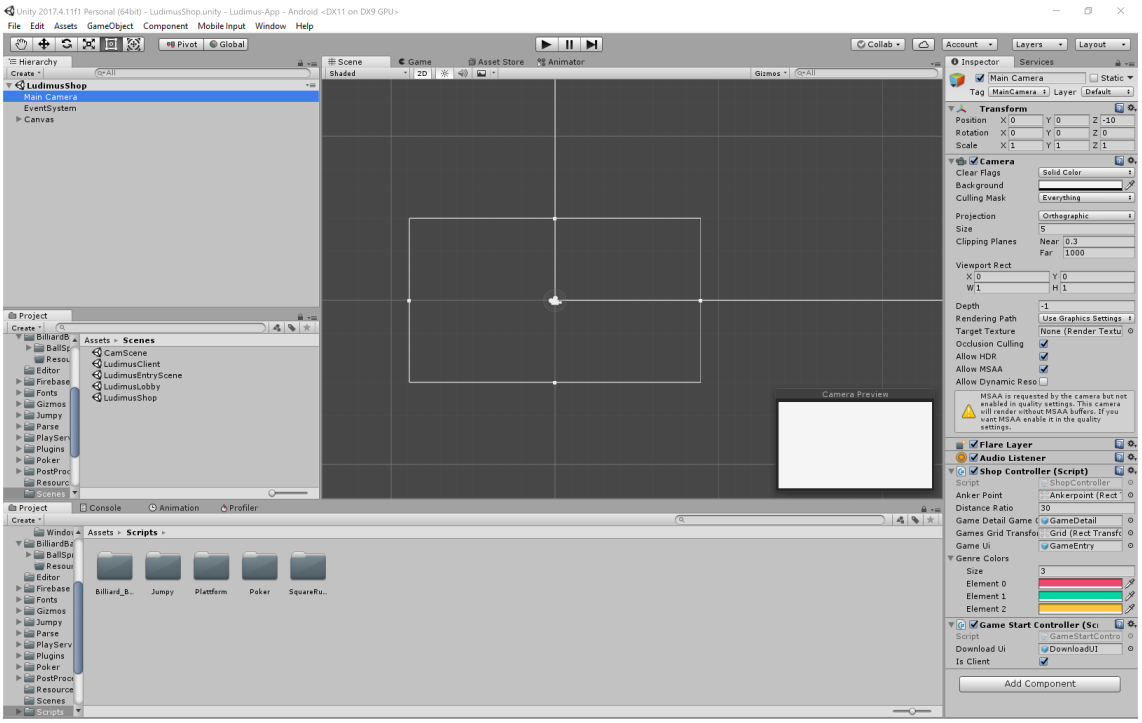

Abbildung 3.1: Unity Editor

3.2 Entwicklung mit Unity (DM)

Die Entwicklung in Unity ist in viele kleine Bereiche aufgeteilt. Logik implementiert man jedoch immer in C-Sharp, sofern man keine explizite Lizenz für C++ erworben hat. Im Laufe des Jahres 2019, soll jedoch eine Visual Scripting Lösung getestet werden. Visual Scripting ist eine Drag and Drop Art der Programmierung. Ein Programm besteht hier aus Nodes. Diese Nodes haben bestimmte Funktionen, die miteinander verbunden werden können. Entwickler können eigene Nodes schreiben, um die Grundfunktionalität zu erweitern. Das bekannteste Beispiel dieser Entwicklungsmethode ist die Unreal Engine 4, die dieses Konzept schon seit Beginn nutzt.

3.2.1 Editor

Der Editor ist in Tabs gegliedert, wie in Abbildung 3.1 zu sehen, die beliebig angeordnet werden können. Wichtige Tabs sind:

3.2.1.1 Hierarchy

Hier werden alle Objekte, die in der aktuell ausgewählten Szene sind, angezeigt. Objekte können verschachtelt sein oder unabhängig voneinander herumverschoben werden. Die Details der ausgewählten Elemente werden im Inspector angezeigt.

3.2.1.2 Scene

Das Scene Fenster zeigt die Welt, die aktuell ausgewählte Szene, in der die Objekte existieren. Alle Objekte haben eine eindeutige Position, mit einer x, y und z Koordinate. Der Benutzer kann sich in diesem Dreidimensionalen Raum frei bewegen und Objekte anordnen, um seine Level zu gestalten. Da Unity sowohl 2D als auch 3D Entwicklung unterstützt, kann man die Perspektive einstellen.

3.2.1.3 Project

Der Project Tab die In Engine Version des Fileexplorers. Er verfügt über eine schnellere Suchfunktion und ein Favoritensystem. Hier ausgewählte Objekte können sowohl in die aktuelle Szene hineingezogen werden oder die Details, im Inspector, angesehen werden.

3.2.1.4 Console

Die Konsole zeigt alle Konsolen Einträge auf, gruppiert nach Art, sprich „log“, „warning“ oder „error“. Wählt man einen Eintrag aus, so sieht man dessen Details und kann auch zur Zeile im Code springen, der diesen ausgelöst hat. Die Standard Lösung von Unity verfügt über keine Suchfunktion, jedoch gibt es Tools im Asset Store, die diese Lücke füllen.

3.2.1.5 Animation

Soll sein Spiel über bewegliche Inhalte verfügen, so sind Animationen oft eine gute Lösung. Unity verwendet hier das Keyframe Prinzip, bei dem alle Eigenschaften zu zwei bestimmten Zeitpunkten, Keyframes, gespeichert werden, und anschließend alle Änderungen über die Zeit verteilt durch geführt werden. Unity unterstützt diese Art der Animation für 3D-Modelle, Sprites und UI-Elemente. 3D-Modelle und Sprites können mit einem Skelett versehen werden, dass bewegt werden kann um auch das Modell selbst zu bewegen.

3.2.1.6 Game

Ist es Zeit das Programm zu testen, so kann man es mithilfe des großen Spielen-Knopf starten. Unity wechselt danach automatisch in den Game Tab. Hier sieht man seine Szene durch eine Kamera, genauso wie sie Spieler später sehen würden. Dieser Modus ist für schnelle Testläufe ideal. Es gibt jedoch unterschiede zwischen dieser spielbaren Version und einer gebauten App, weshalb es ratsam ist, beide Versionen zu überprüfen.

3.2.2 Komponenten

Objekte in Unity bestehen aus Komponenten. Diese geben ihnen ihre Funktionalität und bieten gleichzeitig Flexibilität. Alle vom Entwickler generierten Scripts erben automatisch von der Basisklasse MonoBehaviour, wodurch sie als Komponenten verwendet werden können. Als Basis bietet Unity jedoch schon eine Reihe an Komponenten an,

um Beispielsweise ein Objekt auf Gravitation reagieren zu lassen. Diese bereitgestellten Komponenten sind eingeteilt in: Mesh, Effects, Physics, Navigation, Audio, Video, Rendering, UI, AR und vielen mehr.

3.2.2.1 Mesh

Diese Gruppe besteht aus 3 verschiedenen und essentiellen Komponenten, für die 3D Entwicklung. Der Mesh Filter gibt an welches Modell dieses Objekt darstellen soll. Der Mesh Renderer hingegen beschreibt, welche Materialien das Objekt bekommt oder ob es Schatten erhalten oder werfen soll. Um 3D Text in der Szene anzuzeigen, benötigt man den Text Mesh Komponenten.

3.2.2.2 Effects

Egal ob Lagerfeuer, kleine Luftsteifen, um Bewegung deutlicher zu machen, oder Lens Flare Effekte, wenn der Spieler in die Sonne sieht, Effekte sind in Spielen häufig verwendet und Unity bietet hier eine gute Basis um seine eigenen Effekte zu erzeugen.

3.2.2.3 Physics

Dieser Punkt ist unterteilt in 2D und 3D Physics, um für Klarheit zu sorgen.

3.2.2.3.1 2D Physics Um die vorher erwähnte Gravitation für dieses Objekt zu aktivieren, benötigt man einen Rigidbody, hier konkret Rigidbody2D. Unity bietet eine globale Vektor Variable für die Gravitation, mit einer x und einer y Richtung. Im Rigidbody kann man die Stärke des Effekts auf dieses eine Objekt verändern. Die Werte für Masse, Trägheit und verschiedene Physics Materialien können eingetragen werden. Letztere bestimmen, wie stark das Objekt von Reibung beeinflusst wird und wie federnd es ist. Objekte können auch in ihrer Bewegung oder Rotation für verschiedene Achsen eingeschränkt werden. Zusätzlich zum Rigidbody findet man hier alle Arten von Collidern, die Unity für 2D zu bieten hat. Diese Collider beschreiben die Zone, in der Kollisionen registriert werden. Collider können miteinander verknüpft werden, um die Objekte perfekt einzuhüllen. Box-, Circle-, Edge- und Polygon-Collider sind die am häufigsten verwendeten. Während Box- und Circle-Collider nur geometrische Formen beschreiben und lediglich in deren Größe geändert werden können, bieten sowohl Edge-, als auch Polygon-Collider Möglichkeiten, die für 3D nicht denkbar sind. Edge-Collider werden oft für Plattformen verwendet, da sie nur eine Linie darstellen, die an verschiedenen Punkten abgebogen werden kann. Das selbe Prinzip verfolgt der Polygon-Collider, nur verbindet er Start- und Endpunkt miteinander. Hinzukommt eine automatische Erkennung, der Form, bei Sprites. Als dritte Gruppe dieser Kategorie gelten die Joints. In verschiedenen Ausführung regeln sie das Verhalten zwischen zwei Objekten. Räder, Dämpfungen, Seile und mehr können so realisiert werden. Zu guter Letzt gibt es noch Effectors. Diese regeln die Bewegungen, wenn zwei Objekte miteinander kollidieren. So können „One-Way“ Plattformen oder Förderbänder geschaffen werden.

3.2.2.3.2 3D Physics Die Rigidbody Implementierung für den dreidimensionalen Raum stellt eine abgespeckte Version, im Vergleich zur 2D, dar. Hier kann die Gravitation nur ein- oder ausgeschaltet werden und keine Physics Materialien zugeteilt werden. Bei den Collidern finden sich Implementierungen der Standard Formen, wie Würfel oder Kreis ,wieder, hinzugekommen sind jedoch der Mesh Collider, der eine ähnliche Funktionsweise, wie der Polygon-Collider, darstellt und der Wheel Collider, der detaillierte Entscheidungsmöglichkeiten bezüglich Reifendruck, Federung, Reibung, in verschiedene Richtungen, und vielen mehr, bietet. Auch Effectors werden hier wieder aufgelistet, jedoch ist sticht die Cloth Komponente hier heraus. Sie ermöglich es Kleidung zu simulieren, um zum Beispiel im Wind zu flattern.

3.2.2.4 Navigation

Navigation ist nur für 3d Projekte verwendbar und ermöglicht es Entwicklern für jeder Objekt einzustellen, ob es begehbar ist, um anschließend Gegnerobjekten, oder anderen Objekten, die Wegfindung benötigen, einen sogenannten Nav Mesh Agent zu geben. Dieser Agent enthält Werte für Höhe, die maximale Schräge, die er bezwingen kann, und die maximale Stufenhöhe. Der Entwickler muss nur noch ein Ziel übergeben und das Objekt kann sich selbstständig dorthin bewegen.

3.2.2.5 Audio

Hier können Entwickler Objekte mit Fähigkeiten für das Zuhören oder das Sprechen ausstatten. Bestimmte Audiofilter für Echoeffekte können auch erstellt werden.

3.2.2.6 Video

Als einziges Element in dieser Gruppe, ist der Video Player das Werkzeug zum Präsentieren von vorher gerenderten Zwischenszenen.

3.2.2.7 Rendering

In jedem Spiel wird eine virtuelle Kamera verwendet, durch deren Linse gespielt wird. Diese Kamera Komponente ist unter dem Begriff Rendering, mit einer Skybox-, einer Licht- und der Sprite Renderer Komponente zu finden.

3.2.2.7.1 Camera Hier können wichtige Einstellung bezüglich der Darstellung getroffen werden. Von oben beginnend, kann der Entwickler gleich den Hintergrund festlegen. Standardmäßig ist er blau. Neben Farben können Entwickler jedoch auch Skyboxen auswählen. Mit der Option der Culling Mask, können bestimmte Schichten aktiviert oder deaktiviert werden. Die Projektion beschreibt die Art, wie mit entfernten Objekten umgegangen werden soll. Während Perspective Objekte, die in der Ferne liegen kleiner darstellt und so für 3D Spiel ideal ist, behält Orthograpfic die Originalgröße der Objekte unabhängig von deren Entfernung zu Kamera bei und eignet sich somit für 2D Spiele. Abhängig davon kann der Entwickler das Field of View oder die Kameragröße einstellen.

Um zu verändern wie nah oder fern Objekte von der Kamera entfernt sein dürfen um gesehen zu werden, dient die Clipping Planes Option. Die restlichen Optionen sind für verschiedene Rendering Arten oder ob das gerenderte Bild gespeichert werden soll, um eine Karte zu erzeugen.

3.2.2.7.2 Skybox Die Skybox Komponente wird verwendet, um einen Himmel zu simulieren. Dazu benötigt man ein Skybox Material, welches dann hier verwendet werden kann.

3.2.2.7.3 Light Die Licht Komponente wird verwendet um Sonnenlicht, Licht von Lampen, Lagerfeuer und mehr zu simulieren. Dieses breite Aufgabengebiet wird mittels Untertypen realisiert.

PointLight Punktlichter erleuchten einen bestimmten Bereich von einem Punkt aus in alle Richtungen. Lagerfeuer oder Lampen sind hier die Einsatzzwecke.

Arealight Licht wird von einem bestimmten Punkt in einer Kegelform, in eine Richtung, versendet. Diese Methode ist für Scheinwerfer oder Spezielle Straßenlaternen sehr nützlich.

Directionlight Um die Sonne zu simulieren, werden direktionale Lichter verwendet. Diese werfen Licht von überall in eine bestimmte Richtung, weshalb die Position keine Rolle spielt.

3.2.2.7.4 Sprite Renderer Diese Komponente ermöglicht es Sprites in der Welt anzuzeigen. Das ausgewählte Sprite kann in der Farbe im Editor geändert werden, weshalb Sprites oft in Grautönen eingelesen werden, um sie in der Engine anzupassen, durch die Änderung des Order Layer vor oder hinter anderen Sprites angezeigt werden oder durch Materialen bestimmte Effekte, wie Wasserreflexionen, erhalten.

3.2.2.8 UI

Alle Komponenten, die mit der Erstellung eines Userinterfaces zusammen hängen, sind hier untergebracht. Es sind Presets für Buttons, Dropdown Menüs, Eingabefelder und vieles mehr vorhanden. UI-Elemente werden wie normale Objekte im dreidimensionalen Raum gesehen, weshalb Elemente auch nicht mit Code, wie html oder css, erzeugt und verändert werden können.

3.2.2.9 AR

Je nach Plattform ändert sich die Größe dieses Menüs. SDKs für Geräte ähnlich der HoloLens bieten hier Komponenten für Spatial Recognition und mehr.

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class UnityShowCase : MonoBehaviour {
6
7       // Use this for initialization
8       void Start () {
9
10      }
11
12      // Update is called once per frame
13      void Update () {
14
15      }
16  }
```

Abbildung 3.2: Unity Basisscript

3.2.3 Scripts

Diese Komponenten bieten ein solider Grundkonstrukt. Um Funktionalität in ein Spiel zu bringen werden jedoch, eigens geschriebene, Scripts benötigt, die wie folgt aufgebaut sind. Wie bereits erwähnt, erben diese Programmteile von der Basisklasse MonoBehaviour. Lässt man sich die Scripts in der Engine generieren, so erhält man ein File ähnlich dem in Abbildung 3.2.

3.2.3.1 Lifecycle Methoden

Für Einsteiger sind die zwei generierten Methoden, Start und Update, kommentiert. Unity bietet verschiedene Lifecycle Methoden, die unterschiedliche Zwecke und Anwendungsbereiche haben, an.

3.2.3.1.1 Awake Die erste Methode, die für jedes Objekt aufgerufen wird ist Awake. Hier soll Code ausgeführt werden, der nur einmal ausgeführt werden darf.

3.2.3.1.2 OnEnable Objekte in Unity können jederzeit aktiviert oder deaktiviert werden. Werden Objekte aktiviert wird diese Methode aufgerufen. Diese Methode kann genutzt werden um andere Objekte auch zu aktivieren.

3.2.3.1.3 Start Start wird immer nach OnEnable aufgerufen und wird primär für die Verknüpfung zwischen Komponenten benutzt.

3.2.3.1.4 FixedUpdate Unity unterscheidet in zwei verschiedene Updatemethoden. FixedUpdate wird, wie der Name vermuten lässt, immer in selben Zeitintervallen aufgerufen, während Update immer dann verwendet wird, sobald ein neues Bild erschienen

ist. FixedUpdate wird für Physikberechnungen und Eingabeüberprüfungen verwendet, die zwischengespeichert werden um in der Update Methode verwendet zu werden.

3.2.3.1.5 Update Hier wird die Logik des Spiels behandelt. Sollen Objekte bewegt werden, soll dies mit den vorher eingelesenen Werten hier geschehen. Haben bestimmte Aktivitäten eine Abklingzeit, bevor sie wieder verwendet werden können, so werden die Überprüfungen hier durchgeführt.

3.2.3.1.6 LateUpdate Wird für Kamerabewegungen genutzt und wird nach allen anderen Updates aufgerufen. Anschließend wird das aktuelle Bild erzeugt und der Kreislauf beginnt bei FixedUpdate erneut.

3.2.3.1.7 OnDisable Wird ein Objekt während dieses Bildes deaktiviert, so wird diese Methode verwendet um zu reagieren.

3.2.3.1.8 OnDestroy Wird dieses Objekt zerstört, zum Bespiel sobald der Spieler verloren hat, so dient diese Methode um den Score zu speichern oder ein bestimmtes UI aufzurufen.

3.2.3.2 Kollisionen

Kollisionen passieren, wenn zwei Collider desselben Typs, sprich 2D mit 2D und 3D mit 3D, miteinander kollidieren. Abhängig von deren Eigenschaften werden verschiedene Events aufgerufen.

3.2.3.2.1 OnCollisionEnter Kollidiert ein Collider mit einem anderen und beide haben die Standardeinstellungen ausgewählt, so wird diese Methode während dem Bild der Berührung aufgerufen.

3.2.3.2.2 OnCollisionEnter2D Ist das 2D Abbild der oben beschriebenen Methode.

3.2.3.2.3 OnCollisionStay/OnCollisionStay2D Solange zwei Collider miteinander kollidieren, wird diese Methode benutzt.

3.2.3.2.4 OnCollisionExit/OnCollisionExt2D Trennen sich die zwei Objekte nun wieder, kommt diese Methode zum Einsatz.

3.2.3.3 Trigger

Jeder Collider hat die Option ein Trigger zu sein oder nicht. Trigger registrieren zwar auf Kollisionen im Code, prallen jedoch nicht voneinander ab. Kollidiert ein Objekt, welches

```
GameObject newObject = Instantiate(Resources.Load("myObject")) as GameObject;
```

Abbildung 3.3: Objekt erzeugen über Resources

```
GameObject newObject = Instantiate(gameObject);
```

Abbildung 3.4: Objekt erzeugen durch Klonen eines Objektes

einen Collider mit einem aktiven Trigger hat, mit einem anderen Objekt, unabhängig von dessen Optionen, so werden folgende Events stattdessen ausgeführt.

OnTriggerEnter/OnTriggerEnter2D

OnTriggerStay/OnTriggerStay2D

OnTriggerExit/OnTriggerExit2D

3.2.3.4 Objekte erzeugen

Im Editor können Objekte mit Drag and Drop in die Szene hineingebracht werden. Will man jedoch zur Laufzeit Objekte erzeugen, so funktioniert dieser Ansatz nicht. Die Methode Instantiate erledigt dies, es gibt jedoch verschiedene Möglichkeiten diese zu benutzen. Durch die letzten Parameter kann eingestellt werden, wo und als Kind welchen Elements das Objekt erzeugt wird. Im ersten Parameter muss man das Objekt selbst angeben, wobei der Entwickler hier 3 Optionen hat.

3.2.3.4.1 Resources Bestimmte Ordnernamen sind mit speziellen Funktionen belegt. Der StreamingAssets Ordner zum Bespiel ist für nicht zu kompilierende Inhalte bereitgestellt. Um Objekte einfach zu erzeugen, dient der Resources Ordner. Gespeicherte Objekte können aus diesem über ihren Name erzeugt werden mit der Zeile aus Abbildung 3.3.

3.2.3.4.2 Klonen Um Objekte schnell klonen zu können, dient die Zeile aus 3.4.

3.2.3.4.3 Referenzen Die selbe Syntax, jedoch einen anderen Verwendungszweck hat die Erstellung von Objekten mittels Referenzen. Ein Objekt kann als öffentliche Variable gespeichert werden, um später eine Instanz davon zu erstellen.

3.2.3.5 Öffentliche Variable und Verlinkung zwischen Komponenten

Unity bietet mehrere Möglichkeiten für die Erstellung von Referenzen zwischen Komponenten, wobei deren Einsatz sich deutlich unterscheidet. Um auf Komponenten eines Objektes zugreifen zu können, benötigt man zuerst eine Referenz auf dieses Objekt selbst. Die Methode GetComponent() gibt anschließend den gewünschten Komponenten zurück. Referenzen auf Objekte kann man durch Filtern aller Objekte bekommen,

```
foreach (GameObject g in GameObject.FindGameObjectsWithTag("myTag"))
{

}

foreach (object o in GameObject.FindObjectsOfType<MyType>())
{

}
```

Abbildung 3.5: Beschaffung aller Objekte mit einer bestimmten Eigenschaft

```
[SerializeField]
private float myField;
```

Abbildung 3.6: Privates Feld mit SerializeField Annotation

zusehen in Abbildung 3.5. Objekte werden meist mit Tags versehen, weshalb die erste Methode öfter verwendet wird. Will man jedoch nur ein bestimmtes Objekt, so ist es nicht ratsam alle Objekte auf deren Namen zu überprüfen. Hier bietet Unity ein extrem brauchbares Features, welches bei öffentlichen Variablen Standard ist, jedoch mit der Annotation SerializeField auch für private Felder verwendet werden kann, siehe Abbildungen 3.6 und 3.8. Die Werte dieser Felder können über den Inspector gesetzt werden, wie in 3.7 und 3.9 zu sehen.

Erstellt man öffentliche Felder mit den Komponenten oder einem Objekt als Typ, wie in Abbildung 3.8, so kann man diese im Inspector per Drag and Drop belegen, zu sehen in 3.9.

Diese Methode wird verwendet, wenn Referenzen auf andere Objekte nötig sind. Den eigenen Rigidbody, sollte man jedoch nicht so referenzieren. Hier sollte eine private Variable angelegt werden, die mit GetComponent() in der Start Methode initialisiert wird, siehe Abbildung 3.10.

3.2.4 Alternativen

Unity ist nicht die einzige Spieleengine und andere Optionen wie die Unreal Engine 4 oder Gamemaker Studio, wären valide Optionen gewesen. Unity bietet jedoch soliden Support für sowohl 2D als auch 3D, wodurch wir in keinster Weise eingeschränkt waren. Den größten Vorteil der Unreal Engine, die unglaublich mächtige Rendering Pipeline, würde von unseren Spielen nie ausgereizt worden, weshalb wir uns für die Unity Game

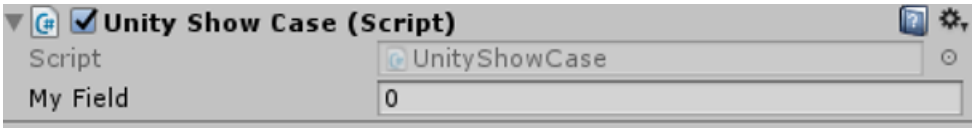

Abbildung 3.7: Inspector Ansicht von Feldern

```
public Rigidbody MyRigidbody;

public GameObject MyGameObject;
```

Abbildung 3.8: Öffentliche Felder mit komplexen Datentypen

Abbildung 3.9: Inspector Ansicht von mehreren komplexen Feldern

Engine entschieden, nicht zuletzt wegen unserer schon vorhandenen Erfahrung und der riesigen Community.

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

Abbildung 3.10: Rigidbody des eigenen Objektes erhalten

3.3 Firebase (DM)

Firebase ist eine einfach zu bedienende Serveralternative, die Datenbank, Fileserver, Authentifizierung, Website Hosting und mehr über ein Webinterface vereint. Im Vergleich zu regulären Servern macht es Firebase einfacher für den Entwickler, sodass dieser mehr Zeit in sein Produkt selbst investieren kann. Firebase bietet fortgeschrittene Sicherheitsmechanismen, Synchronisierung mehrerer Serverstandorte, Erreichbarkeit immer und überall und Offline Funktionalität. Alle Produkte sind mit verschiedensten Anmeldeoptionen verknüpfbar, wie Google Sign-In, Facebook Sign-In, Email/Passwort, Google Play, usw..

3.3.1 Datenbank

Firebase bietet hier zwei Möglichkeiten zur Speicherung von Daten, wobei beide auf NoSql-Datenbanken zurückgreifen. Daten können über das Webinterface angesehen, geändert und auch hinzugefügt werden.

3.3.1.1 Cloud Firestore

Diese neue Datenbanklösung ist zwar momentan noch in der Beta, ist jedoch von Beginn an für Skalierbarkeit optimiert. Wie für NoSql-Datenbanken üblich, wird enormer Fokus auf Lesegeschwindigkeiten und weniger auf Schreibgeschwindigkeiten gelegt. Verstärkt wird dieser Effekt dadurch, dass Indexe auf alle Felder gelegt werden um die gleichen Datenzugriffszeiten für 100 als auch für 100 Millionen Daten zu garantieren. Cloud Firestore bietet momentan SDKss für Android und Ios Apps, Web, Node, Java, Python und Go an. Der Service verfügt unter Android, Ios und Web über die Möglichkeit, Daten lokal zu zwischenspeichern, wenn die Verbindung einmal verloren geht und diese dann mit dem Server zu synchronisieren, sobald das Problem behoben worden ist. Cloud Firestore ist eine dokumentenbasierte Lösung, was bedeutet das Objekte als Dokumente in einer Collection gespeichert werden. Diese Art der Speicherung wird oft als semistrukturierte Datenspeicherung bezeichnet. Dokumente haben verschiedene Felder, wobei Felder wieder Collections mit Dokumenten enthalten können. Diese sogenannten Subcollections ermöglichen in der Theorie zwar ein 1:1 Abbild der Datenstruktur aus Objektorientierten Sprachen, da diese jedoch nur einzeln gequeried werden können und nicht inkludiert werden können, verursacht diese Methode nur mehr Komplikationen. Eine Realisierung mittels Arrays von Referenzen, wäre hier die einfachere Variante, da Arrays inkludiert

werden und, sofern das Programm das einlesen dieser händelt, einfach geparst werden können. Dieser Aufbau ähnelt sehr einer Relationalen Datenbankstruktur.

3.3.1.2 Realtime Database

Realtime Database bietet viele der selben Vorteile, wie Cloud Firestore. Nur der Fokus auf Skalierbarkeit ist nicht so stark ausgeprägt, jedoch ist das System stabiler, da es schon länger auf dem Markt verfügbar ist. SDKs gibt es für Android, Ios, Web, C++ und Unity. Neben der besseren Verfügbarkeit der SDK für Unity war ein Mitentscheidungsgrund die Stabilität, die die Realtime Database bietet. Die Synchronisierung Offline geänderter Daten ist für Android und Ios verfügbar, nicht jedoch für Webanwendungen. Wir benutzen jedoch keine der drei Plattformen, und selbst wenn können unsere Nutzer keine Daten ändern oder einfügen, weshalb dieses Feature keine Bedeutung für uns hatte. Die Skalierbarkeit ist momentan noch kein Bedenken und der Umstieg auf Cloud Firestore ist auch während der Entwicklung noch leicht genug. Die Realtime Database benutzt die Key-Value Notation für die Datenspeicherung. Diese Schreibweise ähnelt JSON sehr, wodurch das einlesen und auslesen von Daten deutlich erleichtert wird.

Wir entschieden uns für die Realtime Database aus mehreren Gründen. Die Verfügbarkeit der SDK für Unity ist ein KO-Kriterium, da diese Technologie nicht änderbar ist und die Verwendung der Rest-Schnittstelle von Cloud Firestore außer Frage stand. Zusätzliche Features wie bessere Skalierbarkeit, bessere Offline Funktionalitäten oder strukturiertere Daten bringen wenig bis keinen Vorteil, da unsere Datenbank nur die Spiele speichert, die zur Verfügung stehen, diese keine komplexen Attribute haben, welche Referenzen benötigen würden und wir keine Offline Funktionalität brauchen. Falls die Datenbank jedoch über eine gewisse Größe hinweg wachsen würden, wäre ein Umstieg auf Cloud Firebase denkbar, da die Preispolitik diesen Usecase unterstützt.

3.3.2 Storage

Firebase bietet ein simples Filesystem, mit dem man Dateien in Ordnerstrukturen speichern kann. Von jeder Datei wird die Größe, der Typ, das Erstell- und Änderungsdatum und die Download URL angezeigt. Durch die SDKs ist die Navigation durch die Ordnerstruktur und das downloaden von Dateien unkompliziert. Als Administrator erhält man eine Übersicht über alle Downloads, die momentane Speichergröße und die momentane Anzahl an Elementen, für einen bestimmten Zeitraum.

3.3.3 Hosting

Firebase ermöglicht es auch eine Webseite zu hosten. Eine Übersicht über alle Uploads mit deren User ist vorhanden um auch zwischen Versionen zu wechseln. Man kann seine Seite auch mit einer eigenen Domain verbinden.

3.3.4 Authentifizierung

Wie bereits erwähnt bietet Firebase hier viele Möglichkeiten für Entwickler Ihre User anzumelden. Anmeldeoptionen von etablierten Netzwerken wie Facebook, Twitter, GitHub, Google Play oder Google sind unterstützt, ebenso wie Telefonauthentifierung, Gastaccounts oder Authentifizierung über Email und Passwort. Es stehen auch Templates für Email-Adressen Bestätigung, Passwort zurücksetzen, usw. zur Verfügung. Im Webinterface ist eine Auflistung aller registrierten Accounts, mit deren Anmeldeoption, dem Datum der Erstellung und dem Datum des letzten Logins, ersichtlich. Werden Telefonbestätigungen verwendet, sieht man diese als Graph dargestellt. Die Einbindung der Authentifizierung in andere Firebaseprodukte ist über sogenannte Regeln implementiert. Hier kann der Admin einstellen ob, oder wer Schreibzugriff auf die Datenbank hat. Kann jeder schreiben, der eingeloggt ist, so gibt es die globale Variable „auth“, deren Wert man mit null vergleicht. Diese Regeln können mit einem Simulator getestet werden um Fehler zu vermeiden.

3.3.5 Preispolitik

Firebase bietet drei verschiedene Modelle, wobei eines kostenlos ist und die restlichen beiden kostenpflichtig sind. Der „Spark Plan“ ist die gratis Version. Man hat Zugriff auf Datenbank, Fileserver, Website Hosting, Authentifizierung und den Großteil der restlichen Produkte. Jedoch sind bestimmte Nutzungsgrenzen für die verschiedenen Tätigkeiten festgelegt. Die Datenmenge der Realtime Database, darf zum Beispiel nicht über ein Gigabyte groß sein und es dürfen auch nicht über zehn Gigabyte heruntergeladen werden. Für Ludimus sind das jedoch astronomische Grenzen, die nur mit extrem vielen Spielen erreicht werden können. Unsere Anwendung verbraucht vielmehr Fileserverkapazitäten, die bei diesem Modell bei fünf Gigabyte liegen und es dürfen maximal ein Gigabyte pro Tag heruntergeladen werden. Mit genügend Nutzern sind das die ersten Schwellpunkte die überschritten werden. Für 25 Dollar pro Monat kann man Subscriber des „Flame Plans“ sein. Dieser hebt die Nutzungsgrenzen aller Produkte um einiges. Da wir diese Erhöhung jedoch nur für ein Produkt brauchen, und wir hier nicht allein sind, ist die dritte Möglichkeit, der „Blaze Plan“, „Pay as you go“, heißt man zahlt nur, was man benötigt. Zur Schätzung der Kosten bietet Firebase hier einen Simulator und selbst wenn Ludimus viele Nutzer, mit vielen Spielen hätte, würde dieser Plan nur 20 Dollar kosten. Zusätzlich dazu verfügt man dann jedoch auch über alle Funktionen, die Firebase bietet. Bigdata Analysen und mehrere Datenbanken- und Storage-Buckets für Parallelität sind nur einige davon.

3.4 Netzwerkverbindungen (ES)

3.4.1 TCP

3.4.1.1 Definition

Das Transmission Control Protocol (TCP) ist ein zuverlässiges, verbindungsorientiertes, Bytestrom Protokoll. Die Hauptaufgabe von TCP besteht in der Bereitstellung eines sicheren Transports von Daten durch das Netzwerk. TCP ist im RFC 793 definiert. Diese Definitionen wurden im Laufe der Zeit von Fehlern und Inkonsistenzen befreit (RFC 1122) und um einige Anforderungen ergänzt (RFC 1323). Das Transmission Control Protocol stellt die Zuverlässigkeit der Datenübertragung mit einem Mechanismus, der als Positive Acknowledgement with Re-Transmission (PAR) bezeichnet wird, bereit. Dies bedeutet nichts anderes als das, daß das System, welches Daten sendet, die Übertragung der Daten solange wiederholt, bis vom Empfänger der Erhalt der Daten quittiert bzw. positiv bestätigt wird. Die Dateneinheiten, die zwischen den sendenden und empfangenden TCP-Einheiten ausgetauscht werden, heißen Segmente. Ein TCP-Segment besteht aus einem mindestens 20 Byte großen Protokollkopf und den zu übertragenden Daten. In jedem dieser Segmente ist eine Prüfsumme enthalten, anhand derer der Empfänger prüfen kann, ob die Daten fehlerfrei sind. Im Falle einer fehlerfreien Übertragung sendet der Empfänger eine Empfangsbestätigung an den Sender. Andernfalls wird das Datengramm verworfen und keine Empfangsbestätigung verschickt. Ist nach einer bestimmten Zeitperiode (timeout-period) beim Sender keine Empfangsbestätigung eingetroffen, verschickt der Sender das betreffende Segment erneut.

3.4.1.2 Verifizierung

TCP ist ein verbindungsorientiertes Protokoll. Verbindungen werden über ein Dreiwege-Handshake (three-way handshake) aufgebaut. Über das Dreiwege-Handshake werden Steuerinformationen ausgetauscht, die die logische Ende-zu-Ende-Verbindung etablieren. Zum Aufbau einer Verbindung sendet ein Host (Host 1) einem anderen Host (Host 2), mit dem er eine Verbindung aufbauen will, ein Segment, in dem das SYN-Flag gesetzt ist. Mit diesem Segment teilt Host 1 Host 2 mit, das der Aufbau einer Verbindung gewünscht wird. Die Sequenznummer des von Host 1 gesendeten Segments gibt Host 2 außerdem an, welche Sequenznummer Host 1 zur Datenübertragung verwendet. Sequenznummern sind notwendig, um sicherzustellen, daß die Daten vom Sender in der richtigen Reihenfolge beim Empfänger ankommen. Der empfangende Host 2 kann die Verbindung nun annehmen oder ablehnen. Nimmt er die Verbindung an, wird ein Bestätigungssegment gesendet. In diesem Segment sind das SYN-Bit und das ACK-Bit gesetzt. Im Feld für die Quittungsnummer bestätigt Host 2 die Sequenznummer von Host 1, dadurch, daß die um Eins erhöhte Sequenznummer von Host 1 gesendet wird. Die Sequenznummer des Bestätigungssegments von Host 2 an Host 1 informiert Host 1 darüber, mit welcher Sequenznummer beginnend Host 2 die Daten empfängt. Schlußendlich bestätigt Host 1 den Empfang des Bestätigungssegments von Host 2 mit einem Segment, in dem das ACK-Flag gesetzt ist und die um Eins erhöhte Sequenznummer

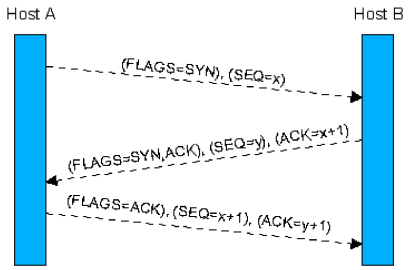

Abbildung 3.11: three-way handshake

von Host 2 im Quittungsnummernfeld eingetragen ist. Mit diesem Segment können auch gleichzeitig die ersten Daten an Host 2 übertragen werden. Nach dem Austausch dieser Informationen hat Host 1 die Bestätigung, daß Host 2 bereit ist Daten zu empfangen. Die Datenübertragung kann nun stattfinden. Eine TCP-Verbindung besteht immer aus genau zwei Endpunkten (Punkt-zu-Punkt-Verbindung).

3.4.2 TCP Client

3.4.2.1 Übersicht

Stellt Clientverbindungen für TCP-Netzwerkdienste bereit. Die TCPClient Klasse bietet einfache Methoden zum Herstellen einer Verbindung, senden und empfangen von Daten. Der TCPClient verbindet sich mit einem TCPListener und stellt danach einen Network Stream zur Verfügung welcher ausgelesen werden kann und auf welchem geschrieben werden kann. Zum Verbinden mit einem TCPListener gibt es zwei Möglichkeiten: Der TCPClient stellt “Connect” Methoden zur Verfügung Einen TCPClient erstellen unter Angabe der IP-Adresse und des Ports. Im Konstruktor wird dann die Verbindung automatisch hergestellt. Wenn die Verbindung besteht kann mit .GetStream() der Network Stream des TCPClients abrufen und diesen mit .Read(Byte[],Int32,Int32) auslesen oder diesen mit .Write(Byte[],Int32,Int32) beschreiben. Wenn der Kommunikationsaustausch abgeschlossen ist sollte der TCPClient mit .Close(Int32) geschlossen werden.

3.4.2.2 Konstruktoren

Name: **TcpClient()**
Info: Erstellt einen TCPClient der noch zu keinem TCPListener verbunden ist.

Name: **TcpClient(IPEndpoint)**
Info: Erstellt einen TCPClient welcher sich im Konstruktor bereits zu dem angegebenen IPEndpoint verbindet.

Name: **TcpClient(string, Int32)**

Info: Erstellt einen TCPClient welcher sich im Konstruktor bereits zu der als string angegebenen IP-Adresse und dem als Int32 angegebenen Port verbindet.

3.4.2.3 Methoden

Name: **Close()**
Info: Schließt die Verbindung zum Remotehost

Name: **Connect()**
Info: Verbindet den TCPClient mit dem Remotehost. Dazu gibt es verschiedene Parameter;

- *Connect(IPAdress, Int32)*, wobei hier die IP-Adresse bereits als IPAdress-Objekt und der Port als Int32 mitgegeben wird.

- *Connect(IPAdress[], Int32)*, wird benutzt wenn der Remotehost mehrere DNS-Einträge hat und man nicht genau weiß welcher der richtige ist. Es wird eine Liste aller IP-Adressen mitgegeben, wobei es sich auch hier um IPAdress-Objekte handelt. Auch hier wird der Port als zweiter Parameter als Int32 mitgegeben.

- *Connect(IPEndPoint)*, hier wird direkt ein IPEndPoint-Objekt mitgegeben, welches bereits alle relevanten Infos über den Remotehost beinhaltet.

- *Connect(string, Int32)*, einfachste Methode da hier die IP-Adresse einfach als string mitgegeben wird. Auch hier wird als zweiter Parameter der Port als Int32 angegeben.

Name: **ConnectAsync()** Info: Verbindet den TCPClient mit dem Remotehost asynchron. Dazu gibt es verschiedene Parameter;

- *ConnectAsync(IPAdress, Int32)*, wobei hier die IP-Adresse bereits als IPAdress-Objekt und der Port als Int32 mitgegeben wird.

- *ConnectAsync(IPAdress[], Int32)*, wird benutzt wenn der Remotehost mehrere DNS-Einträge hat und man nicht genau weiß welcher der richtige ist. Es wird eine Liste aller IP-Adressen mitgegeben, wobei es sich auch hier um IPAdress-Objekte handelt. Auch hier wird der Port als zweiter Parameter als Int32 mitgegeben.

- *ConnectAsync(string, Int32)*, einfachste Methode da hier die IP-Adresse einfach als string mitgegeben wird. Auch hier wird als zweiter Parameter der Port als Int32 angegeben.

Name: **Dispose()**
Info: Gibt alle vom TCPClient verwendeten Ressourcen frei.

Name: **GetStream()**
Info: Gibt den Network Stream zurück, welcher für das Senden und Empfangen von Daten wichtig ist.

3.4.2.4 Verwendung in Ludimus

Da Verbindung zwischen Smartphone und Server(PC/Laptop) baut auf einem Network Stream auf, also ist es notwendig die Verbindung per TCPClient und TCPListener durchzuführen. Zusätzlich bietet TCP hier viele Vorteile welche die Wahl zum TCPClient nicht schwer fiel. In Ludimus ist jedes Smartphone welches sich mit dem Server, per TCP-Verbindung, verbindet ein TCPClient welche in einer Liste gespeichert werden und so für jeden zur Verfügung stehen. Zusätzlich gibt es die .SendData(string key,string value) Methode welche den zugriff auf den Network Stream über den TCPClient löst.

3.4.3 TCP Listener

3.4.3.1 Übersicht

Die TCPListener-Klasse bietet einfache Methoden, die das akzeptieren von eingehenden Verbindungsanfragen. Die TCPListener Klasse wird zusammen mit dem TCPClient benutzt um eine Verbindung miteinander aufzubauen und per Network Stream zu kommunizieren. Dem TCPListener selbst wird dabei im Konstruktor, als Parameter, die IP-Adresse und den Port , auf welchem der TCPListener hören soll mitgegeben. Als IP-Adresse kann auch Any mitgegeben werden um den TCPListener auf der lokalen IP laufen zu lassen. Um den TCPListener zu starten muss einfach .Start() aufgerufen werden und schon horcht der TCPListener auf der angegeben IP-Adresse und Port. Eingehende Verbindungsanfragen werden in eine Warteschlange gereiht. Mit .AcceptSocket() oder .AcceptTCPClient() wird die erste Verbindungsanfrage aus der Warteschlange genommen und kann dann an einen TCPClient weitergegeben werden. Die .AcceptSocket() und .AcceptTcpClient() Methoden blockieren jedoch und warten bis auch wirklich eine Verbindungsanfrage kommt. Diese beiden Methoden in einem Thread auslagern macht durchaus Sinn. Wenn man jedoch keinen Thread benutzen will kann man auch einfach mit der .Pending() Methode abfragen ob in Warteschlange etwas eingereiht ist. Mit der .Stop() Methode kann der TCPListener gestoppt werden. Die .Stop() Methode jedoch schließt nicht alle vorhandenen TCPClient Verbindungen, diese danach noch sauber zu schließen liegt in der Hand des Programmierers.

3.4.3.2 Konstruktoren

Name: **TcpListener(Int32)**
Info: Erstellt einen TCPListener welcher auf dem, als Int32 angegebenen, Port horcht.

Name: **TcpListener(IPAdress, Int32)**
Info: Erstellt einen TCPListener welcher auf der lokalen, als IPAdress Objekt übergebenen,

IP-Adresse und dem, als Int32 angegebenen, Port horcht.

Name: **TcpListener(IPEndpoint)**
Info: Erstellt einen TCPListener welcher auf dem angegebenen IPEndpoint horcht.

3.4.3.3 Methoden

Name: **AcceptSocket()**
Info: Gibt den ersten Socket in der Request-Warteschlange zurück. Blockiert den Thread, in welchem die Methode aufgerufen wurde.

Name: **AcceptSocketAsync()**
Info: Gibt den ersten Socket in der Request-Warteschlange zurück. Wird asynchron ausgeführt.

Name: **AcceptTcpClient()**
Info: Gibt den ersten TcpClient in der Request-Warteschlange zurück. Blockiert den Thread, in welchem die Methode aufgerufen wurde.

Name: **AcceptTcpClientAsnyc()**
Info: Gibt den ersten TcpClient in der Request-Warteschlange zurück. Wird asynchron ausgeführt.

Name: **Start()**
Info: Der TcpListener startet und wartet ab jetzt auf eingehende Verbindungsanfragen.

Name: **Stop()**
Info: Der TcpListener stoppt und horcht nicht mehr auf eingehende Verbindungsanfragen. Wichtig ist hierbei jedoch, dass durch Stop nicht die TcpClient Verbindungen sauber geschlossen werden.

3.4.3.4 Implementierung in Ludimus

In der PC/Laptop App wird beim Starten ein TCPListener gestartet welcher auf die eingehenden Verbindungsanfragen der Smartphones, auf welchen ein TCPClient läuft, horcht. Der TCPListener nimmt maximal acht Verbindungen an. Er wird in einem extra Thread geöffnet welcher permanent im Hintergrund läuft, somit wird, bis acht Verbindungen erreicht wurden, permanent auf Verbindungsanfragen geachtet. Wenn eine Verbindungsanfrage ankommt wird sie mit hilfe der .AcceptTcpClient() Methode angenommen und einem Player-Objekt übergeben.

3.4.4 Network Streams

3.4.4.1 Übersicht

Network Ressourcen werden im .Net Framework als Streams angesehen, diese heißen dann Network Streams. Eigenschaften von Network Stream sind unter anderem, dass sie egal welchen Datentyp sie senden oder lesen, dies mit den .Write() und .Read() Methoden möglich ist. Außerdem sind sie sehr ähnlich wie die anderen im .Net Framework oft benutzten Streams sind. Der einzige unterschied ist das die Network Streams nicht suchbar sind. Die .Seek() und .Position() Methoden werfen bei Network Streams eine NotSupportedException. Network Streams sind ein Teil des System.Net.Sockets Namespaces. Die Network Stream Klasse bietet Methoden für das Senden und Empfangen von Daten, dabei hat die Network Stream Klasse Methoden für asynchrones und synchrones Senden und Empfangen. Um einen Network Stream zu bekommen muss bereits vorher eine Socket oder TCP Verbindung zwischen den beiden kommunizierenden Geräten vorhanden sein. Wichtig hier ist aber, dass wenn man den Network Stream schließt nicht automatisch auch die Socket oder TCP Verbindung schließt. Um das ganze synchron zu machen reichen die oben bereits angesprochenen .Write() und .Read() Methoden welche komplett ausreichen jedoch den jeweiligen Thread blockieren. Um das ganze asynchron zu machen sollte man die .BeginRead()/.EndRead() und .BeginWrite()/.EndWrite Methoden benutzen. Diese geben ein IAsyncResult zurück und sind damit dafür besser geeignet. Solange man jedoch einen eigenen Thread für die .Write() und für die .Read() Methode hat können beide synchron auf dem Network Stream eingesetzt werden.

3.4.4.2 Methoden

Name: **Close(Int32)**
Info: Schließt dem Network Stream nach einer, durch Parameter angegebenen, Zeit in welcher noch Daten über den Network Stream gesendet werden.

Name: **CopyTo(Stream,Int32)**
Info: Kopiert den gesamten Network Stream in einen anderen Stream, welcher durch die Parameter angegeben wird. Zusätzlich kann, in Form eines Int32 auch eine Buffergröße angegeben werden, diese dient zur Beschränkung des kopierten Bereichs.

Name: **Dispose()**
Info: Gibt alle vom Network Stream benutzten Methoden frei.

Name: **Flush()**
Info: Löscht die kompletten Daten des Streams.

Name: **Read(Byte[],Int32,Int32)**
Info: Lest den Network Stream. Das Byte Array gibt dabei die den Buffer an aus in welchem die empfangen Daten geschrieben werden. Der zweite Parameter gibt an, an welcher Stelle angefangen wird in den Buffer zu schreiben und der dritte Parameter gibt

an wie viel eingelesen werden soll.

Name: **Write(Byte[],Int32,Int32)**
Info: Schreibt in den Network Stream. Das Byte Array gibt dabei die den Buffer an welcher die Daten speichert die versendet werden. Der zweite Parameter gibt an, an welcher Stelle des Buffers angefangen wird zu senden und der dritte Parameter gibt an wie viel vom Buffer versendet werden soll.

3.5 Sonstige Technologien

3.5.1 Figma (DM)

Figma ist ein Online Mockup Tool zum schnellen Erstellen von Userinterfaces. Entwürfe werden über die Cloud synchronisiert und können von mehreren Benutzern kollaborativ verwendet werden. Durch die Verfügbarkeit im Browser und das ermöglichen von Gruppenarbeiten, entschieden wir uns für Figma anstatt für Illustrator, welches wir noch zu Beginn benutzten.

3.5.2 Blender

Blender ist ein Open Source 3D-Modellierungstool. Es ist gratis nutzbar und bietet neben der Modellerstellung noch viele weitere wichtige Features, wie Animationen, Materialerstellung oder die Kolorierung von Objekten. Durch die in Punkt 3.1.9.1 angesprochene Synergie, zwischen Blender und Unity, ist der Importprozess reibungslos.

3.5.3 Vectr (DM)

Vectr ist ein einfaches Tool zum Erstellen von Vector Grafiken. Die Funktionalität ist zwar auf ein paar Formen und Textfelder beschränkt, jedoch war dies ausreichend für all unsere Icons, weshalb wir von Photoshop zu Vectr wechselten.

3.5.4 Illustrator (DM)

Zu Beginn des Projektes benutzten wir Illustrator noch für die Mockup Erstellung. Wie in Punkt 3.5.1 erwähnt, änderte sich dies jedoch und Illustrator wurde nur für die Erstellung des Billiardtisches, aufgrund der Mustererstellung für das Tuch, die Holzränder und die Metallecken, genutzt.

3.5.5 Delegates (ES)

3.5.5.1 Übersicht

Ein Delegat ist ein Typ, der Verweise auf Methoden mit einer bestimmten Parameterliste und dem Rückgabetyp darstellt. Nach Instanziierung eines Delegaten können Sie die Instanz mit einer beliebigen Methode verknüpfen, die eine kompatible Signatur und einen kompatiblen Rückgabetyp aufweist. Sie können die Methode über die Delegatinstanz aufrufen. Delegaten werden verwendet, um Methoden als Argumente an anderen Methoden zu übergeben .Ereignishandler sind nichts weiter als Methoden, die durch Delegaten aufgerufen werden.

3.5.5.2 Anwendung

Um ein Delegate nutzen zu könnenen muss zuerst ein delegate erzeugt werden:

```
public delegate string DoSth(string name, int y);
```

Beim erstellen des Delegates wird nach dem Keyword “delegate” der Rückgabewert der Methode bestimmt, in diesem Fall wird String zurückgegeben. Darauf folgt der Name, in unserem Fall “DoSth”, danach werden die Parameter welche die Methode benötigt noch angegeben und fertig ist das Delegate. Um dieses jedoch auch benutzen zu können muss es noch instanziiert werden. Dies kann man mit: public DoSth doSthHandler;. Nun kann das Delegate erst richtig benutzen. Nun kann dem Delegate eine Methode welche der obigen ovn der Signatur gleich hinzugefügt werden. Dies kann ganz einfach mit DoSth += myMethod; geschrieben werden. Wurde die Methode zum Delegate hinzugefügt kann man sie und alle anderen Methoden auf dem delegate mit: this.DoSth(“Parameter 1”, Parameter 2); aufrufen.

3.5.5.3 Implementierung in Ludimus

Bei Ludimus werden solche Delegates benutzt um Schnittstellen nach außen bereits zustellen. Diese Schnittstellen können benutzt werden um zum Beispiel eine Methode zum Player InputHandler hinzufügen und so aufgrund des gesendeten Key-Value Pair, des Smartphones, kann man selbst Befehle ausführen und zum Beispiel einen Charakter bewegen. Zusätzlich bietet die Nutzung von Delegates den großen Vorteil das man den Input auf jedes Spiel anpassen kann und somit Keywords, wie “move”, welche von mehreren Spielen verwendet werden öfters zu verwenden kann, da die Abfrage auf diese nicht von der Grundplattform durchgeführt wird sondern vom jeweiligen Spielscript.

3.5.6 ZXing.Net (ES)

3.5.6.1 Übersicht

Um einen QR-Code in C Sharp zu erzeugen wurde die ZXing.Net Library benutzt, welche die Entschlüsselung und Erzeugung von Barcodes unterstützt. ZXing.Net ist der Port der Java basierenden Library ZXing welche ebenfalls für Barcodes benutzt wird.

ZXing.Net unterstütz eine Vielzahl von Barcode-Entschlüsselungen, um genau zu sein Unterstützt es: UPC-A, UPC-E, EAN-8, EAN-13, Code 39, Code 93, Code 128, ITF, Codabar, MSI, RSS-14 (all variants), QR Code, Data Matrix, Aztec and PDF-417.

Der Barcode-Erzeuger unterstützt: UPC-A, EAN-8, EAN-13, Code 39, Code 128, ITF, Codabar, Plessey, MSI, QR Code, PDF-417, Aztec, Data Matrix

3.5.6.2 Einbindung

Grundsätzlich kann man die Library als NuGet Package downloaden. Dies ist jedoch in Unity nicht möglich. In Unity wird das ganze mithilfe einer .dll File gemacht welche in den Plugins-Folder in Unity gezogen wird. Schon kann man sie in den C Sharp Scripts benutzten.

3.5.6.3 Warum ZXing.Net

Die XZing.Net Library beitet eine Einbindung in Unity an und sie ist in C Sharp geschrieben. Sie ist somit die einzige QR-Code Library welche es kostenfrei ermöglicht QR-Codes, in C Sharp zu erstellen und zu entschlüsseln. Einziger Negativpunkt ist die teils mangelnde Dokumentation.

3.5.6.4 Usage

3.5.6.4.1 Erzeugung In Unity muss man zur Benutzung der Library eine WebCamTexture instanzieren. Diese kann man mit ".Play()" und ".Stop()" starten und anhalten. Um nun aus der WebCamTexture Pixel auszulesen wird die Methode ".GetPixels32()" angewandt. Diese lieftert ein Color32 Array zurück. Mit diesem Array selbst fängt man wenig an, da es einfach nur ein Array von Farbwerten ist. Um mit dem Array arbeiten zu können muss zusätzlich noch einen IBarcodeReader instanziert werden. Dies geht mithilfe der "BarcodeReader()" Methode welche ZXing.Net beinhaltet und eben genau einen IBarcodeReader zurück gibt. Zum Entschlüsseln des Barcodes wird einfach die ".Decode(Color32 Array, WebCamTexture Höhe, WebCamTexture Breite)" Methode benutzt welche einen String, welcher den entschlüsselten QR-Code als Text beinhaltet, zurückliefert. Dieser wird das Color32 Array welches vorher erzeugt wurde als erster Parameter mitgegeben. Zusätzlich benötigt die Methode noch die Höhe und Breite der bereits instanzierten WebCamTexture. Dies geht dank den Properties der WebCamTexture ".height" und ".width" ganz einfach. Um herauszufinden ob ein Barcode gefunden wurde wird einfach das Ergebnis der Decode Methode mit null verglichen, da es leer ist wenn kein Code gefunden wird.

3.5.6.4.2 Entschlüsselung Bei dem Erzeugen von Barcodes wird zunächst eine leere 2DTexture erzeugen, diese dient als an Anzeigefläche für den QR-Code. Darauf wird nun mit hilfe eines BarcodesWriters, welchem das Format "BarcodeFormat.QR_CODE" zugewiesen wird und als Optionen die Höhe und Breite der 2DTexture mitgegeben werden. Mithilfe der ".Write(Text)" Methode bekommt man ein Color32 Array zurück, welches nun auf die 2DTexture geschrieben wird. Nun wird diese noch im Frontend angezeigt und schon hat man einen QR-Code erzeugt.

3.5.6.5 Implementierung in Ludimus

Bei der Implementierung in Unity gab es ein gewaltiges Problem. Die Kamera war während der Anzeige im Frontend, wegen der Berechnung der ZXing.Net Library, ob ein Code gefunden wurde, nicht flüssig. Es wurden immer nur im Sekundenabstand Frames angezeigt. Auch wenn das reicht um den QR-Code zu scannen war es bei weitem nicht genug um auch benutzbar zu sein. Die Lösung war einfach einen Kamerastream zu zeigen mit welchem nichts berechnet wurde und welcher somit flüssig war.Dies jedoch klingt leichter als gedacht, denn in man kann in jeder Unity-Scene nur eine WebCamTexture

am laufen haben. Als Endlösung wird in der Unity-Scene bereits eine WebCamTexture gestartet und dann die Berechnung des QR-Codes, mit einer neuen WebCam

Kapitel 4

How to use (ES)

4.1 Vor dem Start

Da die Ludimus-App ihre Verbindungen innerhalb des lokalen Netzwerks aufbaut, ist es vorausgesetzt das alle Geräte auf denen der User Ludimus benutzten möchte in einem lokalen Netzwerk hat. Dies kann erreicht werden in dem man die Endgeräte alle mit einem W-Lan Netzwerk verbindet oder indem man sich einen Hotspot auf dem Smartphone erstellt und die Geräte damit verbindet. Nun muss auf allen Geräten welche zum Spielen benutzt werden möchten die Ludimus-App installiert sein und schon kann man loslegen.

4.2 Computer-App

Wenn alle generellen Bedingungen erfüllt sind kann man die Ludimus-App auf dem PC/Laptop starten. Nach einer kurzen Ladezeit wird dem User ein Lobbycode und ein QR-Code angezeigt. Diese werden für die Verbindung mit dem Smartphone wichtig und da das Smartphone als Fernbedienung für die App fungiert kann der User sich nun zurücklehnen und bequem Ludimus bedienen.

4.3 Android/IOS - App

4.3.1 Vor dem Einloggen

Beim Starten der Ludimus-App öffnet sich zunächst die Login Maske in welcher man gebeten wird sich einzuloggen. Wenn der User sich bereits einen Account erstellt hat muss er nur seine Daten in der Maske ausfüllen und auf den Login Button drücken um sofort erfolgreich eingeloggt zu sein. Wenn man nun noch keinen Account hat, hat der User zwei Möglichkeiten, er kann sich einen neuen Account erstellen oder sich als Guest anmelden. Wenn er sich dafür entscheidet als Guest sich einzuloggen hat der User keine Spiele selbst zur Verfügung und kann nur Spielern mit bezahltem Account beitreten, in den Spielen selbst gibt es jedoch keinen Unterschied zwischen Guest und zahlenden Account. Wenn der User sich einen Account erstellen will muss er nur auf den Sign-Up Button drücken und es öffnet sich eine neue Maske in welcher der User seine E-Mail und sein Passwort, welches er zur Sicherheit zweimal angeben muss, eingeben kann. Zum Erstellen des Accounts muss er nun erneut auf den, nun rechts platzierten, Sign-Up Button drücken. Der Account wird automatisch in der Firebase-Datenbank gespeichert und man wird sofort eingeloggt.

4.3.2 Nach dem Einloggen

Wenn der Spieler nun eingeloggt ist kann er einen Name wählen und kann einen Lobbycode eingeben, dieser wird in der gestarteten Ludimus-App für den PC/Laptop angezeigt. Alternativ kann man jedoch auch den QR-Code, welcher ebenfalls den Lobbycode beinhaltet und welcher ebenfalls in der Ludimus-App für PC/Laptop angezeigt wird, scannen und somit mit noch weniger Arbeit sich zum Server verbinden. Zum Scannen des QR-Codes einfach auf den Button rechts unten drücken, der Button sieht aus wie

ein QR-Code. Wenn man nun beide Felder ausgefüllt hat kann man auf das W-Lan Symbol drücken um sich zu dem Server zu verbinden. Bei erfolgreicher Verbindung gibts es nun zwei Möglichkeiten. Erstens, sie sind als erster Spieler verbunden. Sie sind nun der Admin dieser Lobby und können Spiele starten und downloaden. Zum Starten eines Spiels einfach auf das Symbol im Hauptmenü drücken und schon lädt das Spiel auf alle Smartphones und auf dem PC/Laptop. Wenn sie jedoch nicht der Admin, der Lobby sind können sie nun nur warten bis der Admin ein Spiel startet. Wenn er dies tut laden sich alle Dateien automatisch auf ihr Handy und sie können, nach einer kurzen Wartezeit, auch schon loslegen und spielen.

Kapitel 5

Ausgewählte Aspekte der Systemerstellung

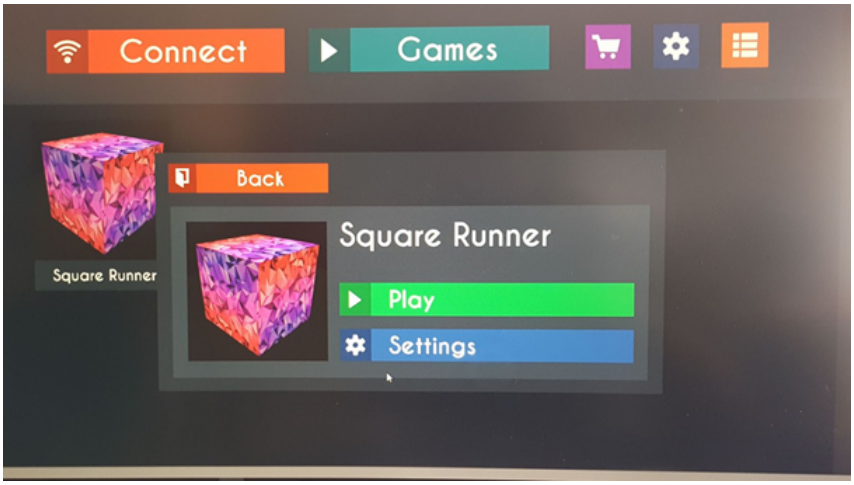

Abbildung 5.1: Erster Shop Prototyp

5.1 Designentscheidungen (DM)

Ludimus war von Beginn an als ein Produkt für Konsumenten gedacht, weshalb wir viele Überlegungen und anschließende Änderungen bezüglich der Benutzeroberfläche anstellten. Viele der Abwandlungen sind mit dem Hintergedanken der Benutzbarkeit der Plattform verbunden und konzentrierten sich auf Farben, sowie Form und Anordnung. Im Folgenden Abschnitt werden generelle Designentscheidungen beschrieben, die sich auf alle Aspekte es Projekts ausgewirkt haben. Änderungen die nur einen Teilbereich umfassten, werden in Punkt 5.1.2 genauer erwähnt.

5.1.1 Generelle Designentscheidungen

Für die ersten Iterationen des Projekts wählten wir dunkle Farben um auch das Spielen in den späteren Stunden zu ermöglichen. Die Grafik 5.1 zeigt das Server Interface, wobei hier das Shop-Interface zu sehen ist, sowie die Detailansicht eines Spieles.

Um einerseits mehr Platz für den eigentlichen Inhalt zu haben und andererseits moderner auszusehen, änderten wir die Proportionen, die Abstände und Teils die Farben, um an die Kacheln von Windows 8, zu erinnern, wie in der Abbildung 5.2 zu sehen. Hier ist die Lobbyansicht ausgewählt.

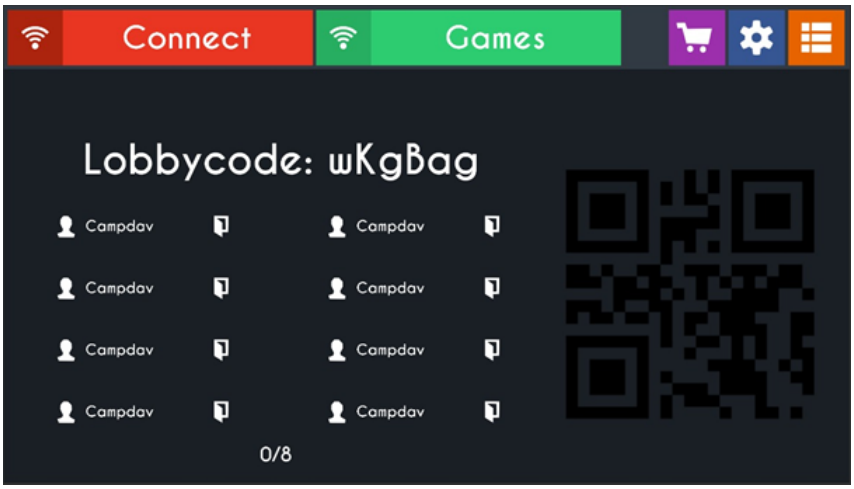

Abbildung 5.2: Lobby Prototyp

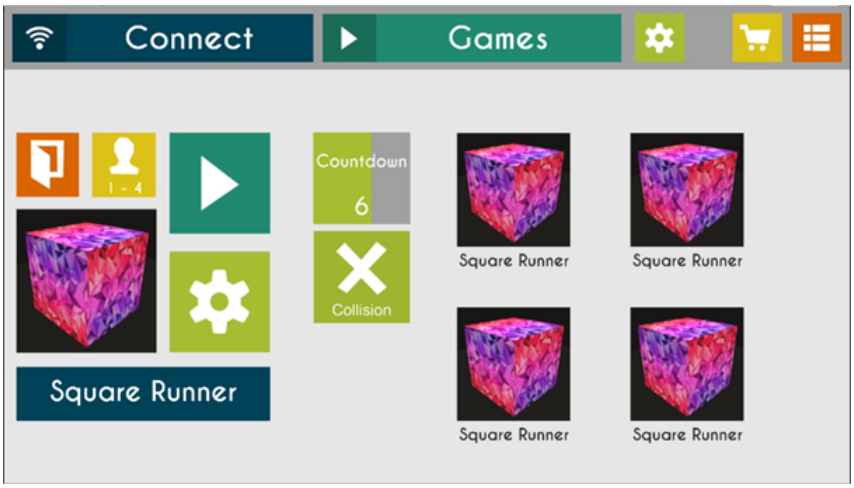

Abbildung 5.3: Shopdetailansicht mit Einstellungen und hellem Farbschema

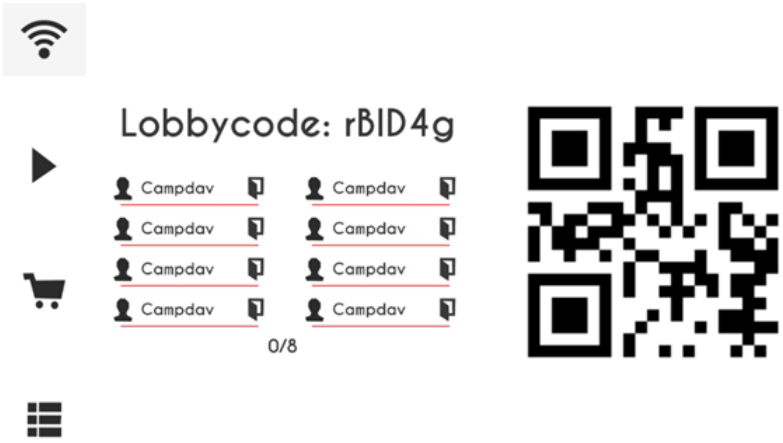

Abbildung 5.4: Schwarz-Weiß Designentwurf

Ludimus war Primär als Produkt für Familien mit Kindern gedacht. Dunkle Farben übermittelten nicht das Gefühl, das wir transportieren wollten, weshalb wir uns für helle Farben auf weißen Hintergrund entschieden. 5.3 zeigt auch eine Umgestaltung der Detailansicht, dazu jedoch in Punkt 5.1.2.1 mehr.

Einen gewissen Rückschritt in Sachen Farben stellte unser nächstes UI dar, zu sehen in 5.4. Der Hintergedanke hierbei war die Vereinfachung der Plattform und ein schwarzweißes Design half uns dabei. Nutzer sahen nun auch erstmals, in welchem Menü sie sich befanden, da dieses nun einen dunkleren Hintergrund bekam. Globale Einstellungen für die Plattform selbst als eigener Punkt, wurden mit diesem Design entfernt.

In den nächsten Iterationen brachten wir einerseits Farbe zurück, rundeten andererseits aber auch alle Buttons ab und gaben ihnen einen Schatten, wie in 5.5 zu sehen. Hatte man in allen bisherigen Versionen noch eine Menüleiste und einen Inhaltsteil, so entfernten wir Erstere um nur den Inhalt darstellen zu können. Drückt der Nutzer nun auf einen der vier Menüpunkte, so schwenkt die Kamera in dessen Ecke und der Inhalt wird dort angezeigt. Mit einem Zurück-Knopf kann der Benutzer anschließend wieder zum Hauptmenü gelangen. Wir implementierten zusätzlich noch eine Farbauswahl für das gesamte Interface. Benutzer konnten sich bestimmte Farbkombinationen auswählen, um sowohl die vier Primärfarben, als auch den Hintergrund zu ändern. Diese Entscheidungen hatten jedoch alle mehrere Nachteile, weshalb wir alles wieder verändern mussten. Die Kameraanimationen waren zwar interessant anzusehen, jedoch verlangsamten sie den

Abbildung 5.5: Buntes Hauptmenü mit Schatten und abgerundeten Ecken

Navigationsprozess Unnötigerweise. Zusätzliche Probleme machten verschiedene Formate, da die Animationen nur auf 16:9 optimiert waren. Obwohl das Farbschema wechseln gut funktioniert hat und Menschen mit verschiedenen Präferenzen die Möglichkeit gab das UI zu verändern, mussten wir das Feature entfernen, da uns die Erkennung unserer Marke wichtig war und wir erreichen wollten, dass sobald Familien unser Logo oder unsere Farben sehen an uns denken können. Wenn jeder seine eigenen Farben festlegen kann, ist dies nicht möglich.
Bisher waren eine Übersicht der Lobby, der Shop, alle Heruntergeladenen Spiele und ein Menü für Optionen in der Serveransicht und der Client konnte nur Name und Lobbycode eingeben und sich anschließend verbinden. Wollte eine Gruppe also spielen, mussten sich zuerst alle Spieler auf deren Handys zum großen Bildschirm verbinden und danach musste ein Spieler das Tablet, den Laptop oder den Fernseher bedienen um ein Spiel zu starten, oder zuerst herunterladen. Speziell beim Spielen im Wohnzimmer über den Fernseher gibt es hier enorme Probleme, die mit der Steuerung von Menüs mit der Fernbedienung zu tun haben. Um diese unnötige Interaktion mit dem großen Bildschirm zu entfernen, gestalteten wir die Aufteilung der Menüs komplett um. Der Server sollte nur noch die Lobby anzeigen, da diese keine Interaktion forderte. Sowohl Spielernamen, als auch der Lobbycode in schriftlicher und grafischer Form, konnten so größer dargestellt werden. Zusätzlich zu einem Login Fenster, welches wir bei App-Start dem Client hinzugefügt haben, wurden Spieler wie immer aufgefordert ihren Namen und den Lobbycode einzugeben um sich anschließend zu verbinden. Dieser Vorgang blieb für alle Spieler bis auf den ersten gleich. Dieser hat nun jedoch die Kontrolle über die Lobby und kann sowohl neue Spiele herunterladen, als auch bereits heruntergeladene Spiele starten. Um das Interface zu vereinfachen, vereinten wir den Shop mit den heruntergeladenen Spielen. Durch den Verzicht einer einheitlichen Trennung kann der Nutzer nicht unterscheiden ob

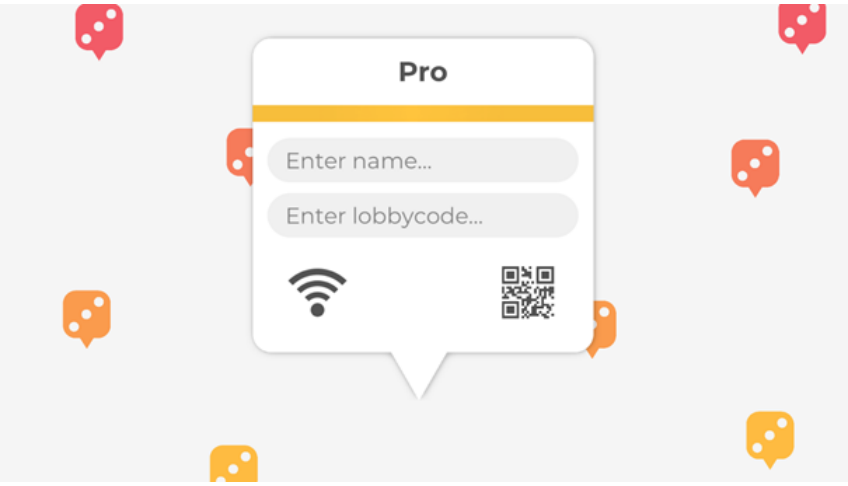

Abbildung 5.6: UI zum Verbinden zu einer Lobby

ein Spiel zuerst heruntergeladen werden muss und probiert so viel mehr neue Spiele. Die Downloadzeit überschreitet nur selten 10 Sekunden, wodurch dies nur noch verstärkt wird.

Wie in 5.6 zu sehen, ist unser Userinterface mit dem Ludimus Logo eng verknüpft, um es so in die Köpfe der Nutzer zu bekommen. Das UI zum Verbinden zu einer Lobby ist in eine Form, die dem Ludimus Logo ähnelt, eingebettet. Zusätzlich steigen im Hintergrund Ludimus Ballone auf, die ihre Farbe in Abhängigkeit des Subskription Levels, des eingeloggten Nutzers, ändern.

5.1.2 Spezielle Designentscheidungen

5.1.2.1 Shop

Der Shop änderte sich über die verschiedenen Versionen am meisten, da die richtige Präsentation der Spiele essentiell für den Kauf eines teureren Subskription Plans ist. Die Änderungen gliedern sich in die Anzeige aller Spiele und die Anzeige eines Spieles.
Zu Beginn wurden alle Spiele in einem Gitter mit Zeilen und Spalten angezeigt, wie in 5.1 zu sehen. Überlegungen bezüglich Sortierung oder Suche wurden zu diesem Zeitpunkt noch nicht angestellt. Um Nutzern zu ermöglichen nur Spiele einer bestimmten Art zu sehen, erweiterten wir deren Datenmodell um das Feld „Genre“. Zusätzlich vergaben wir einen Farbcode für jedes Genre, um dem Nutzer schnell ersichtlich zu machen um welche Art von Spiel es sich handelt. Da wir die Plattform für externe Entwickler öffnen wollten, um mehr Spiele anbieten zu können, fügten wir die Reiter „New“, „Top“ und „New Update“ hinzu. Der Gedanke hierbei war neue Spiele eine kurze Zeit unter „New“

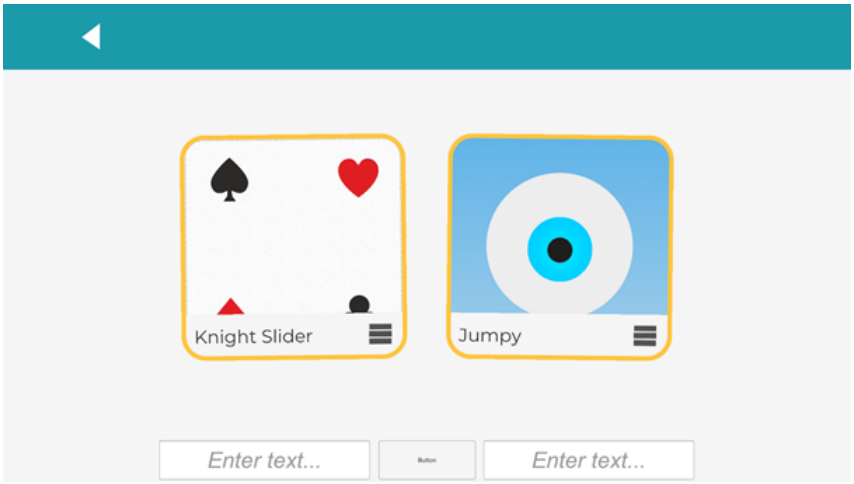

Abbildung 5.7: Finale Shoppräsentation mit Debug Modus

anzuzeigen und Spiele, die oft gespielt werden sollten anschließend unter „Top“ zu finden sein. Um auch ältere Spiele wieder beliebt zu machen oder Entwicklern eine zweite Chance geben zu können, falls der ursprüngliche Release nicht funktioniert hat, würden Spiele die neue Updates erhalten unter „New Update“ gezeigt werden. Den Plan externe Entwickler mit einzubeziehen verworfen wir jedoch, wodurch eine solche Gliederung nicht mehr nötig war, da zu wenig Spiele dafür verfügbar wären. Wie in Abbildung 5.7 zu sehen, vereinfachten wir zusätzlich die Anzeige der Spiele, in dem wir das Icon des Spieles größer machten, die Detailansicht hinter einem Menü versteckten und wir das Genre des Spieles durch die Farbe der Umrandung klar darstellten. Die zwei Eingabefelder, am unteren Ende des Bildschirms, mit dem Knopf in der Mitte gehören zum Debug Modus, der nur für Entwickler ist, dazu mehr in 5.3.

Die in Abbildung 5.3 zu sehenden spielspezifischen Einstellungen wurden entfernt und durch Optionenmenüs im Spiel selbst ersetzt. Durch Änderungen im Startprozess von Spielen wurden Spiele mit mehreren Szenen ermöglicht, wodurch die alte Methode überflüssig wurde.

5.2 Spielstart (DM)

Sobald der erste verbundene Spieler ein Spiel ausgesucht hat, indem er darauf geklickt hat, und die Gruppengröße für das Spiel in Ordnung ist, wird die ID des Spieles an den Server geschickt. Dieser sucht sich sowohl die Downloadlinks für die Client-Dateien, als auch für seine eigenen Server-Dateien. Anschließend überprüft er, ob das Spiel schon alle nötigen Dateien für die aktuelle Version heruntergeladen hat. Falls nicht wird das nachgeholt und anschließend werden die Download URLs für die Clients an alle verbundenen Spieler geschickt. Diese überprüfen auch deren Existenz und laden fehlende herunter. Verfügt ein Client über alle Dateien, so schickt er ein „ok“ an den Server. Sobald der Server Bestätigungen aller Clients erhalten hat, startet er seine Szene. Und schickt ein „ok“ an alle Clients, worauf hin diese auch deren Szenen starten. Dieser Ablauf ähnelt dem zwei Phasen Commit bei verteilten Datenbanken, wodurch die Sicherheit dieses Systems bewiesen ist.

5.2.1 Probleme

Spiele bestehen drei Komponenten, die gemeinsam dafür sorgen, dass die Spiele funktionieren. Sie bestehen aus Szenen, Modellen und Scripts. Szenen und Modelle, auch Texturen, Materialien, usw. sind in sogenannten „Assetbundles“ gespeichert. Alle Szenen des Clients sind in so einem „Assetbundle“ zusammengespeichert, ähnlich einer Zip-File. Alle Szenen des Server sind ebenfalls so persistiert. Die Unterscheidung zwischen Client und Server ist zwar nicht zwingend nötig, jedoch ist es Ressourcenverschwendung, wenn der Server auch Szenen der Clients speichert, da er diese ohnehin nicht startet, und umgekehrt. Die selbe Speicherung gilt für alle Modelle. Der Grund für die Trennung zwischen Client und Server bleibt gleich, jedoch wäre es deutlich einfacher, wenn Clientszenen und Clientmodelle zusammen gespeichert werden und alle Serverszenen und Servermodelle ebenfalls. Leider unterstützen „Assetbundles“ in Unity diese Funktionalität nicht. Als dritter Baustein gelten die Scripts, in denen die Logik der Spiele, das eigentliche Programm, beschrieben ist. Scripts können nicht zu „Assetbundles“ gemacht werden, jedoch gibt es unter Android und Windows die Möglichkeit, sie zu bündeln und dann zur Laufzeit eines Spieles zu verwenden. Unter Ios ist eine derartige Methode nicht möglich, da kein externer Code zur Laufzeit von Apps hinzugefügt werden kann oder darf. Diesen Sicherheitsmechanismus wird Apple nicht entfernen, was bedeutet, dass jeder Code immer in der App vorhanden sein muss. Für den Workflow des Erstellens eines neuen Spieles würde das bedeuten, dass wir alle verwendeten Scripts in das Hauptprojekt transferieren müssen und dann eine neue App Version in den App Store hochladen müssen. Apple User sind ein großer Anteil an unserer Zielgruppe, weswegen ein Ausschluss der Plattform nicht möglich ist. Zwei Speichervarianten für zwei Plattformen, wäre undenkbarer Mehraufwand, weshalb wir uns entschlossen, dass wir für jedes neue Spiel ein Update mit dessen Scripts für alle Plattformen herausbringen.

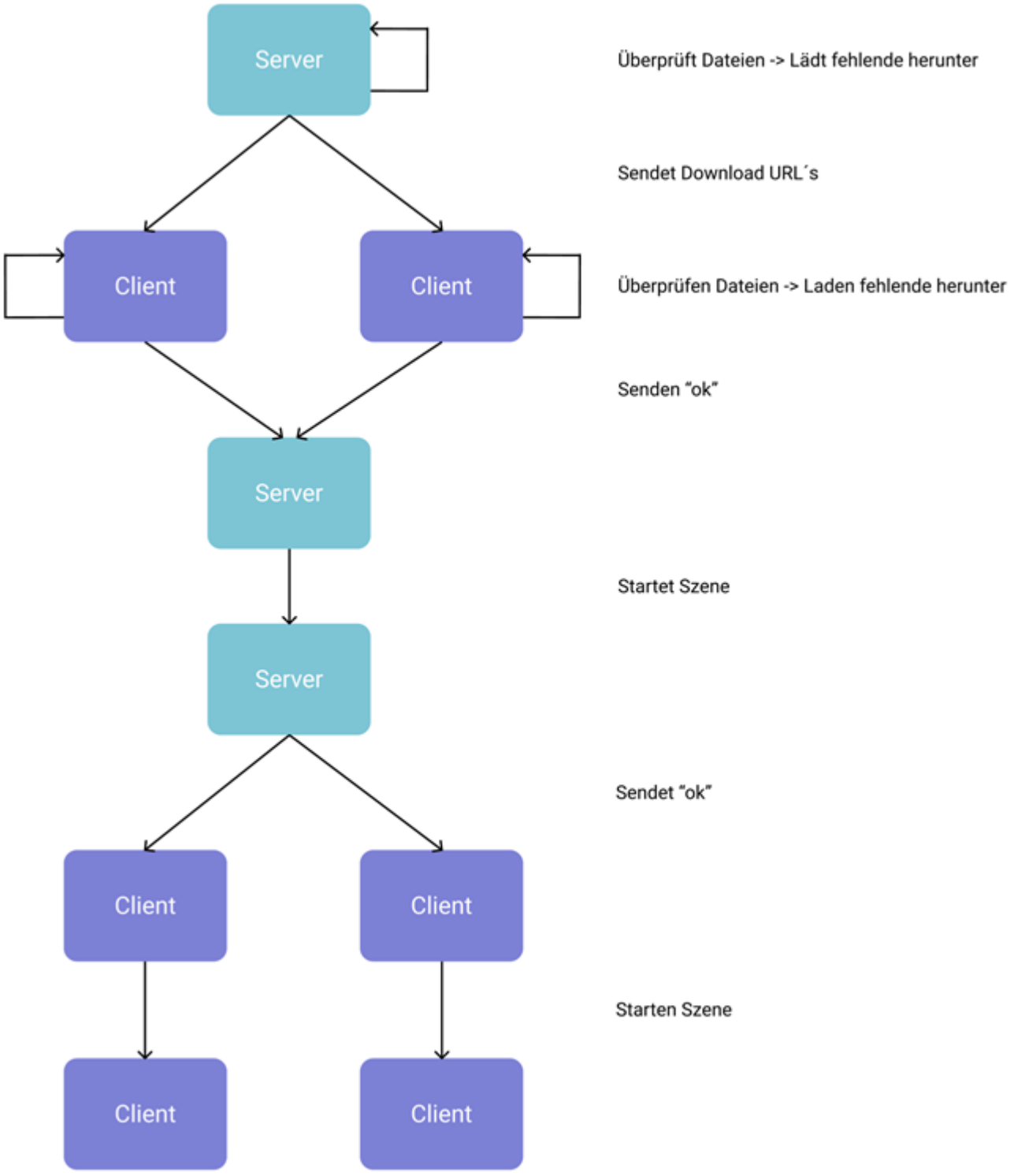

Abbildung 5.8: Vorgang bei Spielstart

5.2.2 Optimierungsansätze

Der Fileserver von Firebase hat mit unserem Gratisbezahlplan eine maximale Downloadgröße und eine Grenze für Downloads pro Tag unabhängig von der Größe der Dateien, die heruntergeladen werden. Das ist der Grund, warum nur so viele Zugriffe gemacht werden sollten, wie nötig. Eine Maßnahme war es, dass der Server alle Download URLs beschafft und an die Clients weitergibt. Dieses Prinzip wurde auch für den eigentlichen Download selbst getestet. Im konkreten bedeutet das, dass der Server alle Plattformen der Clients überprüft und dann die Clientdateien für eben diese Plattformen herunterlädt, zwischenspeichert und anschließend den Clients über das lokale Netzwerk sendet. Dieser Ansatz entlastet den Fileserver, da dieser nur einen Downloadrequest verarbeiten muss, der Rest passiert nur zwischen den Spielern. Die gesamte Downloadzeit steigt jedoch dadurch, dass Dateien zwei Mal pro Spieler heruntergeladen werden, anstatt einmal und Clients können während dieser Zeit nur eingeschränkt kommunizieren, weshalb wir uns für die oben beschriebene Variante entschieden.

5.3 Debug Modus (DM)

Das in 5.2 beschriebene System funktioniert einwandfrei mit fertigen Spielen. Da jedoch Spiele sowohl Datenbankeinträge, als auch hochgeladene Dateien am Fileserver benötigen, ist diese Lösung komplett ungeeignet für Spiele in Entwicklung. Der Debug Modus löst dieses Problem und ermöglicht es, auch kleine Änderungen, mit erheblich geringerem Zeitaufwand, zu testen. Vor dem Debug Modus, wurden die Scripts der fertigen Spiele in das Hauptprojekt hinein kopiert. Die Entwicklung passierte in einem externen Projekt. Diese externen Projekte hatten ein Gerüst, das sich um die Spielerhandhabung und um die Kommunikation zwischen Client und Server im generellen kümmerte. Ein Testdurchlauf der aus „Assetbundles“ für Szenen und Modelle, jeweils wieder unterschieden in Client und Server, bauen, auf Fileserver hochladen, Scripts in Hauptprojekt kopieren und neuste Version für Smartphone und PC bauen dauerte mehrere Minuten. Mit dem Debug Modus fällt das externe Projekt mit Kommunikationsgerüst, das Bauen der „Assetbundles“ und das hochladen auf den Fileserver weg. Für neue Spiele wird nur ein Ordner im Hauptprojekt und eine Client und Server Startszene angelegt. Die Kommunikation zwischen Server und Client ist mit Delegates abrufbar, wodurch man aktuelle Spieler, deren Inputs und vieles mehr gleich verwenden kann. Bestimmte Aspekte, wie Physics, UI oder Gegner-AI sind Client unabhängig und können somit nun, auch ohne diese, in der Engine, mit einem Knopfdruck, getestet werden. Werden sowohl Client, als auch Server benötigt, die Clients für Smartphone und PC bauen. Der letzte Schritt ist nur bei fast fertigen Versionen nötig, da es sonst eher ratsam ist, nur einen Client zu bauen und den Server in Engine zu debuggen. Als erster verbundener Spieler kann man in der Spielanzeige das neue Spiel zwar nicht sehen, über die zwei Eingabefelder, kann man jedoch sein Spiel nun starten. Im linken Feld wird der Name der Startszene des Client angegeben und im rechten die des Servers. Drückt der Entwickler nun auf den Knopf in der Mitte wird sein Spiel gestartet. Die Zeiteinsparen und die Erleichterung der Entwicklung sind enorm, weshalb alle fertigen Spiele mit dieser Methode entwickelt worden sind.

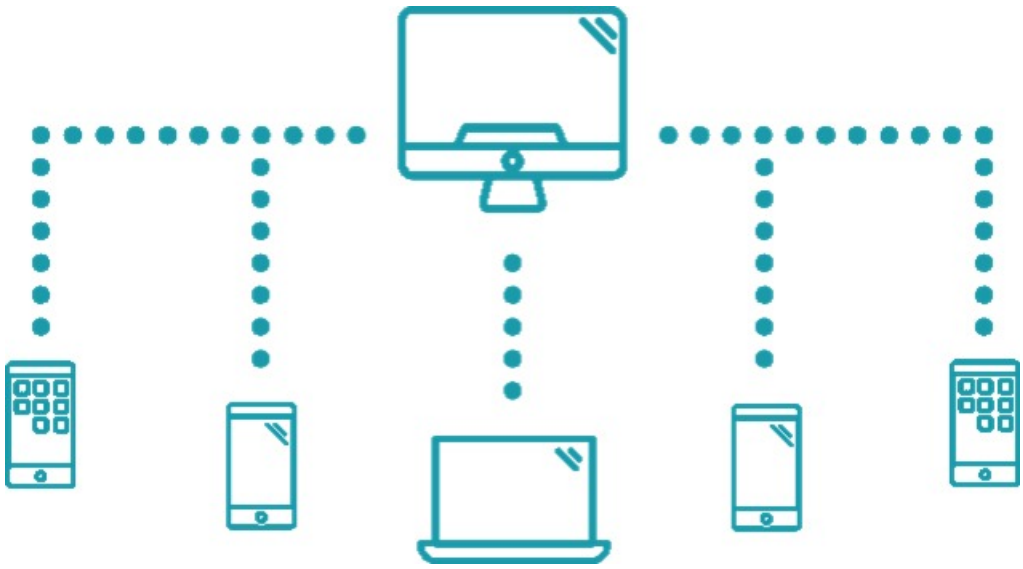

Abbildung 5.9: Aufbau Ludimus

5.4 Aufbau von Ludimus (ES)

5.4.1 Übersicht

Ludimus selbst ist in zwei Applikationen unterteilt. Eine läuft auf einem Gerät welches man als Controller benutzten will, dies kann ein Smartphone oder ein anderes Gerät, auf welchem Android lauft, sein und eine Applikation welche auf einem Laptop, PC, Tablet oder SmartTV läuft. Diese wird ab sofort als Server bezeichnet. Der User muss dann auf seinem Controller einen Code eingeben oder scannen mit welchem sich das Gerät dann zu der Anwendung welche auf dem Server läuft verbindet. Der User bedient nach dem Aufbau der Verbindung die komplette Plattform mit seinem Controller. Die Verbindung ist in der Abbildung 5.10 nochmal deutlicher dargestellt.

5.4.2 Ablaufdiagram Ludimus

5.4.2.1 Erklärung des Ablaufdiagrams(5.10)

5.4.2.1.1 Schritt 1 Im ersten Schritt generiert der Server zunächst einen Lobbycode aufgrund der IP-Adresse des Gerätes. Genauer ist dies im Kapitel Lobbycodes 5.5 beschrieben. Danach öffnet der Server den TCP Listener, dies sieht im Code so aus:
this.listener = new TcpListener(IPAddress.Parse(this.ipAdress), 9393);
this.listener.Start();
Wichtig ist hierbei das der TCPListener auch noch gestartet werden muss nach der Erzeugung. Mehr dazu im Kapitel 3.4.3. Wenn der TCP Listener erfolgreich gestartet wurde kann der Lobbycode auch angezeigt werden, da ab nun sich Spieler verbinden können. Im Code wird das so realisiert:
this.LobbyCodeText.text = Lobbycode: "+ this.playermanager.GetLobbyCode();
Wobei dies in der Initialisierung des UI's und sommit in einer anderen Klasse geschiet. Deshalb wird hier auch die *GetLobbyCode()* Methode des Playermanagers verwendet. *this.LobbyCodeText* ist das TextElement welches in Unity erzeugt wurde. Letzter Teil dieses Schrittes besteht darin den TCP Listener „horchen" zu lassen, damit er auch auf die Anfragen reagieren kann. Code:
var tmpClient = this.listener.AcceptTcpClient();
Dabei ist jedoch zu beachten das this.listener.AcceptTcpClient den Mainthread blockiert und solange wartet bis eine Verbindung kommt. Deshalb wird in Ludimus dies auch in einem eignem Thread ausgeführt.

5.4.2.1.2 Schritt 2 Bei Schritt 2 gibt der User zunächst seinen Name und den Lobbycode in Textfeldern des UI's ein. Dieses sind per 2-Way Binding mit dem Backend verbunden. Wenn der User nun auf den Verbindungs-Button drückt wird der Befehl zum Verbinden ausgeführt, dieser sieht im Code wiefolgt aus
this.client = new TcpClient(DecryptLobbyCode(this.lobbycode), 9393);
Die *DecryptLobbyCode* Methode wird im Kapitel 5.5 genauer besprochen und der TCP-Client wird im Kapitel 3.4.2 genauer besprochen. 9393 ist hierbei der Port welcher bei Ludimus standardisiert 9393 ist.

5.4.2.1.3 Schritt 3 Der im Schritt 1 bereits beschriebende TCP Listener akzeptiert mit *this.listener.AcceptTcpClient();* automatisch eine Verbindung wenn eine Anfrage kommt. Wenn die Verbindung akzeptiert wird, wird zunächst ein Player Objekt erzeugt. Im Code:
var tmpPlayer = new Player();
tmpPlayer.startPlayer(tmpClient, this, this.internalCounter);
Hier wird mit new Player() eben ein neues Objekt erzeugt welchem dann mit der *startPlayer()* Methode alle wichtigen Sachen mitgegebn wird, wie der TCPClient, PlayerManager und Playernumber. In der *startPlayer* Methode wird zusätzlich der Output-Thread geöffnet. Danach wird das Player Objekt in eine Liste aller Spieler eingefügt um immer mit ihm, und allen anderen, zu kommunizieren zu können. Am Schluss dieses Schrit-

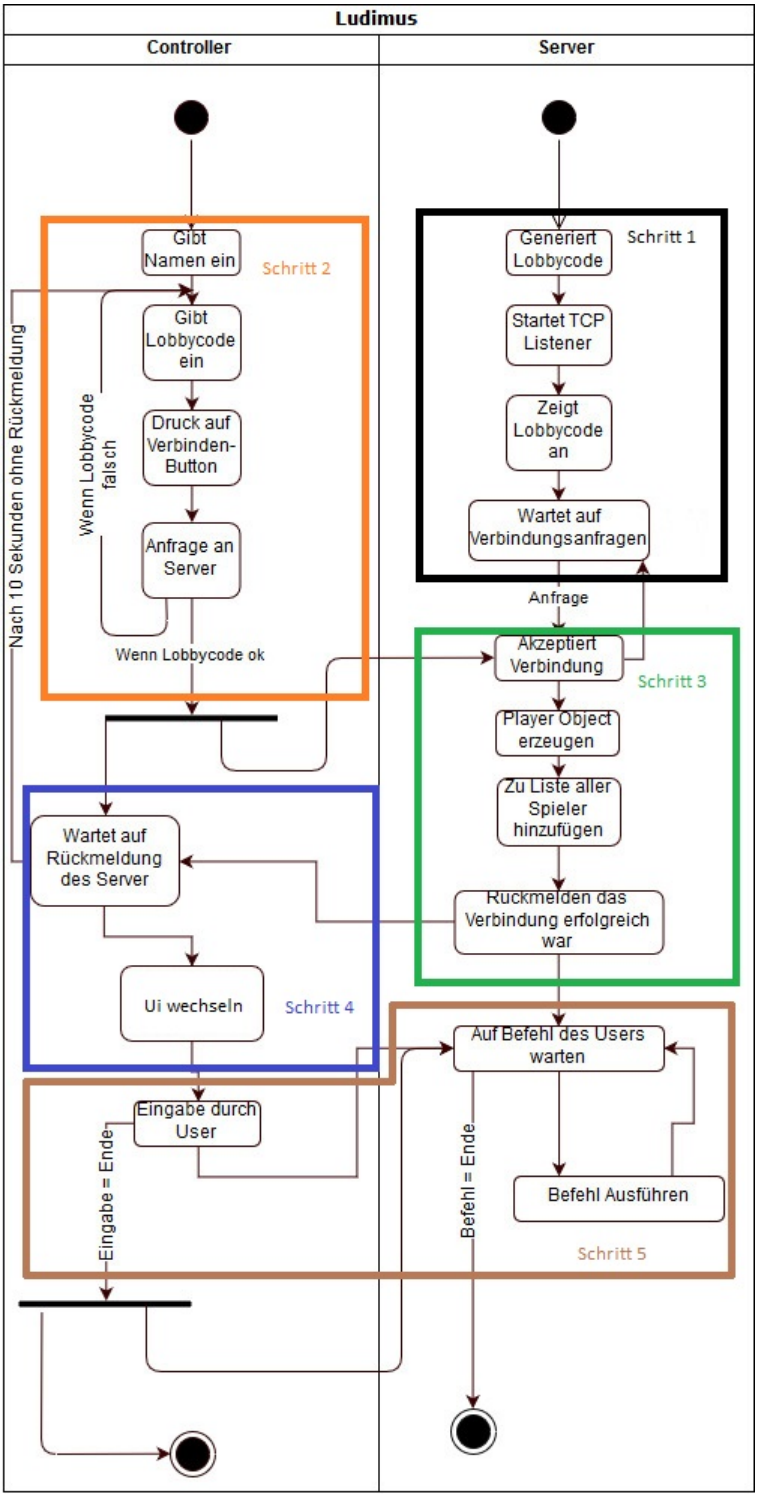

Abbildung 5.10: Ablaufdiagram Ludimus

tes wird dem TCPClient eine Bestätigung geschickt, dass die Verbindung auch wirklich funktioniert hat. Senden von String im Code:
public void SendData(string key, string value){
var dat = new byte[4096];
dat = Encoding.ASCII.GetBytes(key + "+ value + ";");
this.client.GetStream().Write(dat, 0, dat.Length);
this.client.GetStream().Flush();
}
Hier wird zunächst ein Byte-Array der Größe 4096 angelegt. In dieses Byte-Array wird dann der String, welchen wir senden wollen, geschrieben. Die geschiet mit der *Encoding.ASCII.GetBytes()* Methode in welcher als Paramter der String,welchen wir senden wollen, steht. Der String wird der Methode *SendData* als Key und Value mitgegeben. Mit *this.client.GetStream().Write()* kann man nun auf den Networkstream schreiben. Wobei man dieser Methode die zu schriebenden Bytes übergibt und angibt wo angefangen wird zu lesen(da wir vom Anfang lesen wollen 0) und die Länge des zu lesenden Array's angeben werden muss.

5.4.2.1.4 Schritt 4 Wenn der TCPClient, also der Controller, nun dies Besätigung bekommt welchselt er das UI. In dem neuen UI sieht der Spieler den Ludimus-Shop und alle Spiele welche ihm zur Verfügung stehen. Empfangen von Daten im Code:
var data = new byte[4096];
string responseData = string.Empty;
int bytes = stream.Read(data, 0, data.Length);
responseData = Encoding.ASCII.GetString(data, 0, bytes);
Wichtig dazu ist das dies dauerhaft in einem Thread geschiet um den MainThread nicht zu blockieren. Zunächst wird ein neues Byte-Array angelegt welches 4096 Bytes groß ist. In dieses werden nacher die gesendeten Daten eingelesen. Dann muss ein leerer String erzeugt werden welches mit *string.Empty* am cleansten geht. Nun lest man in das *data* Array mithilfe von *stream.Read()* ein. Mitgegebn wird dabei das Array, in welches gelesen wird, ein int, wobei dieser definiert wo im array angefangen wird zu einlesen (da wir bei ersten Stelle des Arrays beginnen wollen 0) und die Länge des Arrays in welches gelesen wird. Mit *Encoding.ASCII.GetString()* kann das Byte-Array dann zu einem String geparst werden. Dieser String wird dann in eine Queue gegeben und im MainThread behandelt 5.6. Das Lesen von Daten, des Networkstreams 3.4.4 geschiet im Server und Controller gleich.

5.4.2.1.5 Schritt 5 Der 5te Schritt besteht aus dem Warten auf Befehlen am Controller und dem Ausführen von Befehlen am Server. Es wird zum Beispiel darauf gewartet das der User durch einen Buttondruck am Controller ein Spiel startet. Passiert dies, wird mithilfe der in Schritt 3 und 4 beschriebenen Datenübertragung dem Server geschickt das er ein Spiel starten soll. Mehr zu Spielstart unter Kapitel 5.2.

5.5 Lobbycodes (ES)

Das Verbinden zur der Laptop/PC Version von Ludimus erfolgt mit sogenannten Lobbycodes. Dieser wird vom Computer aufgrund der IP-Adresse erzeugt und ermöglicht es einem Smartphone Verbindung zum Rechner aufzubauen.

5.5.1 Verbindungsgeschichte

Die Grundart der Verbindung stand von Anfang an fest, Network Streams, aber wie genau soll sich der User nun, möglichst einfach, mit dem Gerät verbinden? Eine Möglichkeit dafür wäre ein Broadcast, bei solchem wird ein Datenpaket an alle, mit dem gleichen Netzwerk verbundenen Geräte geschickt. Bei Ludimus konkret sollte jedes Gerät gefragt werden ob es die Ludimus-App gestartet hat und sich dann bei erfolgreicher Rückmeldung zu der IP-Adresse des Antwortgeräts verbindet. Dieser Approach ist vermutlich der userfreundlichste, da der User hier nichts selbst machen muss und man sich beim Start der App automatisch zum PC/Laptop verbindet. Jedoch gibt es auch hier mehr als genug Probleme, da es nicht bei jedem Netzwerk mit nur einer Implementierung funktioniert, da einige Netzwerke anders eingestellt sind als andere und somit manchen nicht die Art der C# Implementierung zulassen. Zusätzlich fehlte uns hier das Wissen um alle Arten eines C# Broadcasts zu implementieren. Keine Broadcasts aber trotzdem das ganze Netzwerk erreichen? Mit unseren Kenntnissen ein Ding der Unmöglichkeit also muss der User die IP-Adresse irgendwie selbst eingeben, aber wie macht man dies so Userfreundlich wie nur möglich? Nun zum einem durch Kürzung der IP-Adresse und zum anderen durch einen QR-Code, welcher aufgrund der IP erzeugt wird. So kann der User entweder einen kurzen Code eingeben oder gemütlich seine Kamera benutzten zum Scannen des IP-QR-Codes.

5.5.2 Erzeugung des Lobbycodes

Einfach die IP-Adresse eingeben ist dauert aber zu lange da IP-Adressenen nach dem Muster XXX.XXX.XXX.XXX aufgebaut sind und der User somit ganze 12 Zahlen und 4 Punkte eingeben müsste. Das ganze kann man abkürzen indem die Bytewerte der IP Base64 verschlüsselt werden. Die Base64-Verschlüsselung macht zwar den Code wieder etwas länger jedoch ist er nun ein anschaulicher Code. Zusätzliches Merkmal der Base64 Codierung ist, dass wenn der Verschlüsselte String am Ende nicht auf ein Vielfaches von 3 kommt wird der restliche Platz mit “=” aufgefüllt. Das bedeutet ein String: “agFdeww” auf “agFdeww==” gestreckt. Und bei der Codierung von den Bytewerten der IP-Adressen kommt man zufällig auf genau das, man bekommt einen neunstelligen String welcher am Schluss zwei “=” beinhaltet. Da das aber immer so ist kann man diese einfach vor dem Display des Lobbycodes bereits, im Code, wegschneiden und ihn beim Eingeben des Lobbycodes am Smartphone einfach im Code wieder hinzufügen. So enden wir bei einem siebenstelligen Code welcher für den User einfach und schnell zum eintippen ist. Zusätzlich kann man diesen Code auch noch in einen QR-Code verwandeln und ihm auf dem Smartphone scannen um sich noch mehr Zeit zu ersparen. Im Code ist

das ganze so umgesetzt:
Erzeugung

var b =BitConverter.GetBytes((int)IPAddress.Parse(this.ipAdress).Address);
this.lobbycode = Convert.ToBase64String(b).Split('=')[0];

Zuerst wird mit einem BitConverter gearbeitet um mit GetBytes() alle Bytewerte der IP-Adresse zu bekommen und eben diese dann mit Base64 zu verschlüsseln. Mit this.lobbycode wird ein der lobbycode einem Field übergeben und ist nun für alle Methoden verfügbar, zusätzlich werden hierbei bereits die "==" weggekürzt.

Entschlüsselung

var b = Convert.FromBase64String(code + -=");
var a = new IPAddress(b);
return a.ToString();

Hier wird der String, der den verschlüsselten Lobbycode beinhaltet, zunächst einfach mit Base64 entschlüsselt, wobei hier wichtig ist das die zuvor gekürzten "==" wieder hinzugefügt werden. Dann werden, mithilfe von new IPAddress(), die entschlüsselten Bytewerte zu einer IP Adresse geparst und mit .ToString() wird ein normaler String daraus.

5.6 Verbindung (ES)

Zur Verbindung der Smartphones mit dem Laptop/PC werden Sockets benutzt. Bei den Spielen und auch im Menü werden keine großen Informationen verschickt da dies nicht nötig ist. Es reicht immer nur den jeweiligen Befehl per String zu schicken, auch wenn das öfters in der Sekunde passiert, reichen die Network Streams dafür perfekt aus und sind die beste Option. Dabei wird auf dem Laptop/PC, in einem Thread, eine Socket-Connection geöffnet welche wartet bis sich jemand auf diese verbindet.Die Socket-Connection selbst wird mit dem Port 9393 geöffnet, so wird sie von keiner anderen Anwendung benutzt und es werden keine anderen Anwendung von ihr blockiert. Der Aufbau der Daten, welche per Network Stream versendet werden sind Key-Value Pairs. Bedeutet jeder String den wir verschicken hat einen Key und ein Value, womit es ermöglicht wird als Key einen Befehl zu schicken und als Value Parameter mit zu verschicken. Als Trennzeichen des Key-Value Pairs wurde sich für "—" entschieden, da die ein selten gebrauchtes zeichen ist und am Smartphone nahezu unmöglich ist eintippen. Zusätzlich wird der String mit einem ";" beendet, dies dient dazu mehrere Parameter mit zu schicken welche dann mithilfe eines ":" von einander getrennt werden. Ein möglicher Informationssatz könnte wie folgt aussehen: "Playername—Eric;" oder "Cards—K7:K3".

5.7 Aufbau der Verbindung (ES)

5.7.1 PC/Laptop/Tablet

Bei der Desktop Anwendung von Ludimus gibt es zunächst einen PlayerManager, dieser regelt des Kommunikationsaustausch der Sockets, das Verbinden von neuen Smartphones oder das verlassen bereits verbundener. Der PlayerManager ist dabei ein Singelton da seine Werte für alle immer gleich sein müssen. Wenn der PlayerManager das erste und einzige Mal instanziiert wird, öffnet er einen TCPListener in einem Thread. Dieser wartet so lange bis er geschlossen wird, zusätzlich nimmt er keinen TCPClient mehr an wenn bereits acht Clients verbunden sind. Wenn sich ein Smartphone mit dem Rechner verbindet wird automatisch ein Player Objekt erzeugt, welchem eine ID zugewiesen wird und welchem als Parameter der TCPClient mitgegeben wird. Auch wird das Objekt in eine Liste aller Spieler hinzugefügt. Dieses Player Objekt bietet nun einige Grundfunktionen, wie zum Beispiel den Nachrichten schicken und es beinhaltet aber auch einen Thread welcher die Bytes welche der Network Stream des TCPClients erhält, in einen String umwandelt und schlussendlich in die Unity Queue reiht. Die Unity Queue wird benötigt um wichtige Befehle, wie zum Beispiel einen Scenewechsel durch zu führen, da solche Befehle nur im Main Thread erlaubt sind. Sie wird im Update von Unity öfters die Sekunde geprüft. Man kann mit “enqueue(text)” einfach was einreihen indem man es als Parameter übergibt. Mit “.Count” kann man überprüfen ob in der Queue etwas eingereiht ist, wenn ja kann es mit “.dequeue()”,welches den Wert des eingereihten Strings zurückgibt, entreihen. Also reiht man den geparsten String, welcher vom Network Stream des TCPClients kommt, einfach im Thread ein und im Main Thread wird dann der String ausgewertet und auf Grund seines Wertes dementsprechend gehandelt. Standard sind dabei die Befehle:

- **GetID:** die PlayerID wird dem TCPClient geschickt.
- **RestartCurrentGame:** startet die Scene in der sich der Spieler gerade befindet neu.
- **CloseConnection:** beendet die Verbindung mit dem TCPClient sauber.
- **Playername:** Setzt den Spielernamen auf den auf “Playername” folgenden String

5.7.2 Adroid/IOS

Bei Android/IOS ist das ganze etwas einfacher, hier gibt es nur den ClientController, da hier nur ein TCPClient, welcher sich auf die eingegebene IP-Adresse(durch Lobbycode) verbindet. Es werden direkt am Start auch noch alle nötigen Infos angefragt und gesendet, die benötigt werden damit das Playerhandling am PC/Laptop einwandfrei funktioniert. Auch hier wird die Unity Queue benutzt um im Main Thread die wichtigen Befehle auszuführen. Standard sind hierbei die Befehle:

- **CloseConnection:** Beenden der Anwendung und sauberes schließen der TCPClient Verbindung.
- **ID:** Property ID wird auf den mitgeschickten Value gesetzt.
- **CanRestart:** Führt alle Methoden welche auf dem Restart Delegate hängen aus.

5.8 Für Verbindung wichtige Klassen und Methoden / Übergang Schnittstellen (ES)

5.8.1 PlayerManager

5.8.1.1 Übersicht

Wird erzeugt sobald sich ein neuer TCPClient auf den TCPListener verbindet. Hat alle relevanten Infos die der Spieler besitzt, wie zum Beispiel: Playername oder PlayerID. Beinhaltet einen Thread zur Kommunikation mit dem jeweiligen Smartphone.

5.8.1.2 Wichtige Fields/Properties

Name: **InputHandler**
Info: Delegate auf welches man eine Methode der Signatur: BeispielName(String key, String value) hängen kann. Wird aufgerufen wenn neue Daten per Network Stream eintreffen und es noch keinen vordefinierten Befehl für sie gibt.

Name: **client**
Info: TCPClient welcher auch den Network Stream beinhaltet. Verbindung zum Smartphone

Name: **playerName**
Info: Name des Spielers

Name: **playerID**
Info: ID des Spielers(reicht von 0-7, je nach Zeitpunkt des Verbindens)

Name: **manager**
Info: Instanz des PlayerManager's damit die Kommunikation zwischen Player und Koordinator reibungslos verläuft.

5.8.1.3 Wichtige Methoden

Name: **sendData()**
Parameter: *string key, string value*
Returnvalue: void
Info: schickt dem TCPClient per Network Stream ein Key-Value-Pair. Beim Senden mehrerer value müssen diese hier alle bereits in dem value Parameter vorhanden sein und mit einem ":" getrennt sein.

Name: **kickPlayer()**
Parameter: *keine*

Returnvalue: void
Info: Schließt Verbindung mit TCPClient sofort. Ermöglicht es serverseitig Spieler zu entfernen.

5.8.2 ClientController

5.8.2.1 Übersicht

Dient zum Verbinden des Smartphones mit dem Laptop/PC. Läuft auf Smartphone. Startet einen TCPClient welcher sich zu der, vorher per Lobbycode eingegebenen IP-Adresse verbindet. Hat ebenfalls einen Thread laufen, in welchem per Network Streams mit dem Server(Laptop/PC) kommuniziert wird. Ist ein Singelton.

5.8.2.2 Wichtige Fields/Properties

Name: **inputHandler**
Info: Delegate auf welches man eine Methode der Signatur: BeispielName(string key,string value) hängen kann. Wird aufgerufen wenn neue Daten per Network Stream eintreffen und es noch keinen vordefinierten Befehl für sie gibt.

Name: **restartHandler**
Info: Delegate auf welches man eine Methode der Signatur: BeispielName() hängen kann. Wird aufgerufen wenn CanRestart vom Server geschickt wird.

5.8.2.3 Wichtige Methoden

Name: **getInstance()**
Parameter: *keine*
Returnvalue: PlayerManager
Info: Singelton Konstruktor

Name: **sendData()**
Parameter: *string key, string value*
Returnvalue: void
Info: schickt dem TCPClient per Network Stream ein Key-Value-Pair. Beim Senden mehrerer value müssen diese hier alle bereits in dem value Parameter vorhanden sein und mit einem ":" getrennt sein.

Name: **rename()**
Parameter: *string name*
Returnvalue: void
Info: Einfache Methode welche es ermöglicht den Player schnell umzubenennen.

Kapitel 6

Spiele (ES)

6.1 Spieleentwicklung für Ludimus

6.1.1 Grundgedanke

Grundgedanke: Ein Grundgedanke bei der Entstehung von Ludimus war schon immer das andere Entwickler für Ludimus Spiele entwickeln. Da Ludimus eine Spieleplattform ist benötigt sie ja auch viele Spiele und diese alleine zu entwickeln ist nahezu ein Ding der Unmöglichkeit, deshalb sollten Entwickler, welche C Sharp beherrschen mit hilfe unserer Dokumentation eigenständig Spiele entwickeln können, welche wir nach einem Qualitätscheck dann in unsere Plattform aufnehmen. Dazu haben wir auch Ludimus so "offen" wie nur möglich programmiert. Es sollen den Entwicklern so viele Schnittstellen wie nur möglich gegeben sein, damit diese so kreativ wie möglich werden können. Zusätzlich wird jedoch auch auf Qualität geachtet da nicht jedes Spiel auf unserer Plattform erwünscht ist, da die Ludimus Hauptzielgruppe Familien beinhaltet und somit auch in die Entwicklung seines eigenen Spiels darauf geachtet werden muss Kinderfreundlich zu bleiben.

6.1.2 Motivation

Warum sollte man nun ein Spiel für diese fremde Plattform entwickeln? Als Motivation für Entwickler dient zum einem das Gefühl, dass eventuell hunderte bis tausende Familien ihr Spiel spielen und damit Spaß haben. Andererseits aber auch Geld. Jeder Programmierer der sein Werk auf unserer Plattform veröffentlicht kann damit Geld verdienen, jedoch gibt es keine feste Bezahlung. Den Entwicklern stehen 30

6.1.3 Wie programmiere ich Spiele für Ludimus

Wenn man sich dazu entschließt Spiele für Ludimus zu entwickeln benötigt man bereits Kenntnisse in C Sharp und Unity, da Ludimus auf C Sharp und Unity aufbaut. Um ein Spiel zu erstellen benötigt man zwei Unityprojekte. Eins für das was am Server angezeigt wird und ein zweites für das was die Spieler am Handy sehen. Nun gibt es 3 wichtige Klassen welche der Programmierer wissen muss um die Ludimusschnittstellen auch nutzen zu können. Dies sind der Playermanager und der Player für den Server und der ClientController für das Smartphone. Die einzelnen Klassen werden bereits im Abschnitt "Für Verbindung wichtige Klassen und Methoden / Schnittstellen" behandelt. Es werden alle Methoden aufgezeigt welche ein Entwickler zur verfügung hat und auch alle Delegates auf welche sich der Programmierer subscriben kann und somit eigene Methoden dem Player zuweisen kann. Am wichtigsten hierbei sind der InputHandler der Player Klasse und der InputHandler des ClientControllers. Diese werden aufgerufen wenn der jeweils andere, per Network Stream, ein Key-Value Pair schickt und es noch keine vordefinierte Funktion dafür gibt. Die vordefinierten Keys sind unter der jeweiligen "Übersicht der Verbindung" zu finden.

6.1.4 Testen

Zum Testen der Spiele für Ludimus gibt es einen eigenen Debugmodus. Mehr unter 5.3

6.2 Vorgang Spieleauswahl

Bei der Spieleauswahl für Ludimus ging es zunächst um Spiele die jeder kennt, wie zum Beispiel Poker, welches auch für den Proof of Concept benutzt wurde. Ludimus war am Anfang eine Plattform für Brett,- Karten- und Arcadespiele. Sie ist es noch immer jedoch distanzierten wir uns von der Idee Kartenspiele auf Ludimus langsam immer weiter da der Sinn von Ludimus ist viele Spiele auf einmal mitnehmen zu können und dies bei Kartenspielen nicht sinnvoll erscheint. Kartenspiele sind meist extremst klein und man kann mit einem einfachen Pokerkartenset schon sehr viele verschiedene Spiele spielen. Dazu benötigt man keine eigene Plattform. Für Spiele welche jedoch extremst viel Platz im Koffer brauchen und zusätzlich noch teuer sind jedoch macht es Sinn sie zu digitalisieren und alle auf einer Plattform zu haben. Auch macht es Sinn Spiele welche nur digital spielbar sind oder eine eigene Idee auf der Plattform vertreten zu haben. Also soll Ludimus mehr eine Brett- und Arcadespiele Sammlung sein? Nein. Im Laufe der Zeit änderte sich der Fokus von der Zielgruppe "jeder" auf die Zielgruppe Familie. Dies hatte zur Folge das wir nicht mehr Spiele benötigten welche jeder kennt sonder welche die Kindern und Eltern spaß machen. Auch wurde uns klar das Kinder eher Arcadespiele, welche schnell und einfach zu verstehen sind, spielen, als Brettspiele welche lang dauern und eine lange Erklärung benötigen. Zusätzlich weichen unsere Milestones von den tatsächlichen Spielen weit ab, in der Anzahl und in der Art. Warum die Art sich verändert ist bereits erklärt. Warum sich jedoch die Anzahl veränderte hat auch einen einfachen Grund: Qualität. Eine Plattform welche sich an Eltern und Kinder richtet muss Qualitativ hochwertig sein und auch die Spiele müssen poliert sein. Die Plattform selbst wurde zwei mal neu geschrieben da wir mit der Qualität der vorherigen nicht zufrieden waren, zu wenig Schnittstellen hatten oder eine andere Technologie verwendet wurde. Auch bei den Spielen wurde auf Qualität und nicht Quantität gesetzt, so wurden aus den zehn angestrebten Spiele vier Spiele welche sich jedoch gut spielen und keine Fehler aufweisen.

6.3 Probleme bei der Spiele Entwicklungen

Bei der Spieleentwicklung stießen wir auf viele Probleme, welche meistens aber nur das jeweilige Spiel beeinflussen und deshalb erst später aufgelistet werden. Es gab jedoch auch Probleme welche alle Spiele betreffen, zum Beispiel eine Lösung wie die Spiele zur Runtime geladen werden oder wie die Spiele Zugriff auf die Player-Objekte erhalten. All diese Probleme hatten eine relativ einfache Lösung, nur eines nicht. Ludimus auf dem Tablet! Wie genau soll das Tablet die Rechenleistung eines Laptops/PCs schaffen? Es gibt dabei vor allem 2 große Unterprobleme.

- Erstens, Spiele welche schön ausschauen brauchen nunmal viel Rechenleistung. Rechenleistung die ein Tablet nicht zur verfügung hat. Hierfür gibt es 2 Lösungsansätze und keiner davon ist perfekt. Aber zunächst; warum soll das Spiel schön ausschauen? Grundsätzlich müssen Spiele nicht schön sein aber Kinder und Eltern werden lieber etwas spielen was auf sie optisch einen guten Eindruck macht, da es von Qualität zeugt wenn kein Pixelmatsch auf dem Bildschirm ist. Auch macht es einfach mehr Spaß wenn die Kinder erkennen was genau sie jetzt im Spiel machen und nicht raten müssen was diese Pixel überhaupt darstellen sollen. Lösungsansätze:

- Man kann bestimmte Spiele einfach nur auf dem PC/Laptop anbieten. Diese Spiele sind dann unmöglich am Tablet zu spielen, Problem gelöst. Nicht ganz dem zahlenden User wird die Anzahl der Spiele eingeschränkt nur weil er nicht seinen PC in den Urlaub schleppen will. Zusätzlich ist hier ein großer Faktor das der User dann zwar die Möglichkeit hat einfache Spiele zu spielen, diese aber meistens den Kindern nicht so gut gefallen wie die grafisch anspruchsvollen Spiele, zum Beispiel gefällt den meisten Kindern ein grafisch heraus geputztes Geschicklichkeitsspiel mehr als eine Runde Poker.

- Man kann die Spiele grafisch für das Tablet downgraden. Dies erfordert viel Arbeit welche wir nicht zur Verfügung haben und man rutscht schnell ab und hat dann wieder einen Pixelmatsch mit denen Kinder und Eltern nichts anfangen können.

Für eine der beiden Lösungen sich zu entscheiden fiel dem Team schwer da beide fast nur Nachteile bieten. Jedoch entschloß sich das Team mit der ersten Lösung, da die Nachteile beim grafischen Downgraden zu groß waren. Auch spielte hier das zweite Problem eine große Rolle.

- Zweitens, die komplette Kommunikation zwischen Server und Smartphone findet mit Network Stream statt, welche in einem Thread dauerhaft abgefragt werden. Und wieder finden wir hier das Problem mit der Rechenleistung eines durchschnitts Tablets wieder. Da die Kommunikation aber nicht anders ordentlich möglich ist, ist umschreiben des Codes, nur damit es auf Tablets läuft nicht möglich. Hier bleibt eigentlich nur die Option Spiele welche eine komplizierte Kommunikation benötigen, das heißt mehrere Key-Value Pairs pro Sekunde verschickt werden, nicht auf dem Tablet zur Verfügung zu stellen.

Auch wenn das Team nicht zufrieden war mit der Endlösung, manche Spiele einfach nicht für das Tablet zur Verfügung zu stellen, blieb uns keine andere Wahl da die Technik es nicht erlaubt. Leider waren genau diese Spiele, welche für das Tablet nun nicht verfügbar sind, genau die Spiele mit denen die Kinder am meisten Spaß haben. Somit war das eine der schwersten Entscheidungen im kompletten Projekt, da es auch eine der Grundideen war, das alles am Tablet laufen sollte damit man beim Reisen noch weniger Platz braucht. Zusätzlich war diese Entscheidung mit den vorherigen Spieldesignentscheidungen eine der gravierendsten Änderung im Projekt Ludimus.

6.4 Spiele

6.4.1 Poker

6.4.1.1 Erklärung

Gespielt wird mit einem klassischen Poker-Kartenset welches aus 52 Karten besteht und in welchem es zwei verschiedene Farben, Schwarz und Rot gibt. Zusätzlich hat jede Karte noch eine Art, wobei es hier vier verschiedene gibt: Herz, Karo, Pik und Kreuz. Jede dieser Arten hat die Karten 2-10 und noch Junge, Dame, König und Ass. Beim Texas Holdem Pokerspiel erhalten alle Mitspieler zwei Karten. Danach kommen 5 Karten verdeckt auf den Tisch, nun werden im Uhrzeigersinn Aktionen ausgeführt. Der Spieler hat drei Aktionen zur Verfügung:

- Check: Der Spieler gibt an den nächsten Spieler weiter ohne zu erhöhen. Check kann nur durchgeführt werden wenn in dieser Runde noch nicht erhöht wurde.
- Setzen: Der Spieler setzt teil seines Geldes und erhöht somit den gesamt Pot. Ein Spieler kann maximal sein gesamtes Geld setzen, auch genannt All-In.
- Ablegen: Der Spieler wirft sein Blatt weg und steigt aus. Er kann das aktuelle Blatt nicht mehr gewinnen und hat sein bis jetzt gesetztes Geld verloren.
- Mitgehen: Der Spieler kann mitgehen wenn in der aktuellen Runde jemand bereits etwas gesetzt hat. Er setzt automatisch den Betrag zwischen dem bereits von ihm Gesetzten und dem maximal Gesetzten.

Das ganze wird pro Blatt fünfmal wiederholt. Nach der ersten Runde werden jedoch die ersten drei Karten am Tisch umgedreht nun erhält der Spieler einen besseren Eindruck, von dem was er vielleicht für einen Kartenwert hat. Nach der nächsten Runde wird die nächste Karte aufgedeckt und nach der nochmal nächsten Runde wird die letzte Karte aufgedeckt. Wenn alle Karten aufgedeckt sind startet die letzte Runde. Nach dieser werden die Karten, von allen Spielern welche noch im Spiel sind, aufgedeckt und es gewinnt der Spieler mit dem höchsten Blattwert.

6.4.1.2 Geschichte

Poker wurde als Proof of Concept benutzt da es für unser System perfekt ist. Poker wurde auch immer benutzt um Leuten zu erklären wie unsere Plattform funktioniert, da es mit Poker leicht zu erklären ist. Es war das zweite Spiel der Ludimus-Plattform, wurde aber nicht auf die neue Plattform angepasst, da sich Ludimus von Kartenspielen distanzieren wollte und Familienspiele entwickeln wollte. Poker wird auch nicht mehr zur Erklärung benutzt, da wir zu oft gefragt wurden warum wir eine Familienspielplattform mit Poker erklären wollen und nicht mit einem Familienspiel.

6.4.1.3 Aufbau

Es gibt eine PokerPlayer Klasse in welcher alles wichtige für den Spieler geregelt wird. In dieser stehen sein Player-Objekt, zur Verbindung mit dem Smartphone, sein Geld und die Karten die er zurzeit hat. Zusätzlich gibt es auch eine Tisch Klasse welche die Karten des Tisches und den gesamten Pot speichert. Die Kommunikation ist dabei so aufgebaut das am Anfang ein Key-Value Pair zum Smartphone geschickt wird, welches zwei random generierte Karten enthält und so aussehen kann: "Cards—P7:K4;". Mit dem Keyword Cards wird dem Smartphone gesagt er bekommt jetzt seine Karten und mit P7 ist Pik 7 gemeint bzw mit K4 Karo 4. Nun weiß das Smartphone welche Karten es darstellen muss. Wenn der Spieler an der Reihe ist bekommt er: "Turn—1;" geschickt, wobei Turn das Keyword für seinen Zug ist und ein signalisiert True. Nun werden am Smartphone alle mögliche Züge angezeigt. Die Antwort an den Server ist dabei so aufgebaut das als Keyword sein Zug und als Value das Geld welches er setzt, default 0, gesendet wird. Zum Beispiel: "Check—0;","Fold—0;" oder auch "Bet—1000;". Nach fünf durchgängen checkt ein Algorithmus welcher Spieler gewonnen hat und gibt diesem das Geld und zeigt dessen Namen an.

6.4.1.4 Schwierigkeiten

Um zu ermitteln wer gewonnen hat gibt es leider keine simple Formel oder einen Algorithmus. Um beim Poker spielen zu ermitteln wer nun gewonnen hat muss man leider alle Karten mit den am Tisch liegenden vergleichen. Es führt leider hier kein weg um eine lange IF-Verkettung welche alle Möglichkeiten überprüft und dann schaut wie viel sein Blatt wirklich wert ist. Den Spielablauf zu programmieren hingegen erwies sich als simple.

6.4.2 Jumpy

6.4.2.1 Erklärung

Bei Jumpy müssen maximal vier Spieler gegeneinander ein Rennen zu der Zielflagge antreten. Dabei bewegt der Spieler den Charakter, welcher durch Augen visualisiert wird, mit drei Button auf dem Smartphone. Einer Links, welcher auch nach links zeigt, für die Bewegung nach Links, einer ganz rechts, welcher nach oben zeigt, für das Springen und einer Mittig, welcher nach rechts zeigt, für die Bewegung nach rechts. Auch kann ein Charakter sich von einer Wand abstoßen, mit hilfe der Sprungtaste, um noch höher zu kommen. Dabei ist es gewollt das der Spieler eine Wand hoch kann indem er sich immer abstößt und wieder zur Wand springt. Dies gibt den Spielern noch mehr Freiheiten das Level zu gestalten. Nach jeder Runde gibt es für die Spieler welche das Ziel erreichen Punkte, der Spieler der zuerst das Ziel erreicht erhält mehr Punkte. Bevor aber die nächste Runde anfängt darf der Spieler, der zuerst das Ziel erreicht hat ein Objekt in dem Level platzieren und somit das Level umgestalten. Auf diese Art kann der Spieler ein zu leichtes Level schwerer machen und ein zu schwerer Level einfacher gestalten. Zur

Verfügung stehen dem Spieler dazu Plattformen, welche wie normale Levelwände agieren, Sprungbretter, welche den Spieler höher springen lassen, einen Ventilator, welcher den Spieler in die Luft befördert und Stacheln, welche bei der Berührung den Spieler sofort aus der Runde ausscheiden lässt. Für die Bewegung des Objektes wird das UI am Smartphone ausgetauscht und der Spieler sieht nun vier Pfeile in alle Richtungen, Unten, Oben, Links und Rechts. Die Pfeile agieren als Buttons und senden den Befehl an das Spiel, der das Objekt in die gewünschte Richtung bewegt. Auch wird ein Häkchenen-Button am UI erscheinen durch welche Betätigung der Befehl zur Festigung des Objektes an das Spiel gesendet wird.

6.4.2.2 Geschichte

Jumpy entstand als Demospiel für das AEC-Festival wurde aber weit mehr. Das Spiel kam am AEC-Festival so gut an, dass wir es nochmal überarbeitet und optisch verbesserten. Mehr dazu bei der Idee für das AEC-Festival 7.2.2

6.4.2.3 Aufbau

Der Kommunikationsaufbau bei Jumpy ist sehr einfach gehalten. Wenn ein Spieler auf einen Button drückt wird der jeweilige Befehl an den Server in Form eines Key-Value Pairs gesendet welches keinen richtigen Value hat. Bsp.: “startLeft—;”, “stopLeft—;”, “jump—;”. Wenn der Spieler nun das Ziel erreicht wird ihm geschickt, dass er sich nicht mehr bewegen kann. Zum Handling im Backend gibt es drei Listen: ActivePlayers, WinPlayers, DeadPlayers. Wenn ein Spieler das Ziel erreicht wird er von ActivePlayer zu WinPlayers transferiert. Wenn er jedoch ausscheidet wird er zu DeadPlayers transferiert. Wenn die Anzahl von WinPlayers + DeadPlayers der von ActivePlayers gleicht ist eine Runde aus. Nun bekommt jeder in WinPlayers seine Punkte welche auf sein JumpyPlayer-Objekt festgelegt werden. Der erste der Liste WinPlayers darf nun ein Objekt bewegen, welches aus den Unity-Assets geladen wird, welches Objekt er bewegen darf wird mit einem, von C# bereitgestellten Random. Dieser Random funktioniert indem man zunächst ein Random-Objekt instanziiert: Random random = new Random();. Mit der Methode .Next(int Unteregränze, int Obergrenze) kann man nun einen Randomwert in dem Wertebereich erzeugen, dabei ist wichtig das die Untergrenze inklusive und die Obergrenze exklusiv sind, das heißt random.Next(0,100) bedeuted man bekommt einen Wert welcher von 0-99 reicht. Will man einen Wert von 0-100 haben muss der Randomwert so random.Next(0,101) erzeugt werden. Der welcher von der .Next Methode kommt ist jedoch keine echte Zufallszahl da diese mit einem Computer weit schwerer zu erzeugen ist, reicht jedoch für die Benutzung in Jumpy aus. Danach spielt man erneut bis es keine aktiven Spieler hat, bis zu dem Zeitpunkt wo der Score eines Spielers einen gewissen vorher eingestellten Wert erreicht.

6.4.2.4 Schwierigkeiten

Die erste große Hürde war der Walljump, also die Mechanik das ein Spieler sich von der Wand abstoßen kann um an der Wand hoch zu klettern. Dazu wurde ein System eingebaut welches bei einer Objekt-Berührung berechnet von welcher Seite das Objekt die Wand berührt. Auch bei der Bewegung des Charakters gab es Probleme, da der Charakter sich solange bewegen sollte wie der Spieler nach links/rechts drückte. Bei Unity Buttons jedoch gibt es nur eine Event wenn man den Button drückt und auslässt. Bei der fertigen Anwendung wird bei einem Drücken ein Befehl geschickt welcher den Spieler in die gewünschte Richtung solange bewegt, bis der Spieler den Button loslässt und ein Stop-Befehl geschickt wird. Beim Platzieren des Objektes gab es auch die Schwierigkeit wie zwischen Objekt und Spielcharakter umgeschalten wird. Dies wurde so Implementiert indem der Spielercharakter während der Bewegung einfriert und sich nicht bewegen kann. Währenddes wird ein neues Objekt aus den Assets instanziert und mit einer Variable übergeben. Nun kann der Spieler mit dem neuen UI das Objekt platzieren. Wenn die Bestätigung für die Position vom Server erhalten wird, wird das Objekt in eine Liste hinzugefügt, welche alle selbst platzierten Objekte enthält. Zum Schluß wird die Variable auf null gesetzt und dem Smartphone gesendet, dass es das UI wieder wechseln muss. Auch war das Design hier ausgesprochen schwer zu gestalten, da weder mein Kollege und Ich ausgezeichnete Künstler sind. Am Schluss wurden wir aber nach stundenlangem Zeichnen Fertig und sind mit dem Ergebnis sehr zufrieden.

6.4.3 Knightslider

6.4.3.1 Erklärung

Bei Knightslider werden die Spieler als Ritter dargestellt. Es gibt 3 Lanes auf welchen man stehen kann und mit einem wischen auf dem Smartphone kann man die Lane wechseln um Objekten, welche herunterfallen, auszuweichen. Sobald man von einem der fallenden Objekte getroffen wird scheidet man aus. Das Ziel ist als letzter noch im Spiel zu sein.

6.4.3.2 Geschichte

Wir wollten für Ludimus ein zweites endless runner Arcade-Spiel, da diese schnell zu kapieren sind und viel Spaß machen, auch wenn man sie nicht so gut kann. Die Idee einen Ritter zu nehmen entstand eher spontan.

6.4.3.3 Aufbau

Bei Knightslider wird vom Smartphone gesendet in welche Richtung sich der Spieler bewegen will. Dies geschieht mit den üblichen Key-Value Pairs welche hier den Aufbau :”right—;” oder “left—;” haben.

6.4.3.4 Schwierigkeiten

Bei der Programmierung selbst traten keine größeren Probleme auf, bei der Modellierung jedoch schon. Bei Knightslider haben wir das erste mal richtige 3D-Modelle verwendet und das Team hatte vorher keine Ahnung wie man 3D modelliert. Zum Glück gibt es Blender **??**, durch dieses Programm hatten wir die Möglichkeit unsere 3D Objekte selbst und schnell zu modellieren.

6.4.4 Billard

6.4.4.1 Erklärung

Beim Billard liegen 16 Bälle auf einem Billardtisch, welcher eine spezielle Oberfläche hat. Dabei gibt es 1ne Weiße, 7 halb gefärbte, 7 komplett gefärbte und 1ne Schwarze Kugel. Ziel des Spieles ist es mit einem Stab, welcher Queue genannt wird, die weiße Kugel so anzustoßen dass sie eine weiter Kugel anstößt und diese in eine Ecke des Tisches befördert. In den 4 Ecken des Tisches und auf den beiden Längen, jeweils in der Hälfte, sind Löcher. Eine Kugel gilt dann als versengt wenn die Kugel in einem der Löcher landet. Gewonnen hat der Spieler, welcher zuerst alle Kugeln seiner Farbe, dies wird durch die erste Kugel welche versenkt wird entschieden, versenkt und dann noch die schwarze Kugel in das richtige Loch trifft. Die schwarze Kugel jedoch darf erst ganz am Schluss versenkt werden da man sonst automatisch das Spiel verliert.

6.4.4.2 Geschichte

Zunächst wollten wir ein anderes Spiel zu implementieren bei welchem man eine Kugel steuert und andere Kugeln wegstoßen muss. Bei der Entwicklung jedoch erhielten wir immer wieder das Feedback das wir einfach Billard programmieren sollten. Nach einer Ewigkeit gab das Team nach und programmierte das Spiel schnell zu einem richtig Billard um.

6.4.4.3 Aufbau

Der erste Kommunikationsaustausch findet statt wenn der Spieler sein Smartphone berührt. Dabei wird der Koordinatenvektor mit geschickt welcher vom Spiel vermerkt wird. Wenn der Spieler nun seinen Finger vom Smartphone hebt wird der Endvektor an den Server gesendet. Dieser berechnet sich, durch die beiden Vektoren, in welche Richtung der Spieler gezielt hat und durch die Distanz der beiden Vektoren wie fest der Spieler schießen will. Wenn er das berechnet hat, bewegt dann die weiße Kugel um diesen berechneten Vektor. Durch die Physik des Spieles werden alle Kugeln welche berührt werden automatisch weggeschoben.

6.4.4.4 Schwierigkeiten

Die Physik des Spieles. Der Kommunikationsaustausch war schnell erledigt, das die Kugeln jedoch richtig angezeigt und berechnet werden war schon um einiges schwerer. Mit

Unity Materials glang es uns jedoch schlussendlich doch eine gut funktionierende Physik einzubauen.

6.4.5 Squarerunner

6.4.5.1 Erklärung

Ein Arcade-Game bei welchem es hauptsächlich um Geschicklichkeit geht. Der Spieler steuert einen Würfel, auf dem PC/Laptop, mit seinem Smartphone. Zum Steuern des Würfels kann der Spieler sein Handy nach Links und Rechts neigen. Ziel ist es den entgegenkommenden Blöcken auszuweichen und als letzter Spieler noch am Leben zu sein.

6.4.5.2 Geschichte

Square runner war zunächst ein, auf Unity basierendes, Geschicklichkeitsspiel für das Smartphone, entwickelt von meinem Kollegen David Matousch. Der Code für dieses Spiel wurde kurz umfunktioniert und auf eine Server- Client Verbindung aufgeteilt. Zusätzlich wurde noch die Ludimus Schnittstelle mit hilfe den bereitgestellten Delegates implementiert und schon war das erste Spiel der Ludimus-Spieleplattform fertig.

6.4.5.3 Aufbau

Das Spiel funktioniert indem die einzelnen TCPClient-Verbindungen jede 0.25 Sekunden die Neigung ihres Smartphones übermitteln. Beispiel des Aufbaus der gesendeten Key-Value Pairs: “Tilt—0.51;” oder “Tilt—-0.5;”. Mit hilfe der Neigung konnte man berechnen wie der Spieler sein Smartphone hält und in welche Richtung er seinen Charakter bewegen will. Schwierigkeiten: Die Neigung unter Unity zu berechnen war das größte Problem bei Square Runner. Zum Glück war auch dies leicht lösbar denn mit:
- Input.acceleration.x
- Input.acceleration.z
- Input.acceleration.y
kann man in Unity schnell und einfach die Rotation des Smartphones bekommen. Zusätzlich waren bei diesem Spiel auch sehr viele Fehler im zusammenhang mit der Nutzung der Network Streams, da es das erstes Spiel war und wir uns erst vertraut mit der Materie machen mussten.

Kapitel 7

Ars Electronica Festival (ES)

7.1 Übersicht

Ars Electronica ist eine der weltgrößten Bühnen für Medienkunst, ein Festival für digitale Musik, eine Messe für Kreativität und Innovation und Spielwiese für die nächste Generation – Ars Electronica ist ein weltweit einzigartiges Festival für Kunst, Technologie und Gesellschaft.

7.2 Ludimus Testlauf

7.2.1 Wie es dazu kam

Dank des Einsatzes von DI. Professor Peter Bauer durften Teams aus dem ThinkYourProduct-Projekt ihre Ideen und Projekte auf dem AEC Festival präsentieren. Das ThinkYourProduct-Projekt ist für alle 4ten Klassen der HTL Leonding zugänglich. In dessen Rahmen überlegt man sich neue Ideen und Technologien aus und arbeitet diese aus. Aus genau so einer Idee ist auch Ludimus, unsere Diplomarbeit entstanden. Team Ludimus sah dieses Festival als einen Testlauf um zu sehen ob unsere Idee ankommt und ob die Kinder oder Eltern begeistert sind.

7.2.2 Idee

Wir bauen eine gemütliche Wohnzimmerumgebung nach und lassen die Menschen auf einer Couch sitzen während sie Ludimus testen, war der Grundgedanke. Auch war ein großer Bildschirm notwendig, wie sollen die Leute sonst das Spielgeschehen aus einer gewissen Distanz gut sehen? Zusätzlich wird noch ein lokales Netzwerk benötigt, durch welches es möglich ist sich, mit der Ludimus-App am Smartphone installiert, mit dem Laptop, welcher die Ludimus-App ebenfalls gestartet hat, zu verbinden. Noch ein paar Deko-Elemente und schon ist der Ludimus-Stand für das AEC Festival fertig. Fehlte nur noch ein Spiel welches schnell und lustig war aber trotzdem noch so anspruchsvoll, dass es für kurze Zeit fesselt. Die Lösung: ein Jump and Run in welchem die Spieler das Ziel erreichen müssen und dabei schneller als ihre Mitspieler sein sollten. Und zur Belohnung erhält der erste im Ziel mehr Punkte und darf das Level, in welchem gespielt wird, verändern. Mit Veränderung ist gemeint, dass der Spieler ein Objekt in der Welt platzieren darf um das Level leichter oder schwerer zu machen. Und welcher Spieler zuerst dreimal gewinnt, gewinnt insgesamt. Ein kurzes aber trotzdem anspruchsvolles Spiel welches spaßig ist. Dabei gab es jedoch ein Problem, das Spiel hatte nichts mit dem Error-Thema des Festivals zu tun. Also bauten wir einen “Bug” in das Spiel ein, bei welchem die Spieler ihr Smartphone schütteln müssen um seinen Charakter wieder bewegen zu können. Dieser Bug tritt bei dem Spieler auf welcher in der jetzigen Runde gerade die meisten Punkte hat. Und fertig war es die Idee für perfekte Spiel für das Festival welches sogar zum Thema passte.

7.2.3 Realität

Zunächst benötigten wir eine Couch. Zum Glück hatte eine Klasse unserer Schule eine Couch welche wir uns borgen durften, also wäre das wichtigste schon mal geschafft. Wegen dem Bildschirm mussten wir uns auch keine Sorgen machen denn auch hier war uns die HTL Leonding eine riesige Hilfe da sie uns einen großen Bildschirm der sonst für Präsentationen gedacht war bereit stellte. Dieser hatte zwar eine kleine Anzeigeverzögerung zwischen Laptop und Bildschirm war sonst aber perfekt. Als nächstes benötigen wir nur noch ein lokales Netzwerk, damit sich die Spieler verbinden können. Dafür wurde ein Hotspot auf dem Smartphone eines Teammitgliedes erstellt und schon konnte man spielen. Als Deko-Elemente wurden ein paar Sessel und Blumen noch hinzu gestellt und der Ludimus-Stand war offiziell komplett. Fehlte nur noch das Spiel. Dieses wurde liebevoll Jumpy genannt und ist im Kapitel "Jumpy" genauer ausgeführt.

7.3 Ergebnis

Das AEC-Festival war für Team Ludimus durchaus erfolgreich. Wir erhielten durchgehend positives Feedback und waren positiv Überrascht das es so gut läuft. Der Stand war nur sehr selten wirklich leer im Verlauf der vier Tage und wir erhielten viele Meinungen und konstruktive Kritik. Auch waren viele andere Programmierer auf dem AEC-Festival mit welchem wir uns auch austauschen konnten und uns weiterbilden konnten. Über den Verlauf des gesamten Festivals loggten sich über 150 Leute, mit ihrem Google-Account, auf der Ludimus-Website ein um die App zu downloaden. Dies war ein riesiger Erfolg für unser Projekt. Jedoch realisierten wir auch wie hartnäckig Eltern eigentlich sind wenn es sich um Videospiele oder moderne Technologie handelt. Den Satz: "Ich kann sowas ja gar nicht, muss ich gar nicht erst probieren" hörten wir locker über 20 mal Tag, auch wenn wir erklärten, dass es auch andere Spiele gibt wie zum Beispiel Poker blieben sie bei der Meinung, dass sie sowieso nicht mit dem Smartphone umgehen können und es lieber nicht probieren wollen. Dank dieses Feedback realisierten wir jedoch auch großes Problem, denn wir mussten die Eltern in Zukunft auch dazu bringen mit den Kindern zu spielen. Schließlich war das der Zweck von Ludimus. Als Lösung wurden einfach genaue Anleitung hinzugefügt und es wurde immer wieder betont wie wichtig es nicht sei, dass die Eltern mit ihren Kindern spielten.

7.4 Probleme

Eigentlich verliefen die vier Tage sehr gut, hin und wieder erhielten wir genervte Bemerkungen von Eltern die nicht mochten das ihre Kinder Videospiele spielten. Außerdem war manchen Eltern nicht bewusst, dass wir ein Projekt und nicht die Kinderabgabe sind. Aber all das bei Seite war es ein großer Erfolg für Ludimus und wir konnten viel aus dem Feedback lernen und dank der Menschen auf dem AEC-Festival unsere App noch besser machen.

Literaturverzeichnis

Abbildungsverzeichnis

1.1 Ludimus Logo 8

3.1 Unity Editor 16
3.2 Unity Basisscript 21
3.3 Objekt erzeugen über Resources 23
3.4 Objekt erzeugen durch Klonen eines Objektes 23
3.5 Beschaffung aller Objekte mit einer bestimmten Eigenschaft 24
3.6 Privates Feld mit SerializeField Annotation 24
3.7 Inspector Ansicht von Feldern 24
3.8 Öffentliche Felder mit komplexen Datentypen 25
3.9 Inspector Ansicht von mehreren komplexen Feldern 25
3.10 Rigidbody des eigenen Objektes erhalten 25
3.11 three-way handshake 30

5.1 Erster Shop Prototyp 44
5.2 Lobby Prototyp 45
5.3 Shopdetailansicht mit Einstellungen und hellem Farbschema 45
5.4 Schwarz-Weiß Designentwurf 46
5.5 Buntes Hauptmenü mit Schatten und abgerundeten Ecken 47
5.6 UI zum Verbinden zu einer Lobby 48
5.7 Finale Shoppräsentation mit Debug Modus 49
5.8 Vorgang bei Spielstart 51
5.9 Aufbau Ludimus 54
5.10 Ablaufdiagram Ludimus 56