ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

Evaluation von gestenbasierten Gaming-Techniken zur Unterstützung von exekutiven Funktionen im Kontext klinischer Psychologie

Diplomarbeit an der Universität Ulm

Vorgelegt von:
Alexander Fürgut
alexander.fuergut@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert
Dr. Winfried Schlee

Betreuer:
Marc Schickler
Patrick Fissler

2013

Fassung 27. Januar 2014

Satz: PDF-LaTeX 2$_\varepsilon$

Kurzfassung

Videospiele werden häufig als reine Unterhaltung wahrgenommen, bieten aber als sogenannte Serious Games ein weitaus größeres Potenzial, denn sie können mithilfe der Psychologie kognitive Eigenschaften des Spielers trainieren oder ihn lernen lassen. Durch den technischen Fortschritt entstehen zudem immer neue Möglichkeiten der Mensch-Computer-Interaktion wie der Leap Motion Controller, mit dem Finger- und Handbewegungen berührungslos zur Eingabe verwendet werden.
Diese Arbeit nutzt das Spiele-Framework Unity, um die Anforderungen an ein Serious Game, mit der Bewegungssteuerung der Leap Motion zusammenzuführen und einen Prototypen zu entwickeln, mit dem die exekutiven Funktionen trainiert werden sollen. Dieser kann vor allem älteren Menschen in der Therapie helfen, da deren Exekutivfunktionen häufig beeinträchtigt sind.
Dafür wurde unter anderem der Trail Making Test als Computerspiel umgesetzt und in einem Piraten-Szenario mit frei begehbarer 3D-Welt eingebettet.
Da es keine Erfahrungswerte mit dem Einsatz der Leap gibt, wird diese im Rahmen dieser Arbeit ebenfalls evaluiert und verschiedene Steuerungskonzepte erarbeitet.

Inhaltsverzeichnis

1

Einleitung

Computerspiele werden oft als reine Unterhaltung wahrgenommen, bieten aber ein weitaus größeres Potenzial, denn sie können mithilfe der Psychologie kognitive Eigenschaften des Spielers trainieren oder ihn lernen lassen. Diese sogenannten *Serious Games*, sind grafisch und spielerisch oft sehr schlicht gehalten und bleiben deshalb hinter ihren Möglichkeiten zurück.
Durch moderne Spiele-Frameworks wie *Unity* und frei verfügbare Ressourcen, ist es mittlerweile einfacher geworden, ansprechende Spiele zu entwickeln.

Aufgrund des technischen Fortschritts entstehen zudem immer neue Möglichkeiten der Mensch-Computer-Interaktion, die innovative Spiel- und Steuerungskonzepte ermöglichen. Einen Anstoß zu dieser Arbeit hat die Vorveröffentlichung der *Leap Motion* gegeben, mit deren Hilfe es erstmals möglich ist, Fingerbewegungen im Raum zur Steuerung zu verwenden.

Dabei stellt sich die Frage, wie sich diese Technologie produktiv einsetzen lässt, welche Hürden bei der Implementierung überwunden werden müssen und ob damit eine Steuerung entwickelt werden kann, die für verschiedene Altersgruppen gleichermaßen nutzbar ist.

1.1 Ziel dieser Arbeit

Das Ziel dieser Diplomarbeit ist es deshalb, ein Spiel als *Proof of Concept* zu entwickeln, dass die Steuerung der *Leap Motion*, mit *Unity* und den Anforderungen an ein *Serious Game* verbindet, auf einem psychologischen Fundament aufbaut und grafisch, sowie spielerisch ansprechend gestaltet ist.

Um die Ansprüche an ein Serious Game zu erfüllen, soll das Training der exekutiven Funktionen im Vordergrund stehen, da sich diese mit dem Alter immer weiter verschlechtern, was durch entsprechendes Training gemildert werden kann. Zudem sollen Grundsätze des *Game-Designs* in Form des *GameFlow* Modells beachtet werden.

Dabei soll das Spiel komplett mit Finger- und Handbewegungen gesteuert werden, um dadurch die *Leap* zu evaluieren und ihre Stärken und Schwächen zu beleuchten. Dafür sollen verschiedene Steuerungskonzepte erarbeitet und vorgestellt werden.

Um den Trainingseffekt messen zu können, soll das Spiel zudem Daten speichern, über die ein Psychologe Rückschlüsse auf das Spielverhalten und damit den Zustand der exekutiven Funktionen des Spielers ziehen kann.

1.2 Aufbau dieser Arbeit

Da diese Arbeit damit am Schnittpunkt von Informatik, Psychologie und *Game-Design* entsteht, gibt es einige Grundlagen dieser Fachbereiche, die für ein besseres Verständnis erläutert werden. Dazu werden in Kapitel 2 die Grundlagen und der Aufbau von *Unity* sowie der *Leap Motion* behandelt und die Psychologie dieser Arbeit erklärt.

Was allerdings nicht behandelt werden kann, sind Grundlagen der Informatik und des Programmierens. Diese werden als gegeben vorausgesetzt.

In Kapitel 3 werden anschließend die Anforderungen an das zu entwickelnde Spiel, in Hinblick auf die Psychologie sowie den Einsatz der *Leap* und *Unity* formuliert.

In Kapitel 4 wird neben der Wahl des Szenarios, die Implementierung und der Aufbau des Spieles in *Unity* erläutert. Dazu wird jeweils auf die grafische Umsetzung, den Einsatz der *Leap Motion* zur Steuerung und die entwickelten Skripte eingegangen.

Schließlich wird in Kapitel 5 ein kurzes Fazit der Anforderungen gezogen und in Kapitel 6 folgt ein Ausblick auf Erweiterungsmöglichkeiten der Implementierung, die weitere technische Entwicklung der *Leap* und eine Zusammenfassung der Ergebnisse.

2

Grundlagen

Das im Rahmen dieser Diplomarbeit entwickelte Spiel *Where are my Pirates* basiert auf zwei Schlüsseltechnologien, dem Spiele-Framework *Unity* sowie dem *Leap Motion Controller* zur Gestensteuerung, mit dem Finger- und Handbewegungen erfasst und ausgewertet werden.

Zum besseren Verständnis der anschließenden Kapitel, wird in diesem Kapitel ein Überblick über die Grundlagen von Unity, der Leap Motion und des Zusammenspiels beider Komponenten gegeben. Hierbei werden häufige Fehlerquellen aufgezeigt und Hinweise zur optimalen Verwendung (*Best Practice*) gegeben.

Abschließend wird der psychologische Hintergrund dieser Arbeit, in Form des *GameFlow* und der *exekutiven Funktionen* grundlegend erläutert.

2.1 Unity

Unity ist eine in der Grundversion kostenlose Spiele-Engine und Entwicklungsumgebung von *Unity Technologies* aus San Francisco, welche die plattformunabhängige Entwicklung für mobile Geräte [SSP+13, GPSR13], Desktop Systeme, Spielekonsolen und Webbrowser ermöglicht. Dies wird über zum Teil kostenpflichtige Zusatzmodule realisiert, welche es erlauben, das eigene Projekt auf die Zielplattform zu exportieren.

Unity bringt zur Bearbeitung von Skripten seit Version 3 *MonoDevelop* mit und bietet zudem eine hervorragende Dokumentation samt Handbuch, auf welche in diesem Kapitel sehr viel Bezug genommen wird und hier zu finden ist:
http://unity3d.com/learn/documentation

In dieser Diplomarbeit wird Unity 4 in der Pro Version verwendet, da für die Einbettung der Leap Motion deren Plugin System zum Einsatz kommt, was in Abschnitt 2.3 näher erläutert wird.

2.1.1 Aufbau

Unity baut organisatorisch auf Projekten *(Projects)* und Szenen *(Scenes)* auf. Ein Projekt ist der Oberbegriff für eine einzelne Anwendung bzw. ein einzelnes Spiel und kann mehrere Szenen enthalten. Eine Szene wiederum enthält die eigentlichen Spielobjekte und kann als einzelnes Level eines Spiels betrachtet werden. Sie enthält Objekte, die verschiedene 3D-Modelle oder Elemente wie Lichter, Kameras oder Partikeleffekte sein können.

In Unity ist jedes Objekt der Szene gleichzeitig ein (Spiel)Objekt *(GameObject)*, welches das Containerformat für einzelne Komponenten *(Component)* darstellt. Hierbei ist alles was einem Objekt an Eigenschaften oder Effekten zugewiesen wird, als Komponente oder Teil einer Komponente organisiert. Dabei enthält jedes GameObject mindestens seine *Transform* Komponente, die Position, Rotation und Skalierung festlegt, sowie einen Namen, ein *Tag* und die Zuordnung zu einem *Layer*.

Dabei können per *Layer* und *Tag* Objekte semantisch gruppiert werden, was eine weitere Möglichkeit bietet, diese in Skripten zu unterscheiden. Auch die Sichtbarkeit von Objekten für die Kamera wird üblicherweise per Layer realisiert.

Das Anzeigen von Objektkomponenten ist die Aufgabe des **Inspectors**, auf den in Abschnitt 2.1.2.4 eingegangen wird.

2.1.2 Unitys GUI

Die Benutzeroberfläche von Unity ist in der Standardkonfiguration in vier Fenster aufgeteilt: **Project**, **Scene**/**Game**, **Hierarchy** und **Inspector**, welche in Abbildung 2.1 zu sehen sind.

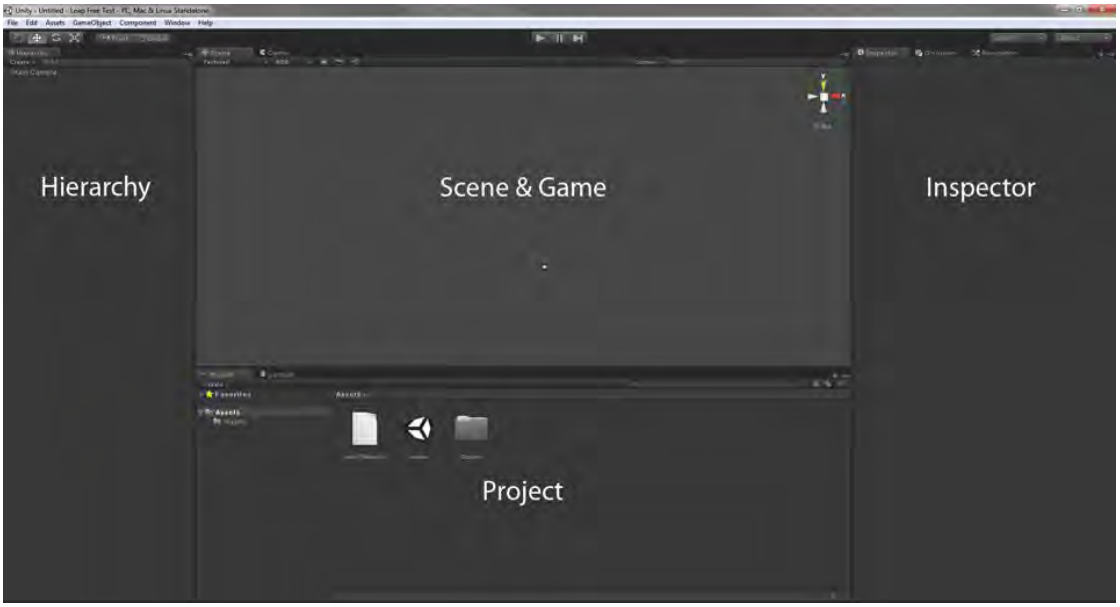

Abbildung 2.1: Unitys GUI

2.1.2.1 Project

Das *Project* Fenster enthält die beiden Menüpunkte *Favorites* und *Assets*, wobei hier nur der *Assets* Ordner interessant ist.
Dieser enthält zwingend alle von Unity für dieses Projekt importierten Dateien, stellt

somit eine Bibliothek mit allen in diesem Projekt verfügbaren Ressourcen dar und dient zum Zugriff auf diese. So erfolgt die Interaktion mit externen Ressourcen immer über den Asset Ordner, indem Ressourcen entweder manuell oder von Unity automatisch dort hinein kopiert werden.
Dabei ist der Aufbau des Project Fensters einem Datei Explorer wie dem Windows Explorer sehr ähnlich. Links ist die Ordnerhierarchie zu sehen und rechts wird der Inhalt des selektierten Ordners angezeigt.

Der Asset Ordner gilt projektweit und ist nicht auf einzelne Szenen begrenzt, da die Assets selbst keine Spielobjekte, sondern die Baupläne für diese sind und keine Entsprechung in der Szene haben müssen. Vergleicht man diesen Ansatz mit objektorientierten Programmiersprachen, stellen die Assets nur die Klassen dar, aufgrund derer die eigentlichen Objekte erstellt werden.

Assets können von externen Quellen (Abschnitt 2.1.4) oder als bereits enthaltene *Standard Assets* (Abschnitt 2.1.3.1) importiert werden sowie über den *Asset Store* (Abschnitt 2.1.3.2) heruntergeladen werden.

2.1.2.2 Scene/Game

Dieses Fenster enthält die beiden Tabs *Scene* und *Game*, die den Inhalt der Szene in einer Bearbeitungsansicht und in einer Ansicht, die dem fertigen Spiel entspricht darstellen.

Abbildung 2.2: Unterschied zwischen dem Scene und Game Tab

Scene stellt die Szene für den Entwickler aufbereitet dar und dient zur Objektplatzierung und -selektierung, weshalb verschiedene optische Hilfen eingeblendet werden. Beispiele hierfür sind das Koordinatenkreuz (*Scene Gizmo*), dass die Kamera Ausrichtung anzeigt, ein Raster, deaktivierbare Beleuchtung sowie die sichtbaren Achsen bei ausgewählten Objekten. Zudem ist die Ansicht nicht an die Sicht einer speziellen Kamera gebunden, sondern kann frei geändert werden.

Game zeigt die Szene so wie sie später für den Spieler aussieht, d.h. die Szene wird aus der Sicht der Hauptkamera angezeigt, welche durch das vorgegebene *MainCamera Tag* als solche markiert ist. Des Weiteren sind alle Werkzeuge und Hilfen für Entwickler deaktiviert und dafür Beleuchtung sowie Effekte aktiviert.

Beim Starten der Szene wird automatisch von der Scene auf die Game Ansicht gewechselt.

2.1.2.3 Hierarchy

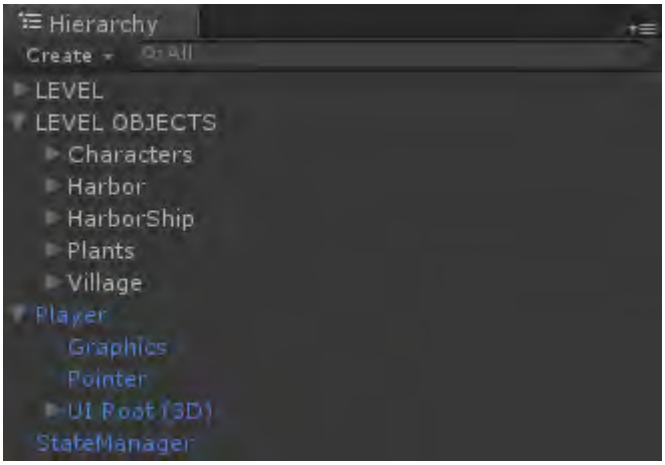

Abbildung 2.3: Hierarchy Ansicht

In der *Hierarchy* ist jedes Spielobjekt der aktuellen Szene enthalten, was verschiedene 3D-Modelle oder Objekte, wie Lichter, Kameras und Partikeleffekte, sein können.

Ist ein Objekt in der Hierarchy ausgewählt, wird es im Scene Fenster hervorgehoben, indem die Achsen sowie bei einem 3D-Modell das Mesh[1] eingeblendet und die Komponenten des Objekts im *Inspector* angezeigt werden.

Hierbei kann jedes Objekt per Drag & Drop Kindobjekte erhalten oder anderen Objekten untergeordnet werden, wobei die Kindobjekte immer Position/Rotation/Skalierung im Verhältnis zu ihren Eltern beibehalten und diese bei Änderungen am Elternobjekt über ihre *Transform* Komponente übernehmen.

Somit können sich Objekte, die miteinander in Beziehung stehen, zugeordnet und gemeinsam verändert werden. Dies erleichtert die Organisation von Skripten, da zum Beispiel sehr einfach auf alle Kindobjekte oder von den Kind- auf die Elternobjekte zugegriffen werden kann.

2.1.2.4 Inspector

Abbildung 2.4: Der Inspector mit Public Variablen

[1] Das Polygonnetz bzw. Gitter mit der Geometrie des 3D-Modells

Der *Inspector* zeigt alle Komponenten eines Objektes mitsamt ihren Eigenschaften an und ermöglicht deren Werte innerhalb von Unity zu ändern. Dies gilt auch für Variablen aus Skripten des Objekts, die als *public* (allerdings nicht *static*) deklariert wurden.

Als Besonderheit von Unity ist dies auch zur Laufzeit der Anwendung möglich, sofern sie im Editor gestartet wird. Dadurch kann direkt aus der Anwendung heraus sehr einfach experimentiert werden. Vor allem bei Werten die eher nach Gefühl gesetzt werden (z.B. ob der Rauch noch etwas schneller aufsteigen, die einzelnen Wolken etwas größer oder die Streuung breiter sein soll), wird die Entwicklung deutlich beschleunigt.

2.1.3 Assets

Abbildung 2.5: Verschiedene Assets und Szenen

Unity organisiert alle verfügbaren Ressourcen im *Assets* Teil des Project Fensters, welcher in Abbildung 2.5 zu sehen ist. Dabei sind Assets Medieninhalte des Projekts, egal ob Bild, 3D-Modell oder Skript. Um sie in der Szene zu verwenden, werden diese per Drag & Drop in das *Scene* Fenster oder die *Hierarchy* gezogen.

Dabei hat der Asset Ordner eine Entsprechung im Projektverzeichnis auf dem Dateisystems und Unity versucht automatisch Dateien, die diesem Ordner hinzugefügt werden, als Asset zu importieren.

In der Szene nicht verwendete Assets, werden von Unity beim *Build* Vorgang nicht mit gespeichert, weshalb diese keinen Einfluss auf die Größe oder Performance des fertigen Spiels haben. Allerdings können sie die Projekt Ladezeiten in Unity selbst erhöhen.

Zwei weitere Möglichkeiten seinem Projekt Assets hinzuzufügen sind die *Standard Assets* und der *Asset Store*.

2.1.3.1 Standard Assets

Unity bietet dem Entwickler eine Reihe von Standard Assets, welche häufig genutzte Funktionen oder Objekte darstellen und standardmäßig enthalten sind. Diese können direkt beim Anlegen eines Projektes oder nachträglich über die Import Funktion geladen werden. Dabei bietet die Unity Pro Version zusätzliche Standard Assets, wie z.B. besseres Wasser.

Beispiele für in dieser Arbeit verwendete Standard Assets sind:

- **Character Controller**: Skripte um eine Figur aus der Ego- oder Schulterperspektive zu steuern.
- **Skyboxes**: Der die Welt umschließende Himmel.
- **Terrain Assets**: Bäume, Sträucher, Gras sowie Landschaftstexturen.
- **Water**: Animierte Wasserflächen.

Es wäre folglich sehr mühsam diese Standardelemente jedes mal neu zu erstellen.

2.1.3.2 Asset Store

Eine weitere praktische Asset Quelle ist der Asset Store, welcher eine in Unity intigrierte Vertriebsplattform für Assets aller Art ist. Von 3D-Modellen, über Animationen, Audio-Ressourcen, Skripten, Texturen oder Materialien, bis hin zu kompletten Projekten, Editor Erweiterungen und verschiedenen Diensten, die als Plugin angeboten werden, ist alles vorhanden.

Dabei folgt der Aufbau dem eines App Stores für Smartphones, d.h. Unity User können eigene Assets einstellen und diese kostenlos oder für einen selbst festgelegten Betrag über den Store weitergeben. Ebenso ist es möglich Bilder, Videos oder Bewertungen der Assets einzusehen.

Ein großer Vorteil des Asset Stores ist, dass alle Einsendungen vor der Freischaltung kontrolliert werden und somit garantiert sein sollte, dass jedes Asset auch problemlos in Unity funktioniert und alle relevanten Werte richtig eingestellt sind [unii]. Da externe Ressourcen häufig erhebliche Nacharbeit erfordern und die im Store erworbene Lizenz sehr kundenfreundlich ausfällt, ist der Asset Kauf auch bei höheren Beträgen lohnenswert. Denn einfach ausgedrückt können alle erworbenen Assets in unbegrenzt vielen Projekten eingesetzt werden und in Firmenstrukturen vom Arbeitgeber an die Mitarbeiter weitergegeben werden [unic].

Dabei überwiegt die Anzahl kostenpflichtiger Assets klar den kostenlosen. So herrscht bei der umfangreichsten Kategorie, den 3D-Modellen, ein Verhältnis von 237 kostenlosen zu 3506 kostenpflichtigen Assets, was weniger als 7% kostenlosen entspricht und in Abbildung 2.6 dargestellt ist (Stand 27. Juli 2013 18 Uhr) [unia].

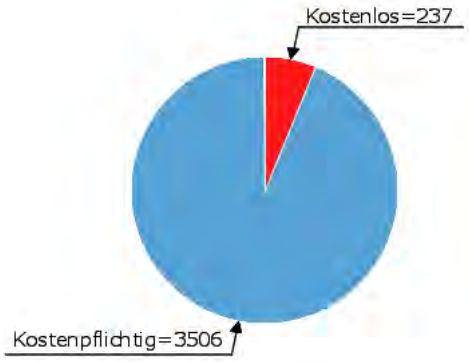

Abbildung 2.6: Asset Store Preisverteilung der 3D-Modelle

Technisch ist das Store Fenster lediglich ein eingebetteter Webbrowser, der auf die Store Website zugreift. Dabei ist die Einbettung allerdings nicht sehr ausgereift, denn die langen Ladezeiten sowohl des Fensters selbst als auch der Navigation im Store,

machen die Verwendung unnötig schwer. Dies kann etwas gemildert werden, indem die Store Webseite direkt verwendet wird.

2.1.4 Einbettung von 3D-Modellen

Selbstverständlich bietet Unity auch die Möglichkeit 3D-Modelle zu verwenden, die nicht aus dem Asset Store stammen. Dabei müssen im Optimalfall nur die Dateien des Modells in den Asset Ordner des Dateisystems kopiert oder per Drag & Drop in Unitys Asset Ordner verschoben werden, woraufhin der Import startet und das Modell einem anschließend als Asset zur Verfügung steht. Dieser Vorgang kann einige Minuten in Anspruch nehmen.

Alle 3D-Modelle von **Where are my Pirates** stammen, wenn sie nicht direkt aus dem Asset Store geladen wurden, von kostenlosen 3D-Modell Seiten. Dort funktioniert der Import nur selten problemlos, da bei der Modell Erstellung für diese Seiten, ein Unity Import wohl nicht oberste Priorität ist.

Unity unterstützt an verbreiteten 3D-Dateiformaten *.FBX* und *.OBJ* , welche direkt importiert werden können, sowie proprietäre Dateiformate wie *.MAX* von Autodesk 3ds Max oder *.BLEND* von Blender. Dabei ist zu beachten, dass beim Import von proprietären Formaten das entsprechende Programm installiert sein muss, da Unity das Modell intern in *.FBX* umwandelt, indem die Export Funktion des Ursprungprogramms genutzt wird [unda].
Nach dem erfolgreichen Import sollte zuerst der *Scale Factor* des Assets im Inspector kontrolliert werden, da dieser oft auf 0.01 gesetzt wird und deshalb nach dem Hinzufügen des Assets in die Szene, nichts zu sehen ist.

Auch kommt es häufig zu Problemen bezüglich der Texturen des Modells. Diese können zu dunkel, verschoben, falsch skaliert oder gar nicht erst vorhanden sein. Hier ist meist die einfachste Möglichkeit den Import nochmals mit einem anderen Dateiformat zu versuchen oder in Unity die Material Eigenschaften zu kontrollieren.

Sollten die Fehler weiterhin auftreten oder ein Import aus anderen Gründen nicht möglich sein, muss das Model in 3D-Modellierungssoftware überarbeitet werden, was ohne umfangreiches Vorwissen und viel Zeit allerdings nicht möglich ist.

2.1.5 Terrain Erstellung

Ein Feature von Unity, dass dieser Arbeit sehr zugute kommt, ist die *Terrain Engine* und ihre Terrain Werkzeuge (*Terrain Tools*), mit denen schnell und unkompliziert Außenszenen erstellt werden können. Dabei sind die Werkzeuge, wie in Abbildung 2.7 zu sehen, als Komponente eines *Terrain* Objekts organisiert.

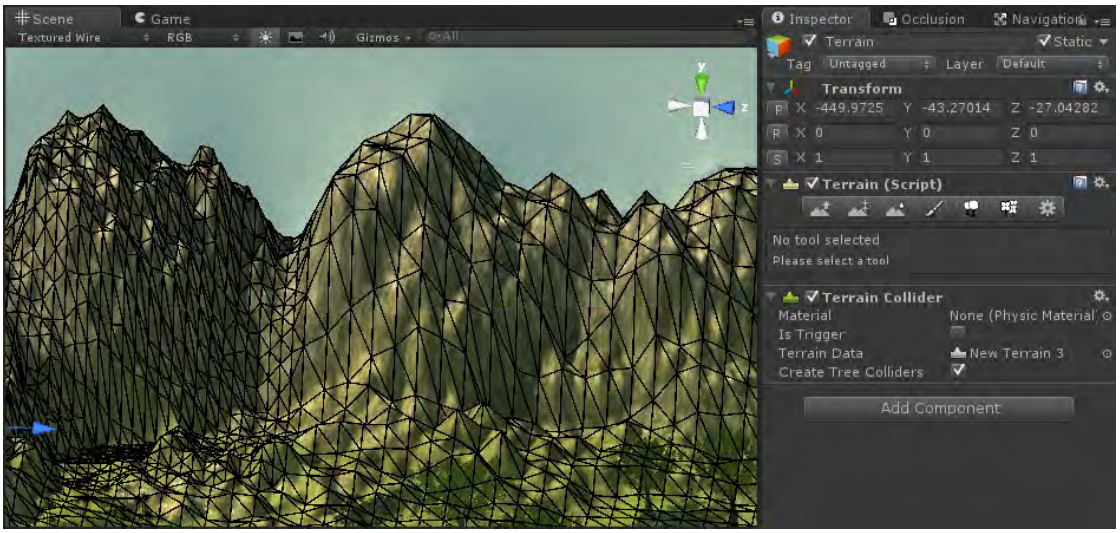

Abbildung 2.7: Terrain mit Wireframe und Terrain Werkzeugen

Mit den *Terrain Tools* können Höhe (Abschnitt 2.1.5.1), Textur (Abschnitt 2.1.5.2), sowie Verzierungen, wie z.B. Bäume oder Gras (Abschnitt 2.1.5.3) der Terrain Grundfläche bearbeitet werden. Dies wird per Pinsel (*Brush*) umgesetzt, mit dem die einzelnen Elemente in die Landschaft gemalt werden.

Der Autor dieser Arbeit empfiehlt, sich an der Reihenfolge der Terrain Werkzeuge zu orientieren. Das heißt, zuerst die Höhenlinien zu erstellen, diese zu texturieren und

anschließend mit Details zu versehen. Dabei lassen sich die einzelnen Schritte, aber in beliebiger Reihenfolge ausführen.

2.1.5.1 Höhenlinien

Wird ein neues Terrain erzeugt, ist dies erst nur eine flache, graue und quadratische Fläche. Auf dieser wird mit den drei Werkzeugen *Raise / Lower Terrain*, *Paint Height* und *Smooth Height* die Höhe bearbeitet, um Gelände von steilen Bergen bis zu flachen Hügeln zu erzeugen. Die maximale Höhe und Tiefe hängt dabei von der eingestellten *Terrain Height* ab.

Was auf diese Art nicht erzeugt werden kann, sind Höhlen oder Überhänge. Diese müssen als 3D-Modell eingefügt und manuell in das restliche Terrain eingearbeitet werden.

2.1.5.2 Texturierung

Wenn Höhen und Tiefen eingearbeitet sind, werden mittels *Paint Texture* die verschiedenen Texturen auf die Landschaft aufgetragen. Da Berge und Felswände andere Texturen benötigen als Wiesen und Felder, empfiehlt es sich für einen realistischen Look diesen Schritt erst nach der Höhenbearbeitung durchzuführen.

Als Textur können beliebige Bilder aus dem Asset Ordner verwendet werden, welche im Inspector über *Edit Texture* > *Add Texture* hinzugefügt werden. Einige grundlegende Texturen sind im *Terrain* Standardasset enthalten und über den Asset Store können viele weitere bezogen werden.
Durch die *Opacity* und vor allem die *Target Strength* Regler des Pinsels lässt sich steuern, wie stark die neue Textur die bereits vorhandene überlagert. Dadurch lassen sich weiche und harte Übergänge gleichermaßen erzeugen lassen.

Wie viele unterschiedliche Texturen auf das Terrain aufgetragen werden, hat als einer der wenigen Faktoren keinen Einfluss auf die Performance, weshalb hier völlig nach ästhetischen Gesichtspunkten vorgegangen werden kann.

2.1.5.3 Bäume, Gras & Details

Unity unterscheidet zwischen Bäumen *(Trees)*, *Gras* und *Details* beim Auftragen auf das Terrain. Dabei sind Bäume und Details 3D-Modelle mit dem Unterschied, dass Bäume sich im Wind bewegen, während Details wie Felsen statisch sind. Gras wiederum wird als 2D-Textur mit Alpha Kanal dargestellt, das sich immer in Richtung der Kamera ausrichtet und so den Anschein erweckt dreidimensional zu sein.
Bei Bäumen und Details sollte aus Performancegründen darauf geachtet werden, dass sie möglichst einfach aufgebaut sind, d.h. mit wenigen Polygonen auskommen und ihre Anzahl möglichst gering ist.

Zur Erstellung von Bäumen bringt Unity außerdem den *Tree Creator* mit, der zahlreiche unterschiedliche Baumarten erzeugen kann. Wobei sich hier ein Blick in die Dokumentation lohnt, da schlecht erstellte Bäume unnötig Performance kosten können [unij]. Dieser wird in dieser Arbeit aber nicht weiter behandelt.

Bäume, Gras & Details lassen sich wie Texturen und Höhenlinien, per Pinsel auf die Landschaft auftragen. Hilfreich ist es, den Mauszeiger im jeweiligen Modus in der Szene zu positionieren und **F** zu drücken, wodurch die Zoomstufe automatisch der Größe des Pinsels angepasst wird.
Generell ist es performanter weniger, aber dafür größere Objekte auf diese Art aufzutragen oder gleich die *Detail Resolution* des ganzen Terrains zu verringern. Unity rendert allerdings nur Objekte in Sichtweite, weshalb auch große Wälder möglich sind, sofern die Einstellungen geschickt gewählt werden.

Ob Bäume und Details eine Kollisionsabfrage besitzen, lässt sich über das als Grundlage verwendete 3D-Modell einstellen. Hat dieses eine *Collider* Komponente, haben dies auch die platzierten Objekte.

2.1.6 Physik-Engine

Einer der wichtigsten Bestandteile von Unity ist die eingebaute *PhysX* Physik-Engine von *NVIDIA*, mit deren Hilfe Kollisionen festgestellt sowie Gravitation, Reibung oder

Bounciness[2] eingestellt und ausgewertet werden können [unif].
Dabei wird in dieser Diplomarbeit nur die Kollisionsabfrage verwendet, die mithilfe der im Folgenden vorgestellten *Rigidbody* (Starrkörper) und *Collider* Komponenten funktioniert.

2.1.6.1 Rigidbody

Ein *Rigidbody* ist eine Komponente, die Objekte physikalisch manipulierbar, d.h. für Bewegungsenergie empfänglich macht, indem sie im Spiel mit anderen Physikobjekten interagiert oder ihr Bewegungsenergie hinzugefügt wird.
So kann z.B. das *Standard Asset Character Controller*, mit einem Skript aus der offiziellen Dokumentation dazu verwendet werden, Rigidbodies zu verschieben [undd].

Über den Rigidbody kann eingestellt werden, welche Masse ein Objekt hat und ob es von Gravitation beeinflusst wird. Weiterhin kann z.B. der *Drag* eingestellt werden, was sich auf den Luftwiderstand beim Fallen bezieht. Ein Luftballon hat beispielsweise mehr *Drag* als eine Holzkiste.

Wichtig ist zudem noch der `IsKinematic` Wert, denn ist dieser auf `true` gesetzt wird der Rigidbody nicht mehr von Bewegungsenergie, Gravitation oder Kollisionen beeinflusst und kann nur noch per Skript gesteuert werden.
Ein Rigidbody benötigt dabei immer auch einen Collider für die Kollisionsabfrage.

2.1.6.2 Collider

Ein *Collider* dient zur Kollisionsabfrage mit anderen Objekten, die ebenfalls einen Collider besitzen, da auch nur dann eine Kollision erfolgen kann. Objekte mit Collider, aber ohne Rigidbody, nennt man *Static Collider*. In diese Gruppe fällt die gesamte unbewegliche Level Architektur wie Häuser, Bäume oder Felsen.

Aus Performancegründen sollte bei *Static Collidern* vermieden werden, zur Laufzeit die *Transform* Komponente zu verändern und sie sollten nach Möglichkeit auch nicht aktiviert oder deaktiviert werden, da dies eine Neuberechnung in PhysX nach sich

[2] Rückprallelastizität, wie stark die Oberfläche zurück federt

ziehen kann. Weil dies außer zu einem enormen Performance-Einbruch auch zu fehlerhaften Physikberechnungen führen kann, sollte eine Veränderung nur über Objekte mit Rigidbody vorgenommen werden [unik, unig].

Collider enthalten außerdem den `IsTrigger` Wert, über den per Skript Zugriff auf Kollisionsinformationen erfolgt, indem z.B. das `OnTriggerEnter()` Event ausgelöst wird. Allerdings ist es nicht intuitiv klar, auf welchem Objekt `IsTrigger` gesetzt werden, welches einen Rigidbody enthalten und ob dieser kinematisch sein muss oder nicht, um ein bestimmtes Event auszulösen. Abbildung 2.8 verdeutlicht die Komplexität dieses Sachverhalts.

Collision detection occurs and messages are sent upon collision

	Static Collider	Rigidbody Collider	Kinematic Rigidbody Collider	Static Trigger Collider	Rigidbody Trigger Collider	Kinematic Rigidbody Trigger Collider
Static Collider		Y				
Rigidbody Collider	Y	Y	Y			
Kinematic Rigidbody Collider		Y				
Static Trigger Collider						
Rigidbody Trigger Collider						
Kinematic Rigidbody Trigger Collider						

Trigger messages are sent upon collision

	Static Collider	Rigidbody Collider	Kinematic Rigidbody Collider	Static Trigger Collider	Rigidbody Trigger Collider	Kinematic Rigidbody Trigger Collider
Static Collider					Y	Y
Rigidbody Collider				Y	Y	Y
Kinematic Rigidbody Collider				Y	Y	Y
Static Trigger Collider		Y	Y		Y	Y
Rigidbody Trigger Collider	Y	Y	Y	Y	Y	Y
Kinematic Rigidbody Trigger Collider	Y	Y	Y	Y	Y	Y

Abbildung 2.8: Übersicht wann Kollisionen ausgelöst oder erkannt werden [undb]

Außerdem kann Collidern ein *Physics Material* zugewiesen werden, über das eingangs genannte Reibung oder Bounciness eingestellt werden kann, um Oberflächenverhalten, wie Eis oder Gummi, zu erzeugen.

2.1.7 Scripting in Unitty

Sobald der Spieler mit der Spielwelt interagieren soll, werden Skripte benötigt. In Unity stehen dem Entwickler dazu *UnityScript*, *C#* oder *Boo* zur Verfügung, welche im mitgelieferten *MonoDevelop* oder einem externen Editor bearbeitet werden können.

2.1.7.1 UnityScript

Eine Besonderheit an Unity ist *UnityScript (US)*, welches eine sehr stark an *JavaScript (JS)* angelehnte Skriptsprache ist, die JS unter anderem um ein objektorientiertes (OO) Klassenkonzept samt Datenkapselung mittels `private/public/protected`, typisierter Variablen zusätzlich zur dynamischen Typisierung und einer OO-Vererbung erweitert.

In Beispiel 2.1 sind Datentypen und Vererbung in UnityScript zu sehen. Da UnityScript Dateien auf .js Enden wird umgangssprachlich trotzdem oft von JavaScript gesprochen.

Listing 2.1: Vererbung und Typisierung in UnityScript

```
1  class ParentClass extends MonoBehaviour {
2      public function Start() {
3          Debug.Log("ParentClass");
4      }
5  }
6  public class SubClass extends ParentClass {
7      public override function Start() {
8          super.Start();
9          var TestVariable : String = "foo";
10         Debug.Log("SubClass : " + TestVariable);
11     }
12 }
13 // Zuerst wird "ParentClass" und dann "SubClass : foo" ausgegeben
```

Grundsätzlich ist es dem Entwickler überlassen in welcher der drei Sprachen entwickelt wird, allerdings gibt es deutlich mehr Tutorials oder Skriptbausteine für C# und UnityScript.

In dieser Arbeit wurden die Skripte anfangs ausschließlich in UnityScript geschrieben, da der Autor JS Vorkenntnisse hatte, womit allerdings nicht direkt auf die C# Leap Motion Bibliotheken für Unity (siehe Abschnitt 2.3.2) zugegriffen werden kann, weshalb fast alle Skripte zu C# geändert wurden.

UnityScript und C# können zwar gemeinsam genutzt und Variablen unter den Skripten ausgetauscht werden, allerdings wird C#-Code vor US-Code kompiliert, weswegen

US-Skripte zwar auf C#-Variablen zugreifen können, aber nicht umgekehrt. Dies kann umgangen werden, indem die C#-Skripte in einen anderen Ordner verschoben werden, der später kompiliert wird [unih].

Das Grundproblem bleibt aber, dass Skripte immer nur in einer Richtung auf Skripte der anderen Sprache Zugriff haben, weshalb alle Skripte, die miteinander interagieren, in der gleichen Sprache geschrieben sein sollten.

2.1.7.2 Best Practice

Da Unity Variablen, die `public` (aber nicht `static`) sind, über den Inspector im Editor zugänglich macht, sollte zur besseren Lesbarkeit eine Benennung mit CamelCase statt Unterstrichen erfolgen (`VariablenName` statt `Variablen_Name`). Unity wandelt dies in der Editor Darstellung um, so wird aus `myFooBar` `My Foo Bar`, aber `my_foo_bar` bleibt gleich.

In UnityScript Dateien sollte `#Pragma strict` in der ersten Zeile stehen, was eine statische Typisierung erzwingt, die wiederum der Performance zugute kommt und bei *iOS* als Zielplattform sogar benötigt wird [unie].

2.1.7.3 Performance

Während die Eigenheiten der jeweiligen Programmiersprachen für die Performance sehr wichtig sind, gibt es einige Eigenheiten von Unity selbst, die beachtet werden sollten.

So enthält eine neue Skriptdatei, egal ob UnityScript, C# oder Boo, standardmäßig eine `Start()`- und `Update()`-Funktion. Dabei wird `Start()` einmal beim Szenenstart aufgerufen und `Update()` in jedem Frame. Somit sind selbst leere `Update()`-Funktionen in der Masse ein Performance-Problem und sollten entfernt werden.

Generell empfiehlt es sich pro Szene so wenig `Update()`- und `Start()`-Aufrufe wie möglich zu haben oder diese in einem zentralen Skript zu kapseln, dessen `Update()` Methode die Methoden der anderen Skripte aufruft.

Außerdem sollte eine Verwendung von `Instantiate()` und `Destroy()`, zum Anlegen und Zerstören von mehrmals genutzten Objekten vermieden werden, da dies die *Garbage Collection* auslastet [undc]. Stattdessen kann mittels *Pooling*[3] und Ein- und Ausblenden der Objekte Rechenzeit gespart werden.

Zwar können die Qualitätseinstellungen bei aktiviertem Konfigurationsdialog, wie in Abbildung 2.9 gezeigt, noch vor dem Spielstart heruntergesetzt werden, allerdings kann dies schlechte Skripte nur bedingt ausgleichen.

Weitere Performance Optimierungen sind in den jeweiligen Implementierungskapiteln beschrieben.

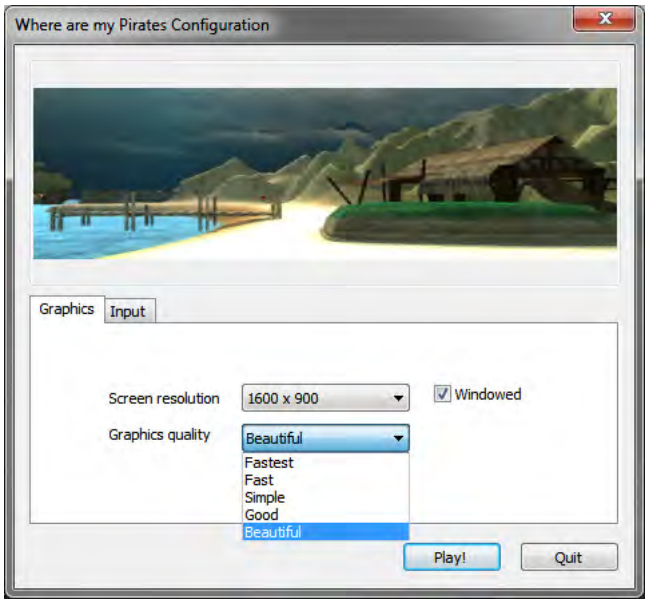

Abbildung 2.9: Der Konfigurationsdialog von **Where are my Pirates**

2.2 Leap Motion

Der *Leap Motion Controller* der Firma *Leap Motion*, stellt eine gestenbasierte Benutzerschnittstelle für Computer zur Verfügung. Dabei ist die Leap ein USB-Gerät von der

[3] Ein *Pool* beinhaltet bereits initialisierte Ressourcen

Größe eines Feuerzeugs, welches die Mensch-Computer-Interaktion um Finger- und Handgesten erweitert.

Abbildung 2.10 zeigt eine Visualisierung der Handbewegungen, wobei die Leap am unteren Bildrand zu sehen ist.

Abbildung 2.10: Bewegungssteuerung über den Leap Motion Controller [leaf]

Das Besondere an der Leap ist die hohe Auflösung und Präzision der Sensoren, sowie die eigens entwickelten Algorithmen, welche erst die Möglichkeit bieten Finger- und Handbewegungen zu erfassen. Die Technologie ist am ehesten mit Microsofts *Kinect* vergleichbar, wobei die Leap eine deutlich höhere Auflösung bei kleinerem Sichtfeld hat, weswegen sie sich für die Erfassung der Finger, aber nicht des ganzen Körpers eignet, was für die Kinect umgekehrt gilt. Dabei kann die Leap bis zu 10 Finger mit Händen und als *Tools* bezeichnete Hilfsmittel wie Stifte erfassen [leae].

Es ist denkbar, dass ein PC mit Leap in Zukunft wie eine Art berührungsloser Touchscreen bedient werden kann, mit den bekannten Zoom- und Scroll-Gesten. Die Funktionalität hierfür ist zwar bereits in den Leap Treibern enthalten, allerdings noch nicht ausgereift genug für einen Einsatz in der Praxis.

Zudem könnten mit der Leap auch Geräte und Prozesse gesteuert werden, die eine kontaktlose Bedienung benötigen, aber mangels alternativen Steuerungskonzepten bisher nicht realisiert werden konnten. Denkbar sind Einsatzszenarien im medizinischen Bereich oder in Bereichen, in denen schmutzige Hände Touchscreens unbenutzbar machen.

Festzuhalten bleibt aber, dass eine berührungslose Steuerung von Geräten mit Finger- und Handgesten in der vorliegenden Genauigkeit, ein völlig neuartiges Konzept darstellt. So neu, dass die Leap Motion zu Beginn dieser Arbeit noch nicht frei auf dem Markt erhältlich war, sondern nur durch eine erfolgreiche Bewerbung des Autors bei Leap Motions Entwicklerprogramm erhalten wurde. Dadurch ist ein Mangel an Erfahrungswerten und Konzepten, für diese Art der Eingabe in der Entwickler-Community vorhanden und es kann in dieser Arbeit nicht auf bewährte Methoden zurückgegriffen werden.

2.2.1 Hardware Aufbau

Auf Hardware Ebene beinhaltet die Leap drei Infrarot-LEDs und zwei CMOS Bildsensoren, mit denen Tiefeninformationen erfasst werden, welche wiederum in die entsprechenden Gesten und das Finger-/Tool-/Handmodell umgerechnet werden.
Dabei deckt die Leap einen Bereich von ca. 60 cm zu jeder Seite, nach oben und in die Tiefe ab, wie in Abbildung 2.11 zu sehen ist.

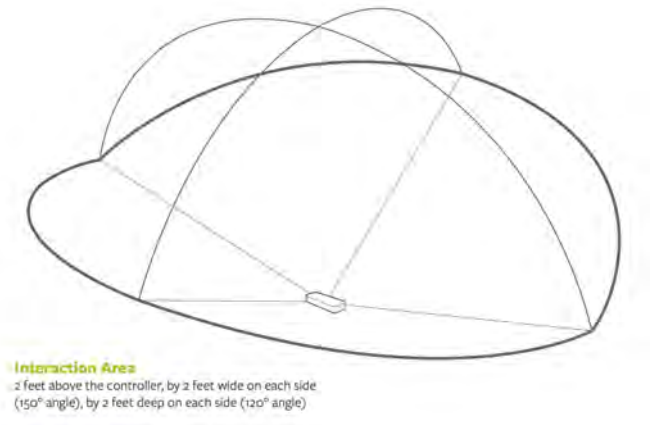

Abbildung 2.11: Der Interaktionsbereich der Leap Motion [leak]

Aufgrund der optischen Datenerfassung, ist die Leap anfällig für Schmutz und Kratzer auf der Oberfläche, da dies die Messung verschlechtert. Wenn die Leap eine Verschmutzung feststellt, löst sie die *Smudge detected!* Meldung aus und die Genauigkeit verringert sich, wobei diese Meldung beim Autor häufig ausgelöst wurde, ohne dass die Oberfläche tatsächlich verschmutzt war.

Außerdem ist für genaue Messungen eine separate Kalibrierung pro Monitor und Standort notwendig, da die Leap Finger- zu Monitorkoordinaten umrechnet und dies sowohl von der Größe des Monitors als auch von der Position der Leap relativ zum Monitor abhängt. Nach der Kalibrierung sollte die Leap möglichst nicht mehr neu positioniert werden, da dies die Genauigkeit wieder verschlechtern kann.
Grundsätzlich kann die Leap auch ohne Kalibrierung verwendet werden, was für Anwendung die keine hohe Genauigkeit erfordern ausreichend ist. Allerdings hat der Autor die Erfahrung gemacht, dass sich das Nutzungserlebnis durch eine Kalibrierung deutlich steigern lässt.

Eine weitere Fehlerquelle sind alle Arten von direkter Lichteinstrahlung, durch Licht mit hohem Infrarot Anteil [leam]. Dies stört ebenfalls die Messung, was zu einer deutlichen Reduzierung der Genauigkeit führt. Besonders direkte Sonneneinstrahlung ist zu vermeiden.

Wie genau die Leap die Daten in das interne Hand- und Fingermodell aus Abbildung 2.12 umrechnet, ist bisher nicht bekannt. Sie scheint auf ein stereoskopisches Verfahren zu setzen, da sie bei Abdeckung eines Bildsensors nicht mehr einsatzfähig ist [spaa]. Die Umrechnung der Daten scheint außerdem vollständig auf dem PC stattzufinden, da die Leap keine Chips, hat um ihrerseits Berechnungen oder Optimierungen vorzunehmen, sondern nur das Nötigste enthält um die Daten per USB weiterzuleiten [spab].

Da die Leap hardwareseitig somit recht einfach aufgebaut ist und die Datenverarbeitung vollständig über den PC erfolgt, scheint ein Großteil der Technik in den entwickelten Algorithmen zu stecken, was zu Pressemitteilungen von Leap Motion passt, in denen die Software als *heart of the Leap* bezeichnet wird [leae].
Dies hat zur Folge, dass ein leistungsfähiger PC, mit laut Herstellerangaben mindestens 2 GB RAM sowie entweder einem *AMD Phenom II* oder einem *Intel Core i3/i5/i7* Prozessor

benötigt wird [lead]. Dafür können aber auch Verbesserungen der Performance und Zuverlässigkeit oder eine Erweiterung des Funktionsumfangs per Software Update nachgereicht werden.

2.2.2 Software Aufbau

Die Software für den Endverbraucher enthält neben dem Treiber, noch das *Leap Motion Control Panel* und den in Abbildung 2.12 gezeigten *Leap Motion Visualizer*. Dabei bündelt das Control Panel Informationen und Tools zur Leap, wie den Visualizer, den Leap App Store *Airspace* oder die Kalibrierungswerkzeuge, kümmert sich um Updates und stellt Hilfen zur Fehlerbehandlung bereit.

Zur Visualisierung und Fehlersuche dient genannter *Leap Motion Visualizer*, welcher in einer optisch verspielten Variante für Endkunden und als *Diagnostic Visualizer* für Entwickler vorhanden ist. Dabei werden in beiden Varianten die von der Leap erfassten Bewegungsdaten optisch als Finger- und Handmodell dargestellt.

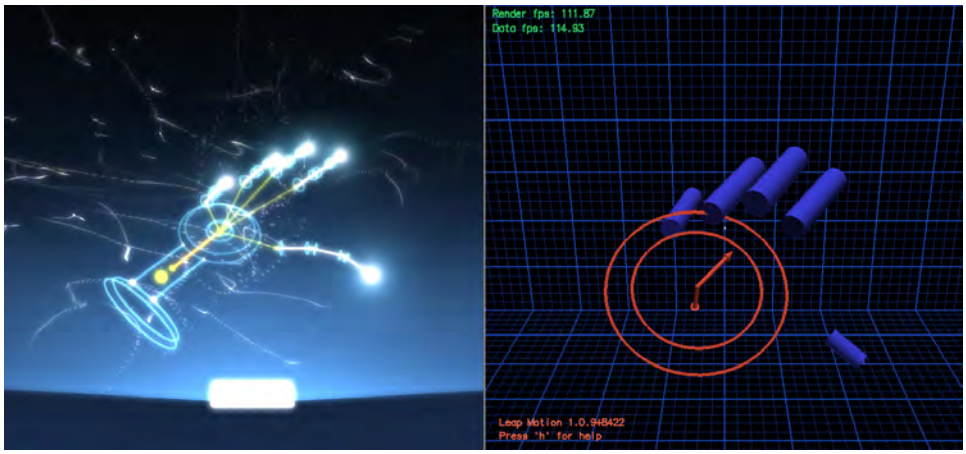

Abbildung 2.12: Links die Endkundenvariante und rechts der Diagnostic Visualizer

Die Leap stellt Bibliotheken in C++, C#, Java, Python und Javascript zur Verfügung und bringt eine angepasste Bibliothek für Unity mit [leah]. Da in dieser Diplomarbeit ausschließlich Unity eingesetzt wird und die Leap Bibliotheken dafür in C# geschrieben ist, liegt hier auch der weitere Fokus.

Die Leap Motion Software arbeitet dabei nicht mit einem *event*-, sondern *frame*-basierten Ansatz, weshalb Parameter, wie Beschleunigung, immer ausgehend von einem vergangenen Frame berechnet werden und die Framerate einen maßgeblichen Einfluss auf die Präzision der Leap hat [leai].

2.2.3 Verfügbare Daten

Möchte ein Entwickler die Leap Motion einsetzen, stellt sich die Frage, auf welche Daten dieser Zugriff hat und wie sie auszuwerten sind. Dabei arbeitet die Leap mit einem abstrahierten Datenmodell, bei dem zwischen Fingern, Händen und Tools unterschieden wird und dafür jeweils Informationen, wie Position und Rotation, verfügbar sind.

Dies befreit den Entwickler in der Theorie von manueller Auswertung und Optimierung, da direkt das Handmodell eingesetzt werden kann. Allerdings ist die Zuverlässigkeit der Leap Daten noch nicht so hoch, dass diese wirklich ohne weitere Verarbeitung eingesetzt werden können. So wurden beim Erstellen dieser Arbeit immer wieder Finger von der Leap an Stellen ausgewertet, an denen sich nichts befand.
Auf der anderen Seite hat ein Entwickler keinen Zugriff auf eine Punktwolke[4] um die Leap, ähnlich der Kinect, als 3D-Scanner einzusetzen oder die Auswertung der Fingerpositionen manuell vorzunehmen. Es ist nicht bekannt, ob die Leap intern überhaupt Punktwolken erzeugt oder ein anderes Verfahren einsetzt.

2.2.4 Hand-, Finger- und Toolmodell

Zugriff auf die Leap Daten erhält ein Entwickler über das `Frame` Objekt, das Listen von Händen, Fingern und Tools enthält, wobei Tools und Finger zusammengefasst auch in der `PointableList` verfügbar sind. Dabei hat ein `Hand` Objekt wiederum Listen der Finger- und Tool-Objekte dieser Hand. So kann der Entwickler je nach Bedarf das geeignetste Objekt direkt über den `Frame` abrufen oder die Daten hierarchisch auswerten.

[4] Eine Punktwolke bezeichnet eine Liste von 3D-Koordinaten in der 3D-Grafik.

2.2.4.1 Handmodell

Das Handmodell enthält Daten über Position, Eigenschaften sowie Bewegung der Hand und enthält eine Liste ihrer Finger und Tools [leai].
Um Greifbewegungen auszuwerten, hat das Handmodell aus Abbildung 2.12, die beiden Eigenschaften `SphereRadius` und `SphereCenter`. Dafür projiziert die Leap eine Kugel in die Handfläche, die diese ausfüllt. So kann über die Veränderung des `SphereRadius` festgestellt werden, ob sich die Hand schließt oder öffnet. Dies könnte theoretisch auch über die Fingerpositionen festgestellt werden, was sich aber, wie im nächsten Abschnitt erläutert wird, als schwierig erweist.

2.2.4.2 Finger- und Toolmodell

Finger und Tools haben grundsätzlich die gleichen Eigenschaften und unterscheiden sich nur darin, dass ein Tool länger, dünner und gerader als ein Finger ist [leai]. Grundsätzlich können alle Gegenstände mit diesen Eigenschaften als Tool eingesetzt werden.
Dabei sind für Finger und Tools, Parameter wie Länge ab der Hand, durchschnittliche Breite und Position der Spitze verfügbar. Es sind aber beispielsweise keine Daten über die Position der Knöchel oder Fingerkrümmung verfügbar, obwohl die Darstellung im Visualizer, wie in Abbildung 2.13 gezeigt, den Schluss zu lässt, dass die Leap diese Daten intern anlegt und verwertet.

Abbildung 2.13: Das Hand-, Finger- und Toolmodell im Visualizer

Anzumerken ist, dass die Leap einzelne Finger nur dann erkennen kann, wenn sich diese weit genug auseinander befinden und einander nicht verdecken. So werden Finger nicht mehr erkannt, wenn sie unter dem Handballen oder eng aneinander liegen.

2.2.5 Gesten

Neben der Auswertung einzelner Finger, Tools oder Hände ist die Leap in der Lage Gesten zu erkennen.
Umgangssprachlich könnte jede Fingerbewegung als Geste bezeichnet werden, Leap Motion bezieht sich damit allerdings nur die Bewegungsmuster aus Abbildung 2.14 [leai].

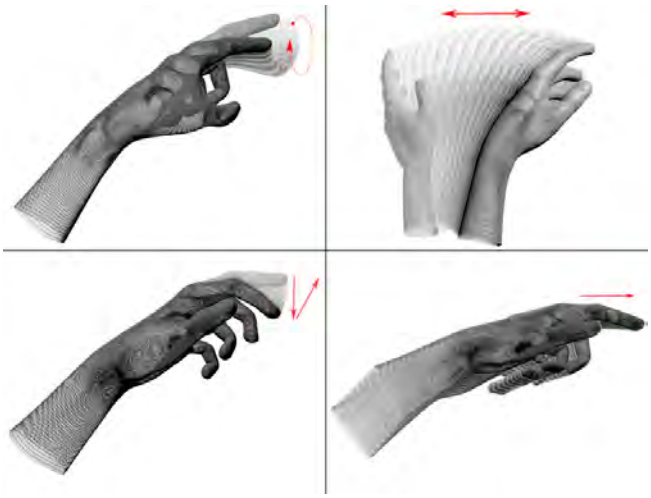

Abbildung 2.14: *Circle* links oben, *Swipe* rechts oben, *Key Tap* links unten und *Screen Tap* rechts unten [leai]

- **Circle**: Bei der ein Finger kreisförmig bewegt wird.
- **Swipe**: Bei der Finger gleichmäßig in eine beliebige Richtung bewegt werden.
- **Key Tap**: Bei der eine schnelle Abwärtsbewegung eines Fingers ähnlich eines Tastaturanschlages erkannt wird.
- **Screen Tap**: Bei der es sich um eine schnelle Vorwärtsbewegung eines Fingers handelt.

Da Gesten erst während der Entwicklung dieser Arbeit durch ein Software Update zur Verfügung gestellt wurden, ist langfristig mit weiteren zu rechnen.
Dabei wurde in dieser Arbeit nur die *Swipe* Geste verwendet, welche sich allerdings als unzuverlässig herausgestellt hat, da die Erkennungsqualität stark schwankt.

2.2.6 Absolute vs. Relative Steuerung

Mangels bestehender Sprachkonventionen, führt der Autor in diesem Abschnitt die beiden Begriffe *absolute Steuerung* und *relative Steuerung* im Zusammenhang mit der Leap Motion ein.
Grundsätzlich stellt sich bei einem Steuerungskonzept mittels Leap die Frage, ob die Bewegungen absolut oder relativ genutzt werden.

Dabei handelt es sich nach Meinung des Autors um eine *absolute Steuerung*, wenn die Fingerposition vor dem Bildschirm der Zeigerposition auf diesem entspricht und eine Fingerbewegung im Raum, den Zeiger um die gleiche Strecke bewegt. Somit findet, bei korrekt kalibrierter Leap, eine 1:1 Übertragung der Fingerbewegungen statt. Dies ist mit dem Zeichenprogramm eines Touchscreens vergleichbar.

Von einer *relativen Steuerung* wird in dieser Arbeit gesprochen, wenn die Bewegung keine direkte Entsprechung auf dem Bildschirm hat und stattdessen eine Änderung relativ zur zurückgelegten Strecke statt findet. Dies ist vergleichbar mit einer Zeigersteuerung per Maus.

Was hierbei bedacht werden sollte ist, dass die absolute Steuerung eine hohe Präzision erfordert, da Ruckler und Aussetzer viel stärker auffallen. Deswegen sollte sich die Leap z.B. nicht im *Smudge Detected!* Modus befinden. Eine relative Steuerung ist hier wesentlich fehlertoleranter.

2.3 Leap Motion mit Unity nutzen

Da es für diese Arbeit notwendig ist, die Leap mit Unity zu nutzen, wird in diesem Kapitel auf deren Zusammenspiel eingegangen. Dabei hat sich die Integration der

Leap Bibliotheken in das Projekt als recht einfach herausgestellt. Wobei es für einen reibungslosen Ablauf einige Voraussetzungen gibt, die im Folgenden erläutert werden.

So konnten die von Leap Motion bereitgestellten Treiber vormals nur mit der kostenpflichtigen Unity Pro Version verwendet werden, da diese Unitys *Native Code Plugins* System nutzen. Dieses ist in der kostenlosen Version allerdings nicht verfügbar, weswegen in dieser Arbeit die Pro Version eingesetzt wird.
Inzwischen hat die Leap Community Möglichkeiten gefunden, die Leap Bibliotheken auch mit der kostenlosen Unity Variante zu verwenden, was im Folgenden ebenso erläutert wird.

2.3.1 Leap Bibliotheken in Unity Projekt einfügen

Nach dem Anlegen des Projekts in Unity, sollten die Leap Treiber auf dem Rechner installiert sein und die Bibliotheken in den richtigen Ordner kopiert werden. Dieser unterscheidet sich je nachdem, ob Unity Pro oder Standard verwendet wird, sowie je nach Betriebssystem. Dies wird in diesem Abschnitt erläutert.

2.3.1.1 Die Leap in Unity Pro nutzen

Die folgenden Bibliotheken aus dem *Leap Motion SDK* müssen in den `[PROJEKTPFAD]/Assets/Plugins` Ordner kopiert werden:

Windows 32-bit

- `LeapSDK/lib/UnityAssets/Plugins/LeapCSharp.NET3.5.dll`
- `LeapSDK/lib/UnityAssets/Plugins/LeapCSharp.dll`
- `LeapSDK/lib/UnityAssets/Plugins/Leap.dll`

32- and 64-bit Mac OS:

- `LeapSDK/lib/UnityAssets/Plugins/LeapCSharp.NET3.5.dll`
- `LeapSDK/lib/UnityAssets/Plugins/LeapCSharp.bundle`

Da es Probleme mit den 64-bit Bibliotheken unter Windows geben kann, sollten auch hier die 32-bit Versionen eingesetzt werden [leaj].

Falls plattformübergreifend für Mac und Windows entwickelt wird, können auch direkt alle Bibliotheken in das entsprechende Verzeichnis abgelegt werden.
Da an dieser Stelle das Plugin System von Unity Pro zum Einsatz kommt, kann das Spiel direkt kompiliert werden, woraufhin Unity sich um die korrekte Einbettung der Bibliotheken in den erzeugten Ordner kümmert.

2.3.1.2 Die Leap in Unity Free nutzen

Wird Unity mit der Standard Lizenz verwendet, sollte das Projekt folgenden Aufbau aufweisen, damit die Leap Bibliotheken in Unity verwendet werden können [leag]:

Windows

```
/SampleProject/
    /Assets/
        LeapCSharp.NET3.5.dll
        Scene.unity
        /Scripts/
            LeapMotionControl.cs
    /Library/
    /ProjectSettings/
    Leap.dll
    LeapCSharp.dll
    msvcp100.dll
    msvcr100.dll
```

OS X

```
/SampleProject/
    /Assets/
        LeapCSharp.NET3.5.dll
```

```
        Scene.unity
        Scripts/
            /LeapControl.cs
    /Library/
    /ProjectSettings/
    libLeap.dylib
    libLeapCSharp.dylib
```

Nach der Kompilierung des Projektes müssen die Bibliotheken hier manuell in das Zielverzeichnis kopiert werden:

Windows

```
/SampleProject/
    Leap.dll
    LeapCSharp.dll
    msvcp100.dll
    msvcr100.dll
    /SampleProject/
    SampleProject.exe
```

OS X

```
/SampleProject/
    libLeap.dylib
    libLeapCSharp.dylib
    SampleProject.app
```

2.3.2 Leap Auswertung in Unity

Wie bereits erläutert, wird in Unity mit C# auf die Leap Motion Bibliotheken zugegriffen. Soll in anderen Sprachen wie UnityScript entwickelt werden, ist es nötig sich eine eigene C# Klasse zu schreiben, die als Wrapper für den Leap Zugriff dient.

Da eine gleichzeitige Verwendung mehrerer Sprachen, wie in Abschnitt 2.1.7.1 erwähnt, einige Probleme mit sich bringt, ist es am einfachsten alle Skripte die Zugriff auf die Leap benötigen in C# zu verfassen.

Um die Leap Daten zu erhalten wird die `Leap.Controller` Klasse genutzt, über die, wie in Beispiel 2.2 gezeigt, auf die einzelnen Frames und damit die weiteren Objekte zugegriffen wird. Außerdem ist ersichtlich, dass der Zugriff auf die Bibliothek in C# objektorientiert erfolgt.

Listing 2.2: Unity und Leap Beispielprogramm

```
1  using UnityEngine;
2  using System.Collections;
3  using Leap;
4
5  public class Leap : MonoBehaviour {
6      Controller controller;
7
8      void Start() {
9          // Das Leap Controller Objekt
10         // ist die Schnittstelle zur Leap
11         controller = new Controller();
12     }
13
14     void Update(){
15         if(controller != null){
16             Frame frame = controller.Frame();
17             // Hat der Frame pointables
18             // wie Finger oder Tools?
19             if (!frame.Pointables.Empty){
20                 // Den ersten Pointable auslesen
21                 pointable = frame.Pointables[0];
22
23                 // Die Pointable Daten verwerten
24             }
25         }
26     }
27 }
```

Als Optimierung könnte in diesem Beispiel noch der letzte Frame gespeichert werden, um zu große Sprünge bei den Daten z.B. mit einer Mittelwertfunktion auszugleichen.

2.4 Psychologischer Hintergrund

Da es das Ziel dieser Arbeit ist, ein *Serious Game* zu entwickeln, wird in diesem Kapitel zunächst kurz auf den *Serious Game* Begriff und anschließend ausführlicher auf die berücksichtigten Psychologischen Hintergründe eingegangen.

Dabei wird in Abschnitt 2.4.2 der *Flow* bzw. *GameFlow* Begriff erläutert, anhand dessen sich positive Eigenschaften eines Spieles ableiten lassen.

In Abschnitt 2.4.3 folgen die Grundlagen der *exekutiven Funktionen*, die durch **Where are my Pirates** hauptsächlich trainiert werden sollen.

Anschließend wird in Abschnitt 2.4.4 noch der *Trail Making Test* erläutert, welcher die Grundlage für eines der beiden Teilspiele dieser Arbeit bildet.

2.4.1 Serious Games

Der Begriff Serious Game wird auf Computerspiele angewandt, deren Ziele über die reine Unterhaltung der Spieler hinausgehen, dabei aber weiterhin Elemente und Mechanismen eines Spieles einsetzen. Spieler sollen durch ein Serious Game neues Wissen lernen und abstraktes Denken sowie Problemlösungsfähigkeiten trainieren [Cha10]. Deswegen können Lernspiele auch als Teilbereich der Serious Games gesehen werden.

2.4.2 Flow-Theorie und GameFlow

Ein Spieler wird vor allem durch den wahrgenommenen Spaß und nur selten durch externe Belohnungen, wie Geld, zum Weiterspielen motiviert. Dabei werden dem Spieler durch spielinterne Mechanismen und Belohnungen, positive Erlebnisse und Spielspaß ermöglicht. Nun ist es allerdings schwierig einen so weitläufigen Begriff wie (Spiel)Spaß messbar zu machen und in ein Modell umzusetzen.

Dies ist leicht ersichtlich, da ein Spiel in verschiedenen Genres verwurzelt sein kann und verschiedene Spieler andere Vorlieben haben.

Ein Ansatz, der in dieser Arbeit berücksichtigt und an dieser Stelle vorgestellt wird, ist der *GameFlow*. Dabei wird zuerst die Verwendung des Flow Begriffs allgemein und anschließend die des GameFlows und seiner Kernelemente im Speziellen erläutert.

2.4.2.1 Der Flow Begriff

Flow bezeichnet in der Psychologie das Gefühl des vollständigen Aufgehens in einer Tätigkeit, nur durch innere Motivation, welches als sehr positiv wahr genommen wird [Csi10]. Dabei geht Flow über Gefühle, wie Spaß oder Freude, hinaus und beschreibt eine längere befriedigende Beschäftigung mit einer Tätigkeit.
Flow entsteht im Grunde genommen, wenn die folgenden fünf Faktoren gemeinsam eintreten [Huh04]:

- Es werden selbstgesteckte und **klare Ziele** verfolgt.
- Es erfolgt kontinuierliche **Rückmeldung**, ob die Tätigkeit gut oder schlecht ausgeführt wird.
- Der Grad der **Herausforderung** bewegt sich zwischen Über- und Unterforderung.
- Es herrscht das Gefühl vor, **Kontrolle** über den Erfolg zu haben.
- Die gesamte **Konzentration** wird intensiv auf ein begrenztes Feld ausgerichtet.

Die Flow-Theorie wurde von *Mihaly Csikszentmihalyi* in den 70er-Jahren entwickelt und durch Befragungen von unterschiedlichen Personengruppen, wie Kletterern, Schachspielern, Komponisten, Tänzerinnen und Basketballspielern, untersucht [Csi10].

Es ist bereits hier ersichtlich, dass mit dieser Theorie sehr gut intensive Spielerfahrungen beschrieben werden können und sich aus diesen Faktoren, Regeln für das Game-Design ableiten lassen. Dabei bildet der Flow allerdings nur die Grundlage für den im weiteren Verlauf dieser Arbeit referenzierten GameFlow.

2.4.2.2 Der GameFlow Begriff

Der GameFlow von *Peta Wyeth* und *Penelope Sweetser* baut auf dem Flow-Modell auf und adaptiert dessen Mechanismen für Spiele. Dabei beinhaltet das GameFlow Modells die folgenden Kernfaktoren [SW05]:

- **Concentration**: Es besteht die Fähigkeit sich auf die Aufgabe zu **konzentrieren**.
- **Challenge** + **Skills**: Die **Fähigkeiten** sollten zu den **Herausforderungen** passen und beide müssen über einem bestimmtem Schwellenwert liegen.
- **Control**: Es muss das Gefühl der **Kontrolle** über den Verlauf des Spiels geben.
- **Clear Goals**: Die Aufgabe hat **klare Ziele**.
- **Feedback**: Die Aufgabe bietet direkte **Rückmeldung**.
- **Immersion**: Tiefes und müheloses **eintauchen**, ohne sich allzu sehr für sonstige Belange zu interessieren.

Als achter Faktor wird noch die soziale Interaktion erwähnt, die allerdings nicht direkt zum Flow gehört und in dieser Arbeit keine weitere Rolle spielt, da keine Mehrspieler-Unterstützung implementiert wird.

So werden die Aspekte des Flows und insbesondere des GameFlows während der Konzeption und Entwicklung von **Where are my Pirates** berücksichtigt, was in Kapitel 3 und Kapitel 4 detailliert dargestellt wird.

2.4.3 Exekutive Funktionen

Exekutive Funktionen bezeichnen höhere kognitive Prozesse, die Denken und Handeln kontrollieren und überwachen, um eine flexible Anpassung an neue und komplexe Aufgaben zu ermöglichen [RNMR10].

So sind exekutive Prozesse an der Lösung bisher unbekannter Probleme, sowie der Verhaltensmodifikation bei veränderten Informationen über die Umwelt beteiligt und dienen der Entwicklung von zielorientierten Strategien. Sie dienen also der schnellen Anpassung an neue und unerwartete Situationen [STK07].

Dabei sind die folgenden drei Faktoren für die Verwendung exekutiver Funktionen entscheidend [RNMR10]:

- **Shifting/Switching**: Aufgaben-, Aufmerksamkeits- oder Strategiewechsel.
- **Updating**: Anpassung und Überwachung von Prozessen des Arbeitsgedächtnis[5].
- **Inhibition**: Unterdrückung vorschneller, dominanter oder automatischer Reaktionen.

Dies bedeutet im Umkehrschluss, dass bei repetitiven und bekannten Tätigkeiten die exekutiven Funktionen nicht in Anspruch genommen werden, als Beispiel sei hier vom Autor, der tägliche Arbeitsweg bei einem neuen Arbeitgeber genannt.

Bei der ersten Fahrt ist eine hohe Konzentration des Fahrers erforderlich, um den Weg zu finden, Verkehrsschilder zu interpretieren und schwierige Verkehrssituationen einzuschätzen. Hier werden die exekutiven Funktionen eingesetzt um schnell Entscheidungen treffen zu können.

Nach einiger Zeit stellt sich jedoch ein Gewöhnungseffekt ein, aufgrund dessen der Fahrer nicht mehr jedes Verkehrsschild separat interpretieren muss, sondern das Verhalten auf einzelnen Streckenabschnitten gelernt hat.
Dies führt soweit, dass die Strecke irgendwann fast automatisch abgefahren werden kann, ohne dass bewusst nachgedacht werden muss. Hier kommen die Exekutivfunktionen nur noch bei Verkehrssituationen außerhalb der Regel zu tragen.

2.4.3.1 Exekutive Funktionen in Computerspielen

Übertragen auf ein Computerspiel bedeutet dies, dass es diese Faktoren berücksichtigen muss, um ein Training der exekutiven Funktionen zu gewährleisten. So muss es bei mehrmaligem Spielen zwingend randomisierte Elemente enthalten, da der Spieler sonst ein festes Verhaltensmuster lernen kann. Zudem muss das Spiel komplex genug sein, um den Spieler zu fordern, da z.B. das Drücken eines Knopfes bei zufällig leuchtender Lampe, trotz Zufall keine besonders schwierige Aufgabe ist und auch hier die Kopplung *„Lampe leuchtet = drücke Knopf“* irgendwann automatisch abgerufen wird.

[5] Umgangssprachlich auch Kurzzeitgedächtnis genannt.

Deshalb ist beispielsweise *Tetris* bei mehrmaligem Spielen, in höheren Leveln für ein Training der exekutiven Funktionen besser geeignet als *Mario*, da letzteres komplett aus dem Gedächtnis gespielt werden kann, auch wenn anfangs das Switching bei neuen Level Elementen stark eingesetzt wird [NTT+12].

2.4.4 Der Trail Making Test

Der *Trail Making Test (TMT)* ist ein beliebter neuropsychologischer Test, mit dem neben der Fähigkeit zur visuellen Suche, die Bearbeitungsgeschwindigkeit und die exekutiven Funktionen eines Patienten untersucht werden. Dadurch ist er eine gute Grundlage für ein Spiel, dass die Exekutivfunktionen testen und gegebenenfalls trainieren soll.

Dazu besteht ein Trail Making Test aus zwei Teilen. Für TMT-A muss der Patient 25 durchnummerierte Kreise, welche auf einem Blatt Papier verteilt sind, in aufsteigender Reihenfolge mit Strichen verbinden. Bei TMT-B muss der Patient grundsätzlich gleich vorgehen, nur dass sich Zahlen und Buchstaben abwechseln, z.B. 1-A-2-B-3-C. Diese Variante ist in Abbildung 2.15 zu sehen.

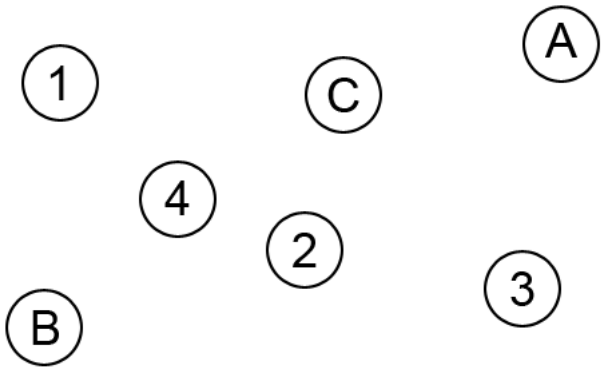

Abbildung 2.15: Ein simpler TMT-B

Währenddessen wird die Zeit gemessen, da der Patient eine bessere Wertung erhält je schneller der Test gelöst wird.

Wird ein Fehler gemacht und der falsche Kreis verbunden, muss der Patient zu dem Kreis vor dem Fehler zurückkehren und von dort aus weiter zeichnen [Tom04].

Hier wird deutlich, dass sich der TMT für die Umsetzung in einer digitale Umgebung anbietet, da die analoge Durchführung mit Personalaufwand verbunden ist. So muss die Zeit gestoppt und während der Lösung des TMT auf Fehler der Patienten geachtet werden. Beides kann leicht digitalisiert werden und hat zusätzlich eine erhöhte Genauigkeit bei der Zeitmessung zur Folge, da ein Mensch mit Stoppuhr niemals so genau sein kann wie ein Computer.

Dabei sind Prinzip und Steuerung so simpel, dass nur wenige Erklärungen bei einer Umsetzung als Spiel notwendig sind. Außerdem sind für den optischen Aufbau lediglich Kreise mit Zahlen oder Buchstaben nötig, weswegen sich die Grafik an beliebige Szenarien anpassen lässt.

Nachdem die Leap Motion längliche Gegenstände, wie Stifte, als Tool erkennt, lässt sich der TMT auch leicht auf eine Steuerung per Leap adaptieren, da weiterhin ein Stift verwendet werden kann.

Nachdem bei einer Digitalisierung des TMT somit nur Optik und Steuerung, nicht aber das eigentliche Prinzip des Tests, angepasst werden, lässt sich ein möglicher Trainingseffekt der digitalen und analogen Version bei einer Untersuchung sinnvoll vergleichen.

3

Anforderungen

Das Ziel dieser Arbeit ist es, ein motivierendes Serious Game zu entwickeln, welches sich durch sein Game-Design positiv von Spielen dieser Art abhebt. Diese sind vor allem grafisch häufig nicht auf dem neuesten Stand sind und ziehen Spieler, aufgrund spielerischer Mängel, nur selten in ihren Bann. Nun profitieren aber gerade Serious Games davon, durch mehr Spielzeit einen höheren Trainingseffekt zu erzielen.

Deshalb ist es das Anliegen von **Where are my Pirates** diese Bereiche zu betonen, indem das Spiel grafisch und inhaltlich so ansprechend wie möglich gestaltet und ein interessantes neues Steuerungskonzept entwickelt wird.
Die so entstehenden Anforderungen an das Framework wird in Abschnitt 3.1 und an das Steuerungskonzept in Abschnitt 3.2 erläutert.

Anschließend wird in Abschnitt 3.3 wird auf die Anforderungen der Teilspiele eingegangen und in Abschnitt 3.4 werden die Voraussetzungen für ein übergeordnetes Szenario erläutert.

Dabei sollen die Kernfaktoren für das Entstehen eines GameFlows durchgehend beachtet werden, was in Abschnitt 3.5 konkretisiert wird.

Das Spiel richtet sich vor allem an ältere Menschen, da sich die exekutiven Funktionen mit fortschreitendem Alter stetig verschlechtern und sie deshalb sehr stark von einem Training profitieren. Dabei sollen zwar alle Design-Entscheidungen so getroffen werden, dass ältere Menschen möglichst barrierefrei mit dem Spiel umgehen können, es zugleich aber auch jüngere Menschen anspricht.

3.1 Spiele-Framework

Es soll ausschließlich 3D-Grafik zum Einsatz kommen, um sich von einfachen Flash- und Browserspielen abzuheben. Damit dabei eine hohe grafische Qualität garantiert ist, werden weder die 3D-Modelle, noch die grafischen Effekte oder gar die vollständige Spiele-Engine vom Autor selbst entwickelt. Dies allein hätte die Entwicklungszeit dieser Arbeit vollständig in Anspruch genommen.

Stattdessen soll ein geeignetes Spiele-Framework zum Einsatz kommen, dass den Entwickler von Standardaufgaben befreit und alle für die Entwicklung notwendigen Werkzeuge bereits enthält. So soll erreicht werden, dass sich der Autor auf die eigentlichen Inhalte konzentrieren kann.

3.2 Steuerung

Als Eingabegerät soll die Leap Motion zum Einsatz kommen, da sie großes langfristiges Potenzial bietet. Nachdem die Technologie zur Erfassung von Fingerbewegungen und Gesten dieser Art völlig neuartig ist, müssen die Steuerungskonzepte von Grund auf erarbeitet und getestet werden.

Weil ein Großteil der Bevölkerung über keine Erfahrung mit kontaktloser Mensch-Computer-Interaktion durch Finger- und Handbewegungen verfügt, muss die Steuerung möglichst einfach und eingängig gehalten werden. Bevorzugt sollen sich die Nutzer das Steuerungskonzept auch ohne weitere Erklärungen, nur durch spielerisches Ausprobieren erschließen können.
Dafür soll sich die Steuerung an den intuitiven Bewegungen der Spieler orientieren und diese nach Möglichkeit nutzen. Das heißt, wenn ein Großteil der Tester eine bestimmte Bewegung zur Steuerung vermutet, wird diese bevorzugt umgesetzt.

Bei der Leap-Steuerung soll ebenfalls die Ergonomie der Hand- und Fingerbewegungen beachtet werden. Der Autor nimmt an, dass längere Verwendung der Leap bei ungünstiger Gestenwahl, schnell zur Ermüdung der Arme führt. Dies soll durch die Wahl der notwendigen Bewegungen zumindest vermindert werden, da nicht klar ist, ob sich dieser Effekt gänzlich vermeiden lässt.

Für Test- und Debugging-Zwecke soll zusätzlich eine Steuerung per Maus und Tastatur implementiert werden.

3.3 Die drei Szenen

Da **Where are my Pirates** auf ein Training der exekutiven Funktionen hin entwickelt wird, soll das Spiel zwei Teilspiele mit begrenztem Aufgabenumfang, sowie eine „Freie Welt“ beinhalten und auswertbare Daten speichern, auf was in diesem Abschnitt eingegangen wird.

Dabei sollen in allen drei Szenarien die Daten so erhoben und aufbereitet werden, dass ein Psychologe mit typischer Statistik-Software wie *R*, sinnvolle Schlüsse über den Zustand der Exekutivfunktionen des Spielers ziehen kann. Welche Daten jeweils gespeichert werden, wird in den einzelnen Implementierungskapiteln beschrieben.

Es wäre im Gegenzug schwierig, ein komplexes Spielkonzept nach Fertigstellung auf das Training der exekutiven Funktionen anzupassen. Dabei kämen zu viele Einflussfaktoren zusammen, die sich bei einer Untersuchung nicht sauber trennen lassen.

3.3.1 Das Trail Making Test Spiel

Diese Arbeit soll als erstes Teilspiel den Trail Making Test aus Kapitel 2.4.4 enthalten, der als Entwurf in Abbildung 3.1 zu sehen ist, da dieser auf Papier als Test der exekutiven Funktionen verwendet wird. Die einzelnen Kreise werden allerdings randomisiert angeordnet, damit die Spieler den Test mehrmals absolvieren und nicht auswendig lernen können. Die Verbindung zwischen den Kreisen soll mithilfe der Leap Motion über Fingerbewegungen hergestellt werden.
Zur Auswertung wird die zum Lösen benötigte Zeit gemessen und gespeichert.

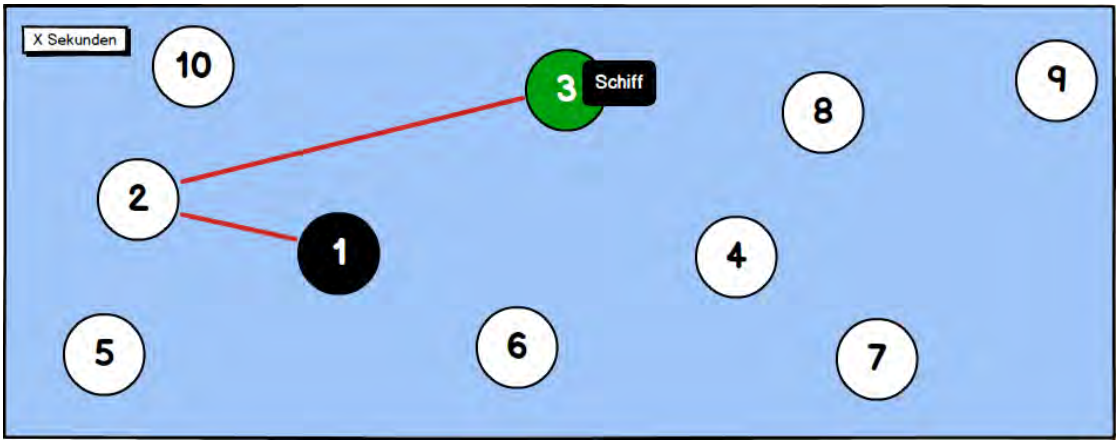

Abbildung 3.1: Ein Entwurf der TMT Szene

3.3.2 Das Fischer Spiel

Beim zweiten Teilspiel, dem Fischer Spiel, soll der Spieler Objekte einsammeln und dabei zwischen zwei Zuständen wechseln müssen. Die Objekte werden den Zuständen zugeordnet und dem Spieler bei richtiger Kombination von Objekt und Zustand positive und andernfalls negative Rückmeldung geben.

Dies soll die Switching Komponente der Exekutivfunktionen trainieren und dazu führen, dass sich der Spieler nicht simple Routinen aneignen kann. Stattdessen ist beabsichtigt, dass der Spieler über Zustandswechsel aktiv nachdenken muss.

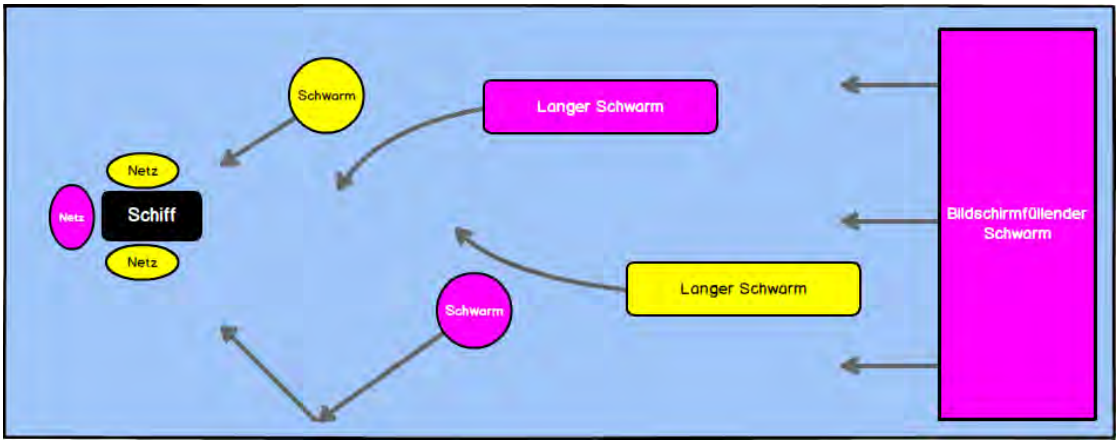

Abbildung 3.2: Ein Entwurf der Fischer Szene

Die Objekte werden wie Abbildung 3.2 zeigt, in verschiedenen Mustern und Bewegungsarten auftreten, um Monotonie zu vermeiden.
Auch hier wird die Gesamtdauer gespeichert, bis der Spieler genügend Objekte eingesammelt hat und zusätzlich wie oft der Farbwechsel eingesetzt wurde.

3.3.3 Die Freie Welt

Nachdem die beiden Teilspiele nicht für sich allein stehen, sondern in eine gemeinsame Welt eingebettet werden, muss ein Szenario gefunden werden, bei dem sich die Einzelteile sinnvoll verbinden lassen.

Um dem Spieler auch in dieser „Freien Welt“ eine auswertbare Aufgabe zu geben, soll hier die Leap Steuerung eingeführt und die Orientierung des Spielers in der Welt getestet werden. Dazu muss der Spieler mehrere randomisiert verteilte Ziele finden, wobei der bei der Suche zurückgelegte Wert gespeichert wird.

3.4 Szenario

Was noch fehlt ist ein gemeinsames Szenario für die beiden Teilspiele sowie die freie Spielwelt, das dem Spieler einen logischen Übergang zwischen ihnen ermöglicht. Die

Begrenzungen der einzelnen Spiele, in Bezug auf Aufgabe, Umfang und Levelgrenzen, sollen sich für den Spieler innerhalb der Spiellogik wiederfinden und dadurch die Immersion erhöhen.

Dabei wird die Freie Spielwelt nur einen begrenzten Umfang haben, dem Spieler aber bevorzugt natürliche Hindernisse, statt unsichtbarer Wände, als Grenze in den Weg stellen. Die Teilspiele werden durch Aufträge von Nicht-Spieler-Charakteren (NPCs von english *non-player-characters*) gestartet. Zudem soll die Spielwelt landschaftlich und grafisch abwechslungsreich sowie mit prägnanten Landmarken gestaltet sein, um dem Spieler die Orientierung zu erleichtern.

Nachdem die Teilspiele über Aufträge der NPCs gestartet werden, muss ihnen die Landschaft einen Grund bieten, sich in ihr aufzuhalten. So akzeptieren Spieler die Charaktere in einem Dorf problemlos, aber erwarten in ungewöhnlichen Umgebungen, wie einem aktiven Vulkan, glaubwürdige Begründungen für deren Aufenthalt.

Die Geschichte dazu soll sehr minimalistisch ausfallen und zum Großteil durch entsprechende Gestaltung der einzelnen Spielteile getragen werden und lediglich den Rahmen für die Minispiele bieten. Erzählt wird sie zusammen mit den Aufgabenbeschreibungen, durch ein oder zwei zusätzliche Sätze der NPCs.

Das Szenario muss so gewählt sein, dass es sich mit der Spielmechanik des Trail Making Test, dem Fischerspiel, der Leap Steuerung und einer passenden Geschichte vereinbaren lässt und dabei die Stärken von Unity nutzt.

3.5 Erreichen des GameFlow

Um während des Spielens positive Gefühle beim Spieler hervorzurufen, sollte ein Computerspiel versuchen ihn in einen Flow bzw. GameFlow zu versetzen und möglichst viele der Kernfaktoren umsetzen, wie in Kapitel 2.4.2.2 erläutert wurde.

Dabei werden im Folgenden die konkreten GameFlow Anforderungen an **Where are my Pirates** erläutert.

- **Concentration**: Um die Anforderungen an die Konzentration möglichst gering zu halten, dürfen die Teilspiele eine maximale Spielzeit von einigen Minuten nicht überschreiten. Die „Freie Welt“ soll zudem keine zeitkritische Komponente enthalten, um die Konzentration des Spielers dort nicht unnötig zu fordern.
- **Challenge** + **Skills**: Da beide Teilspiele eine Zeitkomponente haben, passt sich der Schwierigkeitsgrad ein Stück weit automatisch an den Spieler an. Bessere Spieler lösen das Spiel schneller und versuchen ihre letzte Bestzeit zu schlagen, was auch im Sinne des Trainingseffektes ein wünschenswertes Ergebnis ist. Dabei sind beide spielerisch bewusst einfach gehalten, um neuen Spielern einen sanften Einstieg zu ermöglichen und ältere Menschen nicht zu überfordern.
- **Clear Goals**: Nachdem sowohl die Teilspiele (verbinde alle Kreise/sammle Objekte des richtigen Zustands) als auch die „Freie Welt“ (finde die Objekte) bereits durch ihr Design sehr strukturiert sind, müssen die Ziele lediglich per Benutzeroberfläche und Text klar kommuniziert werden.
- **Control**: Da das Spiel keine umfangreiche Geschichte erzählt, deren Verlauf der Spieler beeinflussen kann, bezieht sich dieser Punkt vor allem auf die Steuerung. Bei einem Rollenspiel etwa, wäre auch Kontrolle des Spielers über die Handlung denkbar.

  So muss der Spieler jederzeit Rückmeldung erhalten, welche Auswirkungen seine Bewegung hat, ob die Leap überhaupt verbunden ist und ob sie gerade Hand- oder Fingerbewegungen erkennt. Auch sollte das Spiel so schnell und flüssig funktionieren, dass keine merklichen Verzögerungen (*Lag*) zwischen Eingabe und Ergebnis entstehen.
  Dies hängt eng mit dem nachfolgenden Feedback-Faktor zusammen.
- **Feedback**: Direkte Rückmeldung in einem Computerspiel erfolgt, neben oben genannten Punkten, hauptsächlich über grafische Einblendungen und die Benutzeroberfläche. Deswegen sollen alle für den Spieler relevanten Informationen dauerhaft eingeblendet und Anordnung sowie Farbe so gewählt sein, dass der Spieler die Verbindung zwischen Information und Spielelement leicht selbst herstellen kann.

Hierbei sind Informationen ausgenommen, die die Aufgabe zu stark vereinfachen würden, wie z.B. eine Landkarte in der Insel Szene.

- **Immersion**: Eine glaubwürdige Spielwelt mit passendem Szenario und einem konsistentem Grafikstil trägt sehr stark zur Immersion bei. Ein Spieler wird nicht mehr in die Welt eintauchen können, wenn er beginnt diese zu hinterfragen.

3.6 Zusammenfassung

Die genannten Anforderungen an das Spiel werden zur besseren Übersicht nochmals in folgender Tabelle zusammengefasst.

Dabei wird unterschieden zwischen funktionalen Anforderungen (FA), die beschreiben was das Spiel tun soll und nichtfunktionalen Anforderungen (NFA), die beschreiben wie und in welcher Art das Spiel die FAs erfüllen soll.

Tabelle 3.1: Zusammenfassung der Anforderungen

Name	Titel	Beschreibung
FA 1	Exekutivfunktionen	Die Exekutivfunktionen sollen trainiert werden
FA 2	Leap Motion	Alles im Spiel soll per Leap Motion gesteuert werden
FA 3	3D-Grafik	Die Grafik soll dreidimensional sein
FA 4	Auswertung	Die Teilspiele und „Freie Welt“ sollen für Psychologen auswertbare Daten erheben und speichern
FA 5	TMT Spiel	Dieses Teilspiel soll den Trail Making Test digital umsetzen
FA 6	Fischer Spiel	Dieses Teilspiel soll Switching durch Zustandswechsel forcieren
FA 7	Freie Welt	Die Insel soll frei erkundbar sein

FA 8	Szenario	Es soll ein Szenario gefunden werden, das glaubwürdig ist und die Teilspiele mit der „Freien Welt“ verbindet
FA 9	Aufträge	Teilspiele sollen durch NPC Dialoge starten
NFA 1	Landschaft	Die Landschaft soll abwechslungsreich mit prägnanten Landmarken und natürlichen Hindernissen gestaltet sein
NFA 2	Geschichte	Die Geschichte soll das Szenario unterstützen und minimal sein
NFA 3	Teilspieldauer	Teilspiele sollen nicht länger als einige Minuten dauern
NFA 4	Teilspielart	Teilspiel sollen spielerisch einfach sein
NFA 5	Teilspiele	Teilspiel sollen klare Ziele haben
NFA 6	Benutzeroberfläche	Benutzeroberfläche soll Spielziele klar kommunizieren und sich durch Farbwahl/Anordnung klar auf bestimmte Spielelemente beziehen
NFA 7	Steuerung	Die Leap-Steuerung soll flüssig funktionieren, intuitiv sein und direkte Rückmeldungen geben
NFA 8	Performance	Alle Szenen sollen ohne Lag spielbar sein

4

Implementierung

Dieses Kapitel behandelt die Implementierung von **Where are my Pirates**, erläutert wie die einzelnen Anforderungen umgesetzt werden und geht auf Probleme sowie deren Lösungen bei der Entwicklung ein.

Zuerst wird in Abschnitt 4.1 auf die Wahl des Szenarios eingegangen und in Abschnitt 4.2 der Aufbau des Startmenüs beleuchtet. In Abschnitt 4.3 wird daraufhin die Implementierung der „Freien Welt“ bzw. Insel Szene, in Abschnitt 4.4 die Implementierung des Trail Making Tests und abschließend in Abschnitt 4.5 die Implementierung der Fischer Szene erklärt.

4.1 Wahl des Szenarios

Wie bei den Anforderungen in Kapitel 3.4 beschrieben, soll das Szenario eine glaubwürdige Verbindung der Teilspiele und logische Übergänge zwischen ihnen ermöglichen. Zudem soll die „Freie Welt“ durch natürliche Hindernisse begrenzt sein und dem Spieler glaubwürdig NPCs präsentieren.

Wie in Abschnitt 4.3.1 näher erläutert ist, wird die „Freie Welt“ mit Unitys Terrain Editor erstellt und enthält Berge als Begrenzung. Da die Grundfläche eines Terrains zudem quadratisch ist bietet es sich an, die „Freie Welt“ als Insel umzusetzen.

Weil die Insel NPCs beherbergen soll, bietet es sich ebenfalls an, ein kleines Dorf oder zumindest einige Häuser zu platzieren, um einen glaubwürdigen Hintergrund für diese zu bilden. Somit ist klar, dass die Insel keine triste Wüste sein kann, da sonst wahrscheinlich keine Besiedlung vorhanden wäre.

4.1.1 Die Epoche

Where are my Pirates hätte grundsätzlich in jeder beliebigen Epoche spielen können. Da allerdings keine 3D-Modelle selbst erstellt werden sollen, hat sich der Autor bei frei verfügbaren Modellen umgeschaut und ist zu dem Schluss gekommen, dass das Angebot bei einem Piraten Szenario am größten ist.

Ein zu modernes Szenario hätte zudem politische und ökologische Fragen wie Überfischung ins Spiel bringen können, die allerdings kein Thema dieser Arbeit sind und durch ein Karibik-Piraten Szenario vermieden werden. Dies hat den weiteren Vorteil, dass der Spieler aufgrund von Urlaubserfahrungen und der Darstellung in den Medien, eher positive Gefühle mit Sonne, Meer und Palmen verbindet.

Das Szenario ist dabei dem Bereich Fantasy zuzuordnen, da auf Klischees zurückgegriffen, 3D-Modelle verschiedener Stile, je nach Verfügbarkeit gemischt werden und auf eine realistische Darstellung des piratigen Lebensgefühls verzichtet wird. Dieses Szenario bietet zudem die Möglichkeit, zusätzliche interaktive Elemente, wie Kanonenschüsse, zu verwenden.

4.1.2 Das Trail Making Test Szenario

Die Teilspiele hätten sowohl auf der Insel selbst als auch im Wasser spielen können, wobei wie im Folgenden erläutert, das Meer als Umgebung für den TMT eingesetzt wird.

Bei der Entwicklung wurde früh festgestellt, dass der TMTs einen optischen Zeiger voraussetzt, um sich per Leap steuern zu lassen, da der Spieler sonst keine Orientierungshilfe für die Position seines Fingers hat. Da das Bewegen des Zeigers die einzige Interaktion des TMT darstellt, soll dieser fester Bestandteil des Szenarios und kein abstraktes Objekt, wie eine farbige Kugel, sein. Genauso sollen die Kreise Teil des Szenarios sein.

Um all dies zusammenzuführen, ist der Zeiger ein Schiff, welches kreisförmige Fischschwärme so schnell wie möglich in der richtigen Reihenfolge erkunden muss, was fertig umgesetzt in Abbildung 4.1 zu sehen ist.

Abbildung 4.1: Das fertige Trail Making Test Szenario

Nachdem es bei einer Erkundung keinen Sinn ergibt, einen Schwarm gar nicht oder mehr als einmal abzusegeln, kann auch die Fehlerbehandlung des TMT (Zurücksetzen um einen Kreis) in das Szenario eingebunden werden.
Außerdem wird diese Aufgabe vom Spieler wahrscheinlich akzeptiert, da Fischfang

eine bedeutende Nahrungsquelle für Inselbewohner ist und eine Untersuchung der Schwärme, etwa zur Bestandsschätzungen, für diese eine nachvollziehbare Tätigkeit darstellt.

Somit sind Zeiger und Kreise fest in das Szenario eingebettet und werden nicht mehr als Fremdkörper wahrgenommen. Lediglich die Nummerierung der Kreise erfolgt noch separat, auf welche aber nicht verzichtet werden kann.

Als Nebeneffekt lassen sich die Fischschwärme durch Schwimmbewegungen animieren und der Hintergrund durch weitere Meeresbewohner, wie Wale, lebhafter gestalten, wodurch der TMT optisch nicht mehr so statisch und trist ist.

4.1.3 Das Fischer Szenario

Wie der Name bereits vermuten lässt, stellt auch in der Fischer Szene das Meer die Umgebung bereit und der Zeiger ist wieder ein Schiff, was in Abbildung 4.2 zu sehen ist.

Abbildung 4.2: Das fertige Fischer Szenario

Das Hauptproblem bei der Anpassung des Szenarios an diese Szene ist, einsammelbare Objekte auf die beiden Zustände aufzuteilen und eine Begründung zu finden, warum nur Objekte eines bestimmten Zustandes gesammelt werden können. Außerdem soll sich mit dem Zustandswechsel auch optisch der Zeiger und damit etwas am Schiff ändern,

dass sich klar mit den Objekten in Verbindung bringen lässt. Es wäre zwar möglich, z.B. nur die Farbe des Schiffs entsprechend zu ändern, allerdings wird nach einer besseren Lösung innerhalb der Spiellogik gesucht.
Weiterhin sollen die Objekte in verschiedenen Bewegungsmustern vorhanden sein.

Da im TMT Szenario Fischschwärme erkundet werden, bietet es sich an, diese im zweiten Teilspiel vom Spieler fangen zu lassen. Somit können die unterschiedlichen Bewegungsmuster glaubwürdig umgesetzt werden, da sich echte Fischschwärme ebenfalls sehr unterschiedlich bewegen.

Fische kommen zudem in den unterschiedlichsten Formen, Farben und Größen vor, weswegen die Einteilung in zwei Zustände, durch zwei Farben glaubwürdig möglich ist. Um zu gewährleisten, dass der Spieler die Fische trotz fortschreitendem Alter oder eingeschränktem Farbsehen noch wahrnehmen kann, wird Lila und Gelb mit unterschiedlicher Helligkeit, Sättigung und Farbton eingesetzt, da die Fähigkeit zur Unterscheidung dieser Eigenschaften im Alter stetig abnimmt [DFS01].

Diese Farbkombination ist beliebig gewählt und trotz vorhandener Farbblindheit noch gut zu erkennen, wie in Abbildung 4.3 zu sehen ist.

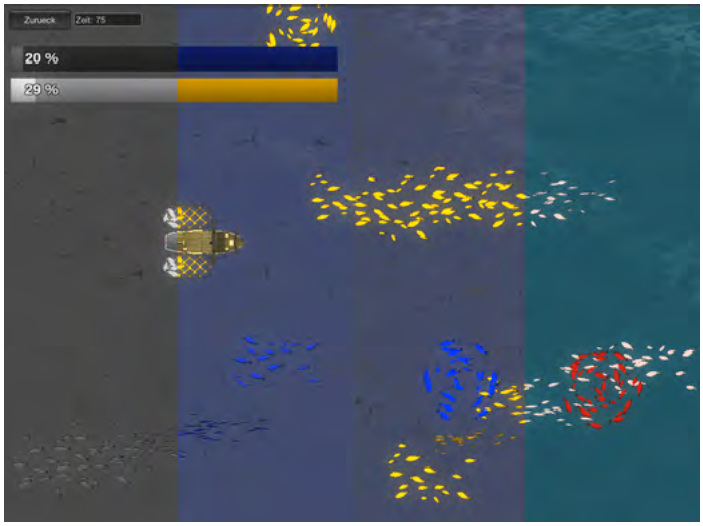

Abbildung 4.3: v. l. n. r. Die Fischer Szene bei Farben-, Grün-, Rot- und Blaublindheit

Um den Zustandswechsel des Schiffes einzubinden, gibt es zwei verschiedene Netzarten die jeweils in der Farbe der Fische kodiert sind, was in Abbildung 4.4 gezeigt wird. Sie orientieren sich optisch an modernen Schleppnetzen, was historisch zwar nicht korrekt ist, in einem Fantasy-Szenario aber nicht weiter stört.

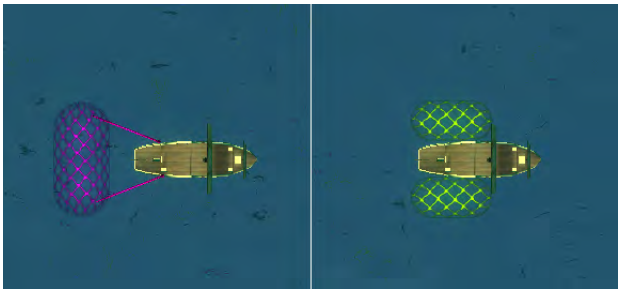

Abbildung 4.4: Die beiden Zustände des Schiffs

Das lila Netz wird hinter dem Schiff her gezogen und die gelben Netze sind seitlich angebracht, was optisch auch ohne die Farben leicht zu unterscheiden ist. Beim Zustandswechsel wird dementsprechend ein Netz eingeholt und das andere ausgeworfen,

4.2 Das Startmenü

Früh entstand der Gedanke, dass es vorteilhaft sein könnte ein eigenes Startmenü zu haben, da verschiedene Werte flexibel geändert werden müssen. Dies betrifft vor allem die Anzahl der Pflanzen auf der Insel sowie der Kreise im TMT und die Einstellung, ob das Spiel per Maus/Tastatur oder Leap gesteuert werden soll.

Eine Maus und Tastatur Steuerung zusätzlich zur Leap wird ohnehin für Entwicklungs- bzw. Debuggingzwecke benötigt und nachdem Leap und Maus nicht gleichzeitig verwendet werden können, ist diese Einstellungsmöglichkeit hilfreich. Aus diesem Grund kann das Startmenü auch nur per Maus bedient werden.

Bei einer Untersuchung mithilfe von **Where are my Pirates** kommen verschiedene Werte für die Anzahl der Pflanzen auf der Insel und der Kreise beim TMT in Betracht. Da über diese Werte sowohl die Schwierigkeit als auch die Dauer der einzelnen Aufga-

ben reguliert wird, ist von Psychologenseite die Möglichkeit zur einfachen Umstellung gewünscht.

Abbildung 4.5: Das Startmenü

Aus Gründen der Software-Ergonomie sind diese Einstellungen deshalb in ein eigenes Menü ausgelagert, welches in Abbildung 4.5 zu sehen ist. Der Einsatzzweck dieser Arbeit sieht zwar vor, dass das Menü nur von den betreuenden Psychologen und nicht dem Spieler selbst gesehen wird, weswegen es eigentlich nicht in das Gesamtszenario hätte eingebettet sein müssen. Der Autor hat sich aber trotzdem dazu entschlossen, das Menü optisch dem restlichen Spiel anzupassen, um einen stimmigen Gesamteindruck zu erhalten.

4.2.1 (N)GUI

Das eingebaute System für Benutzeroberflächen (*GUI* von englisch ***G**raphical **U**ser **I**nterface*), ist einer der Schwachpunkte von Unity, da es nur über Code und nicht per *WYSIWYG* (**W**hat **Y**ou **S**ee **I**s **W**hat **Y**ou **G**et) erstellt werden kann und der Code dafür schnell kompliziert wird.

In Listing 4.1 wird beispielsweise eine simple `GUI.Box` mit eigener Hintergrundfarbe und Text angezeigt, indem eine Textur von einem Pixel Größe in schwarz angelegt und

diese dem `GUIStyle` als Hintergrund zugewiesen wird. Der `GUIStyle` wiederum wird der Box beim Zeichnen übergeben. Die direkte Zuweisung einer Hintergrundfarbe, ohne den Umweg über die Textur zu nehmen, ist nach Wissen des Autors nicht möglich. Wer bereits komplexere GUIs gestaltet hat wird bemerken, wie umfangreich solcher Code werden kann.

Listing 4.1: GUI.Box Hintergrundfarbe ändern

```
1  Texture2D tex;
2  GUIStyle myStyle;
3  void Start () {
4      tex = new Texture2D(1,1);
5      tex.SetPixel(0, 0, Color.black);
6      tex.wrapMode = TextureWrapMode.Repeat;
7      tex.Apply();
8
9      myStyle = new GUIStyle(GUI.skin.box);
10     myStyle.normal.background = tex;
11     myStyle.fontSize = 25;
12     myStyle.wordWrap = true;
13 }
14 void OnGUI() {
15     GUI.Box( new Rect( left, top, width, height), dialogString, myStyle );
16 }
```

Nachdem somit klar ist, dass Unitys eigenes GUI System nicht mächtig genug für ein ansprechendes Startmenü ist, wurde nach Alternativen gesucht. Fündig geworden ist der Autor bei NGUI, einem kostenpflichtigen GUI System aus dem Asset Store, dass auf Einfachheit und Geschwindigkeit hin entwickelt ist und WYSIWYG beherrscht.

NGUI ist hierarchisch aufgebaut, wie in Abbildung 4.6 zu sehen ist, und hat als Elternelement immer den `UI Root` gefolgt von einer Kamera. Soll ein neues Element hinzugefügt werden, wird dies über den neuen Menüpunkt *NGUI > Create > Sprite/Label/Texture/-Widget* angelegt.

Anschließend können über den Inspector die Werte geändert werden, indem z.B. bei einem *Label* der Text und Position angepasst wird.

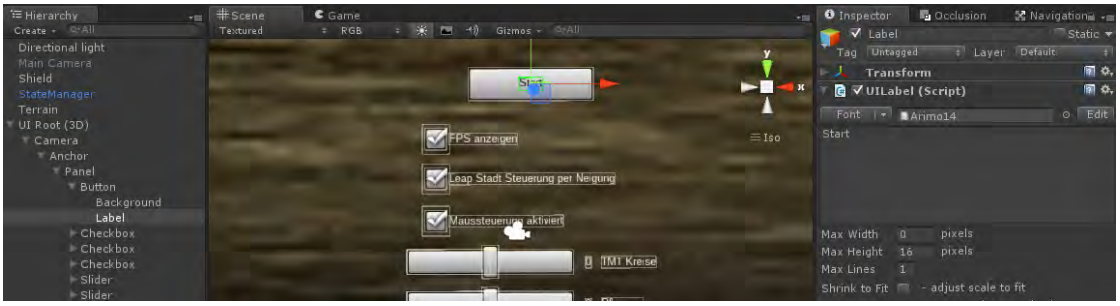

Abbildung 4.6: Das NGUI Menü im Editor

4.3 Aufbau der Insel Szene in Unity

Die „Freie Welt“ in Form der Insel Szene ist das erste, was ein Spieler vom Spiel sieht und dementsprechend ansprechend sollte ihr Eindruck sein. Zwar haben die Spieler wenig Alternativen, wenn sie im Rahmen einer Untersuchung oder Therapie spielen, allerdings vermutet der Autor, dass eine positive Erwartungshaltung eher hilft als schadet.

In diesem Kapitel wird deshalb zuerst auf die Implementierung der Insel Szene eingegangen, indem der Aufbau des Terrains in Abschnitt 4.3.1 erklärt und danach auf die Umsetzung der Leap Steuerung in Abschnitt 4.3.2 eingegangen wird. Anschließend werden weitere verwendete Skripte in Abschnitt 4.3.3 erläutert und Performance-Optimierungen in Abschnitt 4.3.4 behandelt.

Die Egoperspektive wird dabei als Kameraperspektive festgelegt, da sie, im Gegensatz zur Third-Person-Perspektive, eine höhere Immersion bietet. Nachdem das Spiel mit Fingerbewegungen gesteuert wird, fühlt es sich zudem natürlicher an, dabei durch die Augen der Figur zu blicken.
Bei einer Third-Person-Perspektive hätte der Charakter außerdem komplett animiert sein und sich den Gesten entsprechend verhalten müssen, was ohne aufwendige manuelle Anpassung der Animationen nicht möglich gewesen wäre.
Aufgrund der Egoperspektive ist der Charakter zudem geschlechtsneutral und damit für Männer als auch Frauen gleichermaßen gut spielbar.

Abbildung 4.7: Die einzusammelnden Pflanzen

Als vom Spieler einzusammelnde Objekte, kommen die in Abbildung 4.7 gezeigten Pflanzen zum Einsatz. Diese sind so gewählt, dass sie sich durch den kräftigen Farbton der Blüte und ihre Größe von der restlichen Vegetation auf der Insel abheben.

Die Anzahl der Pflanzen kann im Startmenü geändert werden. Dabei sind Werte zwischen einer und 12 Pflanzen möglich, da diese nicht komplett zufällig angeordnet, sondern auf vorher festgelegte Positionen verteilt werden. So ist gewährleistet, dass alle Pflanzen an Stellen sind, die ein Spieler auch erreichen kann und sich eine prägnante Landmarke in der Nähe befindet.

4.3.1 Terrain

Es ist am einfachsten, sich bei der eigenen Vorgehensweise nach den Stärken des Frameworks zu richten, anstatt dagegen zu arbeiten. So ist schnell klar, dass der Landschaftseditor von Unity ein sehr mächtiges Werkzeug ist, dass die Erstellung von abwechslungsreichen und komplexen Landschaften ermöglicht und deshalb eingesetzt wird. Das damit erstellte Terrain ist in Abbildung 4.8 zu sehen.

Im Inneren der Insel befindet sich das Dorf, welches von einem Schutzwall sowie Wald umgeben und durch einen festen Pfad mit der Bucht und damit dem Hafen verbunden ist. Als weitere prägnante Landmarke befindet sich am Rand der Berge, zwischen Hafen und Dorf ein Wasserfall.

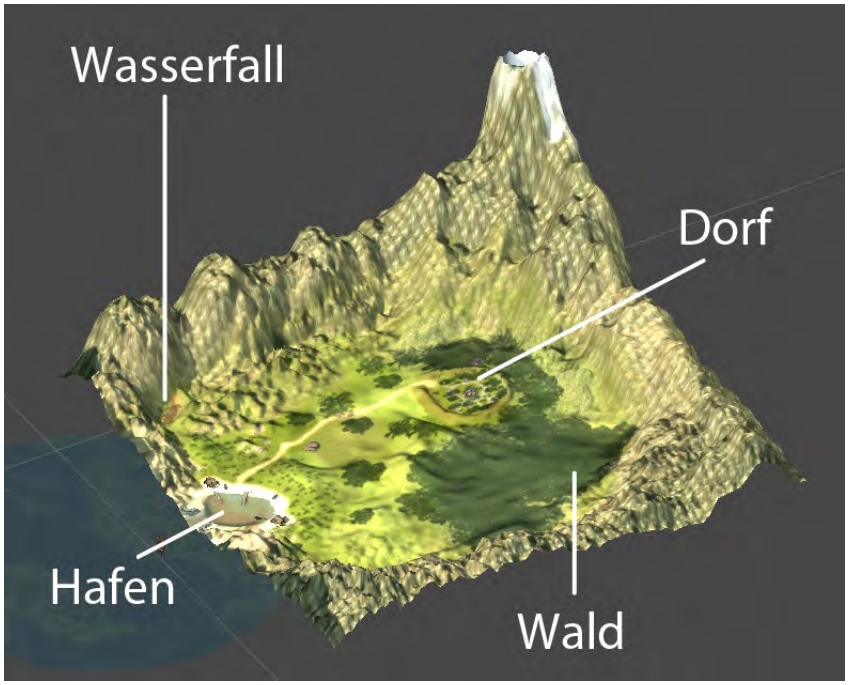

Abbildung 4.8: Das Terrain der Insel

Die Ausmaße der Landschaft sind so gewählt, dass der Spieler keine zu großen Entfernungen zurücklegen muss, aber auch nicht das komplette Terrain auf einmal überblicken kann.

In den nächsten beiden Abschnitten wird noch genauer auf den Aufbau von Hafen und Dorf eingegangen, gefolgt von einem Abschnitt über den Aufbau der Berge.

4.3.1.1 Dorf

Nachdem die Insel von NPCs bewohnt ist, gibt es wie erwähnt ein Dorf im Landesinneren und einige Fischerhütten am Meer. Die meisten Modelle des Dorfs in Abbildung 4.9, stammen aus einem mittlerweile nicht mehr verfügbaren *Medieval Village* Paket, welches umsonst in Unitys Asset Store erhältlich war.
Darin waren sowohl die Gebäude, als auch der Teich im Zentrum, sowie einige kleinere Objekte, wie Fässer und Handkarren enthalten.

Der Spieler beginnt hier das Spiel, in direkter Umgebung des NPCs, über den der TMT gestartet werden kann. Das Dorf fungiert durch die detaillierten 3D-Modelle, sowie die ausgearbeitete Umgebung als Blickfang und lädt auch ohne weiteres Feedback zum Erkunden ein.

Abbildung 4.9: Das Dorf aus der Editoransicht und Egoperspektive

Dem Spieler ist es, sofern keine weiteren Unterweisungen durch einen Betreuer vorliegen, frei überlassen, ob er zuerst die restliche Insel erkundet, um die Pflanzen zu finden, oder den NPC anspricht und damit den TMT startet.

4.3.1.2 Hafen

Der Hafen in Abbildung 4.10 umschließt eine kleine Bucht und enthält neben Fischerhütten und Stegen zwei befestigte Kanonenbatterien am Übergang zum Meer. Vor dem Hafen schwimmt zudem das Schiff, welches in den Teilspielen als Zeiger zum Einsatz kommt.
Die Bucht soll durch Gebäude und Befestigungen als wichtig für die Inselbewohner hervorgehoben werden.

Bis auf das Schiff sind alle im Hafenbereich verwendeten 3D-Modelle kostenlos im Asset Store erhältlich. Die Galeone wurde von *Alexandre Genovese* erstellt und steht unter einer *Creative Commons - Attribution-NonCommercial-ShareAlike* (kurz *CC BY-NC-SA*) Lizenz [cre].

Das bedeutet, dass das Modell nicht kommerziell eingesetzt werden darf, die Urheberschaft des Autors entsprechend benannt werden und eine Veröffentlichung, die Material unter dieser Lizenz enthält, selbst auch wieder unter dieser Lizenz stehen muss.

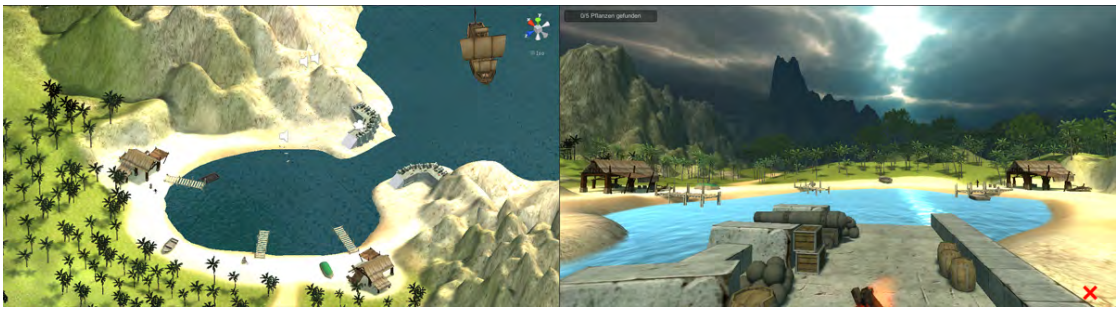

Abbildung 4.10: Der Hafen aus der Editoransicht und Egoperspektive

Sollte **Where are my Pirates** also veröffentlicht werden, müsste es unter der *CC BY-NC-SA* Lizenz stehen. Dies könnte wiederum Lizenzen von anderen Modellen widersprechen, was überprüft werden müsste.

4.3.1.3 Berge

Da bereits bei der Erstellung des Szenarios, die Umsetzung der „Freien Welt“ als Insel beschlossen wurde ist die nächste Frage, ob zusätzliche Begrenzungen der Spielwelt notwendig sind.

Grundsätzlich wäre Meer auf allen Seiten eine ausreichende Grenze, wenn der Spieler das Wasser entweder nicht betreten könnte oder es eine unsichtbare Mauer beinhalten würde. Da beide Varianten ohne zusätzliche Texteinblendungen, wie *„Du kannst nicht (weiter) schwimmen“* dem Spieler zu wenig Feedback liefern, wird nach Alternativen gesucht.

Weil ein von überall sichtbarerer und erhöhter Punkt die Orientierung auf der Insel erleichtert und sich somit das Hinzufügen von Bergen anbietet, werden diese auch als Hauptteil der Begrenzung eingesetzt. Berge werden als unüberwindbare Hindernisse vom Spieler akzeptiert und schränken zusätzlich die Sichtlinie ein, wodurch zur Belebung des Meeres

keine weiteren Inseln und Objekte am Horizont nötig sind. Außerdem stärken Berge aufgrund der in Unity einstellbaren Farbperspektive[1] und durch Bewegungsparallaxe[2] die empfundene räumliche Tiefe der Landschaft.

4.3.2 Leap Steuerung

Die Hauptaufgabe der Insel Steuerung ist die Bewegung der Spielfigur in der Egoperspektive. Zusätzlich müssen Dialoge mit den NPCs gestartet und die Pflanzen aufgesammelt werden können.
Da für keine dieser Aufgaben ein Zeiger notwendig und eine relative Steuerung deutlich fehlertoleranter ist, kommt diese zum Einsatz.

Der Spieler erhält durch die aus der Handbewegung folgenden Änderung, von Blickrichtung oder Position, bereits viel Steuerungsfeedback. Zusätzlich wird in allen Szenen, in der unteren rechten Ecke ein rotes Kreuz eingeblendet, wenn die Leap keine Finger und Hände erfassen kann.

Bei einer typischen Ego-Shooter Steuerung läuft die Spielfigur in Blickrichtung vor und zurück, kann aber zeitgleich Seitwärtsbewegungen (*Strafing*) ausführen. Es zeigte sich früh in der Entwicklung, dass zu viele Freiheitsgrade der Bewegung, die Steuerung zu kompliziert machen und Spieler überfordern, weshalb auf Seitwärtsbewegungen verzichtet wird.

Ebenfalls kann sich der Spieler nicht ducken oder springen, da dies auf der Insel nicht nötig ist und einen weiteren Freiheitsgrad beseitigt.

Somit beinhaltet das erste Steuerungskonzept die Änderung der Blickrichtung, sowie die Bewegung der Spielfigur nach vorne und hinten, was zu einem Freiheitsgrad von f=3 führt. Dabei soll *Blick oben/unten*, *Blick links/rechts* und *Bewegung vor/zurück* jeweils durch eine eigene Handbewegung umgesetzt werden.

[1] Kältere Grün- und Blautöne erscheinen dem Betrachter weiter entfernt.
[2] Unterschiedlich weit vom Betrachter entfernte Objekte bewegen sich optisch langsamer, je weiter sie weg sind, was z.B. in Side-Scroller Spielen eingesetzt wird.

4.3.2.1 Entwurf 1 - Neigungssteuerung

Die Ausgangsstellung der ersten Steuerungsversion ist eine horizontal über der Leap ausgestreckte Hand. Dabei spielt es keine Rolle ob die rechte oder linke Hand verwendet wird und in welcher Position sich die Finger befinden, da die Auswertung anhand der Handflächenposition und -rotation erfolgt, was in Abbildung 4.11 dargestellt ist.

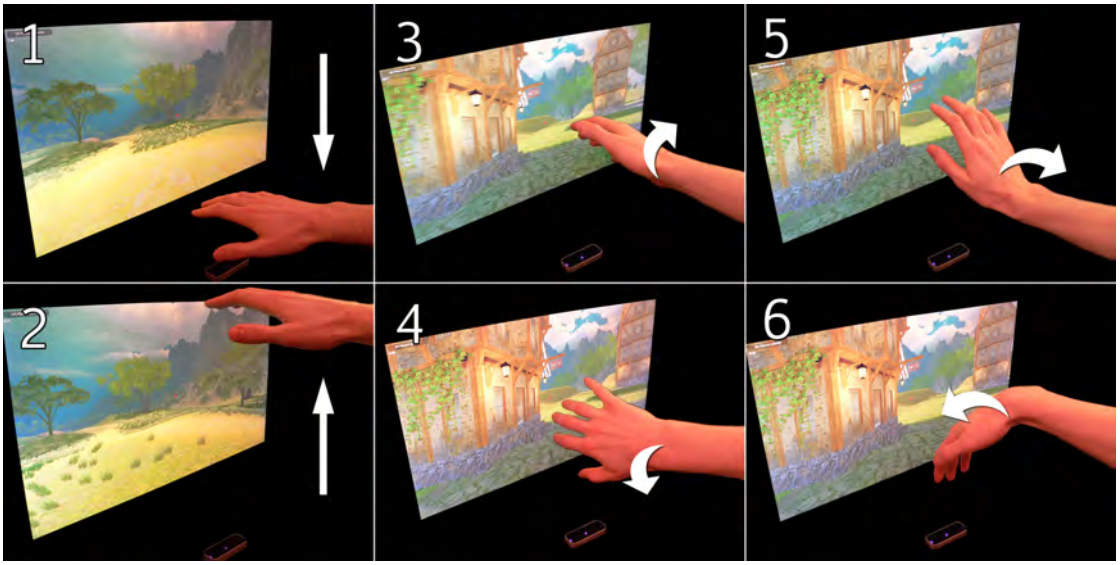

Abbildung 4.11: Die Neigungssteuerung der Insel

- **Bild 1 + 2:** Um nach vorne oder hinten zu laufen, wird die Hand näher zur Leap oder weiter von ihr weg bewegt.
- **Bild 3 + 4:** Um nach links oder rechts zu schauen, wird die Hand nach links oder rechts *geneigt*[3].
- **Bild 5 + 6:** Um nach oben oder unten zu schauen, wird die Hand nach oben oder unten *gekippt*[4].

Alle drei Bewegungen haben einen mittleren Bereich als Ruheposition, in dem keine Änderung des eigenen Freiheitsgrades stattfindet, aber die anderen beiden Freiheitsgrade

[3] Umgangssprachlich formuliert, da es sich eigentlich um eine Pronation und Supination des ganzen Unterarms handelt.
[4] Genauer ist die Palmarflexion und Dorsalextension der Hand gemeint.

geändert werden können.
Dabei läuft die Spielfigur schneller vorwärts, je näher sich die Hand über der Leap befindet und schneller rückwärts, je weiter sie von ihr entfernt ist.
Die Grenzen der Vorwärts- und Rückwärtsbewegung sind beim Autor ungefähr auf seine Monitorgröße eingestellt, d.h. die Bewegungen finden im oberen und unteren Viertel des Monitors statt.

Diese Steuerung ermöglicht es zum einen die Ansicht in zwei Richtungen gleichzeitig zu drehen und währenddessen vor/zurück zu laufen und zum anderen gezielt nur ein oder zwei Freiheitsgrade zu manipulieren.

Außerdem kann die Hand so recht entspannt gehalten werden, da die Position der Finger keine Rolle spielt, solange die Leap die Handfläche erkennt. Zudem kann der Ellbogen abgestützt werden, um den Arm zu entlasten.
Bei der Wahl der Schwellenwerte wird ein Kompromiss aus Ergonomie (möglichst kleine Bewegungen) und Barrierefreiheit (geeignet für Senioren) gewählt.

Nachdem sich der Spieler wesentlich häufiger nach vorne bewegt, wird das vor/zurück laufen nicht über eine Handbewegung nach vorne/hinten gesteuert, sondern hoch/runter. So muss die Hand nur knapp über der Leap bzw. dem Schreibtisch gehalten werden, um den Charakter vorwärts zu bewegen, was ein dauerhaftes Abstützen ermöglicht.

4.3.2.2 Probleme der Neigungssteuerung

Es hat sich in ersten Tests[5] herausgestellt, dass die Steuerung per Neigung nicht intuitiv ist und erst erlernt werden muss. Alle Tester haben, unabhängig von vorheriger Einweisung in die Steuerung, zuerst versucht die Blickrichtung durch horizontale Handbewegungen zu ändern.

Außerdem ist es für einige Tester schwierig das Neigen und Kippen der Hand getrennt voneinander vorzunehmen, weswegen oft beide Freiheitsgrade gleichzeitig geändert werden. Versehentliche Bewegungen nach links und rechts haben sich dabei als nicht

[5] Stichprobe mit Bekannten des Autors.

so hinderlich wie versehentliche Auf- und Abwärtsbewegungen herausgestellt, weil die Navigation mit Blick zum Boden oder Himmel nicht mehr möglich ist.

Nachdem die Steuerung damit bereits zu komplex ist, hat die Tester der zusätzliche Einsatz von Vor- und Rückwärtsbewegungen gänzlich überfordert. Dies hat sich nach einiger Spielzeit zwar gemindert, dennoch wird dieser Steuerungsentwurf als nicht praktikabel verworfen.

4.3.2.3 Entwurf 2 - Finale Steuerung

Um die Steuerung weiter zu vereinfachen, wird im nächsten Entwurf ein weiterer Freiheitsgrad beseitigt, indem Auf- und Abwärtsdrehungen der Ansicht nicht mehr möglich sind. Diese sind im Terrain der Insel nicht unbedingt notwendig und reduzieren die Anzahl der Freiheitsgrade damit auf f=2.

Es wurde kurzzeitig darüber nachgedacht, ob sich die Kamera in der horizontalen automatisch anhand der Steigungen des Terrains ausrichten soll, diese Idee aber als zu aufwendig verworfen.

Abbildung 4.12 zeigt die reduzierte Steuerung mit der Ruheposition.

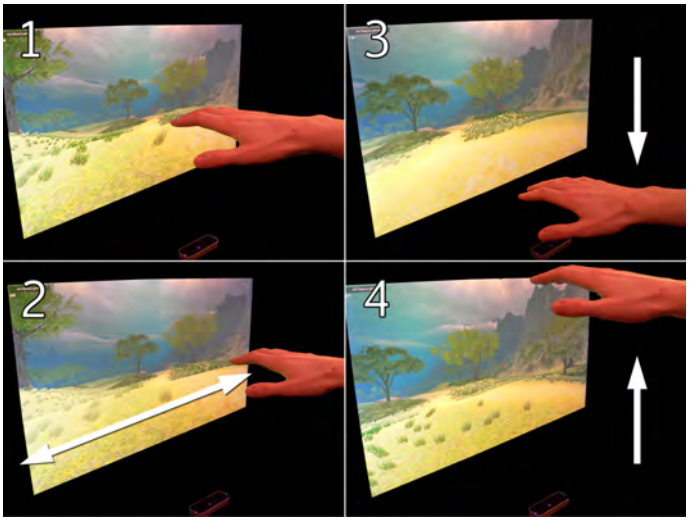

Abbildung 4.12: Die Finale Steuerung der Insel

- **Bild 1:** Die Ruheposition der Hand, ohne Änderung eines Freiheitsgrades.
- **Bild 2:** Die horizontalen Bewegungen der Kamera erfolgt, wie von den Testern erwartet, über Seitwärtsbewegungen der Hand.
- **Bild 3 + 4:** Die Steuerung der Vor- und Rückwärtsbewegung über die vertikale Distanz zur Leap, wird aus ergonomischen Gründen beibehalten.

Damit kann der Spieler nur noch die Ansicht nach links oder rechts drehen und sich gleichzeitig nach vorne oder hinten bewegen. Dieses Konzept ist für die Tester intuitiver zu verstehen und kann bereits nach kurzer Zeit eingesetzt werden.

Da der Code für beide Steuerungsvarianten noch vorhanden ist und eine Untersuchung bezüglich der Ergonomie interessant sein könnte, kann die Neigungsvariante über das Startmenü aktiviert werden.

4.3.2.4 Dialoge und Pflanzensammlung

Damit fehlt noch ein Konzept für Dialoge mit den NPCs und eine Möglichkeit die Pflanzen auf der Insel einzusammeln. Als Sammelbewegung für die Pflanzen liegt eine Greifgeste nahe, da diese intuitiv ist und über Änderungen am `SphereRadius` festgestellt werden kann. Um nicht mehr Gesten und Bewegungen als notwendig zu verwenden, wird die Greifbewegung auch zum Starten der Dialoge verwendet.

Abbildung 4.13: Der NPC Dialog im Dorf

Das eigentliche Dialogfeld ist eine mit Text gefühlte `GUI.Box()`, wie sie in Abschnitt 4.2.1, Listing 4.1 beschrieben wurde und in Abbildung 4.13 zu sehen ist.

Zuerst wurde versucht auch die Bestätigung eines Dialogs über die Greifgeste abzuwickeln, was sich aber als nicht sehr geschickt erwies. Die Tester bestätigten Dialoge oft aus Versehen, weil sie beim Lesen ihre Hand entspannten und dabei öffneten und schlossen.

Darum wird auf die *Swipe* Geste (siehe Kapitel 2.2.5) der Leap zurückgegriffen, um den Dialog zu bestätigen. Diese ähnelt der Geste eines Smartphones, mit der ein Anwender z.B. bei einer Fotogallerie von einer Seite der Anwendung zur nächsten gelangt.

4.3.2.5 Steuerungsskripte - TownLeap.cs

An dieser Stelle werden zugunsten der besseren Übersichtlichkeit nur die Skripte der finalen Steuerung aus der Klasse *TownLeap.cs* besprochen.

Listing 4.2 zeigt das Kernstück der Insel Leap Steuerung, welche sich an der *LeapFirstPersonGrab* Demo von *Pohung Chen* orientiert [Leal].

Listing 4.2: Die zentrale Methode für die Leap Steuerung der Insel

```
1  public void handlePositionControl() {
2      Hand foremostHand = GetForeMostHand();
3      Frame frame = controller.Frame();
4
5      if (foremostHand != null) {
6          statics.hideLeapDisconnectedIcon();
7          ProcessLook(foremostHand);
8          handleForwardMovement(foremostHand);
9          handleGestureRecognition(frame);
10         handleGrabbing(foremostHand);
11     }
12     else {
13         statics.showLeapDisconnectedIcon();
14     }
15 }
```

Aus Performance-Gründen wird ein Großteil der `Update()` Funktionen der Insel in einer zentralen Klasse gekapselt, welche in jedem Frame `handlePositionControl()` aufruft.

Um Fehler durch weitere Objekte im Sichtfeld der Leap zu vermeiden, wird zuerst die dem Monitor am nächsten liegende Hand ausgelesen und anhand dieser in `ProcessLook()` die Ansicht geändert, sowie in `handleForwardMovement()` die Vor- und Rückwärtsbewegungen der Figur bearbeitet.

In `handleGestureRecognition()` (Listing 4.3) wird die Wischgeste zur Bestätigung von Dialogen behandelt und in `handleGrabbing()` (Listing 4.4) die Greifbewegung zum aufsammeln der Pflanzen und starten der Dialoge.
Dabei werden die Gesten über den `Frame` ausgelesen.

Listing 4.3: Prüfen ob die Swipe Geste ausgeführt wurde

```
1  private void handleGestureRecognition(Frame frame){
2      GestureList gest = frame.Gestures(lastFrame);
3
4      if(!gest.Empty){
5          for(int i = 0; i < gest.Count; i++){
6              Gesture g = gest[i];
7
8              if(g.Type == Gesture.GestureType.TYPESWIPE){
9                  TownStatics.wasSwiped = true;
10             }
11         }
12     }
13     else {
14         TownStatics.wasSwiped = false;
15     }
16 }
```

Die Methode `ScaleFactor()` in Listing 4.4 gibt dabei die Größenänderung, der in die Hand projezierten Kugel, seit dem letzten Frame zurück. Damit wird das Öffnen und Schließen der Hand festgestellt.

Listing 4.4: Prüfen ob die Hand zugreift

```
1 private void handleGrabbing(Hand hand) {
2     if(hand.ScaleFactor(lastFrame) <= 0.85){
3         TownStatics.wasGrabbed = true;
4     }
5     else {
6         TownStatics.wasGrabbed = false;
7     }
8 }
```

Die Klasse *TownLeap.cs* selbst, ist eine Komponente des `Player` Objektes, das auch Kamera und GUI beinhaltet, weswegen direkt das `transform` Objekt gedreht werden kann. Damit dreht sich die Kamera als Kindobjekt automatisch mit, was in Listing 4.5 gezeigt wird.

In diesem Beispiel werden ebenfalls die im Leap SDK enthaltenen Methoden, wie `ToUnityScaled()`, zur Konvertierung der Leap in die Unity Skala verwendet.

Listing 4.5: Die Hand- wird in Kamerabewegung umgesetzt

```
1 private void ProcessLook(Hand hand) {
2     float handX = hand.PalmPosition.ToUnityScaled().x;
3     if( (handX < -2 || handX > 2) && !statics.showDialogs){
4         transform.RotateAround(Vector3.up, handX * 0.0025f);
5     }
6 }
```

4.3.3 Weitere Insel Skripte

Während das Leap Skript einen großen Teil der Entwicklungs- und Testzeit dieser Szene einnimmt, werden noch weitere Skripte verwendet, auf deren Besonderheiten in diesem Abschnitt eingegangen wird.

So speichert die in Abschnitt 4.3.3.1 beschriebene *StateManager.cs* Klasse szenenübergreifende Variablen. Die *TownStatics.cs* Klasse in Abschnitt 4.3.3.2 zentralisiert die

`Update()` Funktion, schreibt in Dateien und kümmert sich um die Sichtline des Spielers und die Klasse *TrackMovement.cs* in Abschnitt 4.3.3.3 schreibt die Bewegungen des Spielers über die Insel mit.

4.3.3.1 Szenenübergreifende Zustände - StateManager.cs

Unity löscht standardmäßig alle Teile der aktuellen Szene aus dem Speicher, wenn eine neue Szene geladen wird. Um Zustände, wie z.B. die Position des Spielers auf der Insel und die Anzahl eingesammelter Pflanzen, von einer Szene in die nächste zu übernehmen, gibt es zwei Möglichkeiten.

Die erste ist, dass die aktuelle Szene beim Laden der neuen nicht zerstört wird, was per Skript eingestellt werden kann. Dies hat aber einen deutlich höheren Speicherverbrauch und schlechtere Performance zur Folge und kann zu Problemen führen, wenn Objekte über den Namens ermittelt werden und in beiden Szenen gleich benannt sind.
Oder es wird wie in diesem Fall ein `StateManager` implementiert, der mit Hilfe von `DontDestroyOnLoad(StateManager)` von Szene zu Szene übernommen wird.

Da es somit in jeder Szene nur einen `StateManager` geben darf, kommt das *Singleton Pattern* zum Einsatz, was in Listing 4.6 zu sehen ist. Anschließend kann das `StateManager` Objekt und anhand dessen das `StateManager` Skript aus anderen Klassen heraus aufgerufen werden.

Zu speichernde Zustände und Werte werden als öffentliche Variablen im Skript angelegt.

Listing 4.6: StateManager als Singleton

```
1 public class StateManager : MonoBehaviour {
2     public static StateManager instance;
3
4     void Awake() {
5         if (instance){
6             // Es darf nur eine Instanz geben
7             // alle anderen werden zerstoert
8             Destroy(gameObject);
9         }
```

```
10          else {
11              instance = this;
12              DontDestroyOnLoad(gameObject);
13          }
14      }
15      // [...]
16  }
17  public class OtherClass : MonoBehaviour {
18      public static StateManager stateManager;
19
20      void otherMeStartthod(){
21          // Zuerst wird das StateManager Objekt ausgelesen
22          // und anhand dessen die StateManager.cs Skript Komponente
23          stateManager =
                GameObject.Find("StateManager").GetComponent<StateManager>();
24      }
25  }
```

4.3.3.2 Raycast - TownStatics.cs

Die *TownStatics.cs* Klasse ist eine Komponente des `Player` Objekts und kapselt fast alle `Update()` Funktionen der Insel Szene, stellt klassenweite Variablen bereit und speichert die Auswertungsdaten. Außerdem überprüft die Klasse, ob der Spieler gerade eine Pflanze oder einen NPC ansieht und hebt diese hervor.

Dazu wird in Listing 4.7 als erstes ein Strahl (*Ray*) durch die Bildschirmmitte erzeugt und per `Physics.Raycast()` geprüft, ob sich dort ein Objekt in maximal `maxDistance` Entfernung befindet und dementsprechend hervorgehoben. Dafür wird der *Shader*[6] verändert und die Farben entsprechend gesetzt.

Der verwendete *Toon Shader* ist ein Standard Asset, sollte damit also fehlerfrei funktionieren, führt aber zu Abstürzen wenn beim Build als *Architecture x86_64* eingestellt wird.

[6] Shader sind Software-Module der 3D-Grafik, die optische Effekte verursachen.

Die einzelnen Objekttypen werden anhand ihrer Tags identifiziert, so haben die Pflanzen das `Plant` Tag, die NPCs das `TownNPC` Tag usw.

Listing 4.7: Prüfung ob der Spieler ein Objekt ansieht

```
Ray ray = Camera.main.ScreenPointToRay(
    new Vector3(Screen.width*0.5f, Screen.height*0.4f, 0.0f)
);
RaycastHit hit;

if ( Physics.Raycast(ray, out hit, maxDistance) ){
    if ( hit.collider.gameObject.tag == "Plant"){
        lastHitObject = hit.collider.gameObject;
    }
    else {
        // [...]

    if(lastHitObject != null){
        lastHitObject.renderer.material.shader = toon;
        lastHitObject.renderer.material.SetColor("_Color", Color.white);
        lastHitObject.renderer.material.SetColor("_OutlineColor",
            Color.yellow);
        lastHitObject.renderer.material.SetTexture("_ToonShade", null);
    }
}
else {
    if(lastHitObject != null){
        lastHitObject.renderer.material.shader = diffuse;
        lastHitObject = null;
    }
}
```

Listing 4.8 zeigt, wie mit Unity und C# in Dateien geschrieben wird. Jedes mal wenn der Spieler eine Pflanze aufsammelt, wird der Textdatei eine Zeile mit der Pflanzenposition hinzugefügt. Damit der Zeilenumbruch mit allen Betriebssystemen korrekt funktioniert, wird auf `System.Environment.NewLine` statt `\r\n` oder ähnlichem zurückgegriffen.

Listing 4.8: In eine Datei schreiben

```
1 private void writePlantCollectedToFile(Vector3 pos){
2     string appendString = "collected:"+pos;
3     File.AppendAllText(Application.persistentDataPath+"/"+fileName+".txt",
          string.Format("{0}{1}", appendString, System.Environment.NewLine));
4 }
```

Um Probleme mit Schreibrechten zu umgehen, werden die Dateien in einem Ordner abgelegt, auf den in jedem Fall Schreibzugriff besteht, dem `persistentDataPath` Ordner. Dieser findet im Falle des Autors unter `C:\Users\BENUTZER\AppData\LocalLow\CherumGames\WhereAreMyPirates`.

In Listing 4.9 ist ein möglicher Inhalt der Auswertungsdatei zu sehen. Die Anzahl und Position der Pflanzen werden ebenso gespeichert, wie der zurückgelegte Weg des Spielers und die Position der bereits gesammelten Pflanzen.

Listing 4.9: Inhalt der Insel Auswertungsdatei

```
1 //new session//
2 plantcount:5
3 positions(x/y/z):(294.8, 158.6, 534.8)|(572.6, 156.8, 169.3)|(740.8,
      165.1, 527.8)|(524.3, 172.0, 609.9)|(416.0, 157.9, 311.5)|
4 waypoint:(397.7, 171.4, 572.7)
5 waypoint:(432.9, 178.5, 548.3)
6 waypoint:(417.7, 178.5, 548.2)
7 waypoint:(403.2, 178.6, 550.7)
8 collected:(294.8, 158.6, 534.8)
9 [...]
```

4.3.3.3 Spielerbewegungen speichern - TrackMovement.cs

Die in diesem Abschnitt behandelte *TrackMovement.cs* Klasse aus Listing 4.10, verfolgt die Bewegungen des Spielers und speichert diese mithilfe der *TownStatics.cs* Klasse in eine Datei.

Interessant ist hierbei vor allem der Einsatz der `InvokeRepeating()` Methode, mit der eine andere Methode in regelmäßigen Abständen wiederholt wird.
Theoretisch könnte `addWayPoint()` auch aus der `Update()` Methode ausgeführt werden, allerdings können keine gleichmäßigen Zeitabstände garantiert werden, da die `Update()` Aufrufe von der momentanen *Framerate* abhängt.

Um zu vermeiden, dass die Positionsangaben zu fein aufgelöst sind, werden durch Zeile 11 nur Positionen gespeichert, die weit genug von der letzten entfernt sind.

Listing 4.10: Speichert die Route des Spielers

```
1  public class TrackMovementCS : MonoBehaviour {
2      private Vector3 lastWaypoint;
3      private TownStatics statics;
4
5      void Start () {
6          statics = GameObject.Find("Player").GetComponent<TownStatics>();
7          InvokeRepeating("addWayPoint", 0, 3f);
8          lastWaypoint = transform.position;
9      }
10     void addWayPoint(){
11         if( Vector3.Distance(lastWaypoint, transform.position) > 5){
12             lastWaypoint = transform.position;
13             statics.writeWaypointToFile(transform.position);
14         }
15     }
16 }
```

Zwischenzeitlich bestand die Idee, dass der zurückgelegte Pfad auf einem Screenshot der Insel eingezeichnet wird, um ihn auch optisch auswerten zu können. Die Logik dafür wurde entwickelt, in der finalen Version allerdings wieder verworfen, da letztendlich nur maschinenlesbare Daten zur Auswertung relevant sind.

Nichtsdestotrotz soll der Aufbau kurz vorgestellt werden, da er einige interessante Lösungen enthält. Ein Screenshot, der mithilfe der Skripte angelegt wurde ist in Abbildung 4.14 zu sehen. Dabei gilt, je heller der Rotton desto aktueller die Position.

Abbildung 4.14: Ein Screenshot der Insel mit eingezeichnetem Pfad

Um die Vogelperspektive zu erhalten, wird eine zweite orthographische[7] Kamera platziert, welche die ganze Insel überblickt, sowie senkrecht zum Terrain ausgerichtet ist und zwischen beiden Kameras gewechselt. Die Linien werden mit einer *Line Renderer* Komponente erstellt, die sich auf einem leeren GameObject befindet.

Mithilfe des Codes in Listing 4.11, werden die Wegpunkte auf den *Line Renderer* übertragen und dabei in Zeile 4 die Höhe angepasst, damit sie sich immer über dem Terrain befinden und nicht davon verdeckt werden.

Listing 4.11: Die Wendepunkt des Line Renderers setzen

```
1 public void setLines(){
2     lr.SetVertexCount(waypoints.Count);
3     for(int i = 0; i<waypoints.Count; i++){
4         Vector3 v = new Vector3(waypoints[i].x, 500, waypoints[i].z);
5         lr.SetPosition(i, v);
6     }
7 }
```

[7] Das Bild wird nicht perspektivisch verzerrt.

Der eigentliche Screenshot wird in Listing 4.12 mit `CaptureScreenshot` gespeichert, nachdem die Linien gesetzt, mit der `do while()` Schleife eine freier Dateiname der Art `ScreenshotX.png` ermittelt und zwischen den beiden Kameras gewechselt wurde.

Listing 4.12: Die Kameras ändern und einen Screenshot speichern

```
1  private int screenshotCount = 0;
2  public void takeScreenshot(){
3      setLines();
4      playerCam.enabled = false;
5      skyCam.enabled = true;
6
7      string screenshotFilename;
8      do{
9          screenshotCount++;
10         screenshotFilename = Application.persistentDataPath +
               "/screenshot" + screenshotCount + ".png";
11     }
12     while (System.IO.File.Exists(screenshotFilename));
13
14     Application.CaptureScreenshot(screenshotFilename);
15 }
```

4.3.4 Performance-Probleme

Ein großes Problem der Insel Szene sind starke Performance-Schwankungen, die sich durch Einbrüche der Framerate bemerkbar machen. Teils gehen die Performance-Einbrüche bei angezielten 60 FPS (von englisch *Frames Per Second*) soweit, dass die Framerate auf unter 10 FPS sinkt, wodurch die Szene unspielbar wird.

Grundsätzlich ist es sinnvoller, langsamen Code durch Messungen zu identifizieren, anstatt voreilig nach Verdacht zu optimieren. Um solche Flaschenhälse finden und beseitigen zu können, bringt Unity Pro ein umfangreiches *Profiler* Werkzeug mit.

Damit können Kennzahlen, wie die CPU und GPU Auslastung, sowie der Speicherverbrauch auf Methodenebene ermittelt werden. Dies ist allerdings nicht nur für Unitys

interne Methodenaufrufe, sondern auch für Aufrufe in allen eigenen Skripten möglich, sofern *Deep Profiling* aktiviert wird.

Da Deep Profiling einen großen Overhead mit sich bringt, der die Szene stark verlangsamen kann, können eigene Methoden(teile) auch mit `Profiler.BeginSample()` und `Profiler.EndSample()` zur Messung markiert werden.

In Abbildung 4.15 ist der Profiler mit der bereits optimierten Inselszene zu sehen.

Abbildung 4.15: Die Performance der Inselszene im Profiler

Nachfolgend werden verschiedene Optimierungsmaßnahmen vorgestellt, die unter anderem durch den Einsatz des Profilers gefunden werden.

4.3.4.1 3D-Modelle

Im Profiler stellt sich das Rendern der Szene als größte Performance-Bremse heraus, weswegen die 3D-Modelle und Landschaftsdetails nochmal kontrolliert werden.

Als erster Schritt wird bei allen Objekten überprüft, ob sie als statisch markiert sind, was grundsätzlich auf alle Objekte der Szene, die sich nicht bewegen, rotieren oder skalieren

zutreffen sollte. Nur dann kann Unity das Rendering optimieren, indem Objekte beim Zeichnen zusammengefasst werden[8].

Als nächstes wird die Anzahl der 3D-Modelle in der Szene verringert, indem weniger aber dafür größere Bäume und Gräser verwendet und nicht unbedingt notwendige entfernt werden.

Zusätzlich wird bei den Gebäuden im Dorf der *Level of Detail (LOD)* Wert optimiert. Über diesen kann ein 3D-Modell mehrere, unterschiedlich detaillierte Meshes enthalten, die jae nach Entfernung der Kamera aktiviert werden. Um die Performance zu erhöhen, wird die Schwelle jeder Stufe gesenkt, wodurch früher auf die detailärmere Ansicht gewechselt wird.

Bei genauerer Inspektion der Gebäude im Dorf fällt zudem auf, dass der in Abbildung 4.16 gezeigte Markisenstoff, mit Unitys *Interactive Cloth* und *Cloth Renderer* System erzeugt wird, welches dafür gedacht ist interaktive Stoffe zu simulieren. Damit kann z.B. ein Tuch welches ein Objekt umfließt simuliert werden. Nachdem dies für ein reines Dekorationselement, nicht nötig ist, wird dieser Ansatz durch einen simplen *Mesh Renderer* ersetzt.

Abbildung 4.16: Die Markise als Mesh statt Cloth Renderer

[8] Dieser Vorgang wird *batching* genannt und ist nur in Unity Pro möglich [unid].

4.3.4.2 Sonstige Optimierungen - DirectX 11, Occlusion Culling

Bei der Suche nach weiteren Optimierungsmöglichkeiten, stechen die in mehreren Klassen verteilten `Update()` Methoden heraus, weshalb diese zentral in der *TownStatics.cs* gekapselt werden.

Durch die diese Optimierungen steigt die Framerate auf dem System des Autors von unter 20 auf über 30 FPS, was allerdings weiterhin zu wenig ist und weit unter dem Ziel von 60 FPS liegt.

Eine starke Verbesserung bringt der Einsatz von *DirectX 11*. Dazu muss lediglich unter *File > Build Settings > Platform*, das Feld *PC, Mac & Linux Standalone* ausgewählt und der Haken bei *Player Settings... > Use Direct3D 11* gesetzt werden.

Einen weiteren großen Performance-Schub bringt der Umstieg von Unitys kostenloser auf die Pro Version. Zwar hat die Pro Version keine grundsätzlich besseren Algorithmen, kann allerdings performance-kritische Aufgaben wie das *Occlusion Culling*[9] in Form der *Occlusion Map* bereits vorberechnen. Dazu unterteilt Unity den sichtbaren Bereich in gleichmäßige Quader und berechnet vorab die Sichtlinien aus jedem Quader auf den nächsten.

Die Erzeugung der *Occlusion Map* kann sehr lange dauern, beschleunigt die entsprechenden Berechnung während der laufenden Szene allerdings sehr stark. Automatisch funktioniert dies auch nur für statische Objekte, sich bewegende Objekte müssen manuell optimiert werden.

Ebenfalls eingesetzt werden Skripte aus Unitys *Island Demo*, welche automatisch die Grafikeinstellungen verringern, wenn die FPS unter einen bestimmten Wert fallen. So wird bei Bedarf unter anderem die maximale Sichtweite, auf die Terrain Details sichtbar sind verringert und Wassereffekte ausgeschaltet.

Dadurch steigt die Framerate dauerhaft auf akzeptable 50-60 FPS bei mittleren Qualitätseinstellungen.

[9] Per *Occlusion Culling* wird nur der Teil der Szene gerendert, der aktuell gerade sichtbar ist.

4.4 Aufbau der Trail Making Test Szene in Unity

In diesem Kapitel wird der Aufbau der Trail Making Test Szene beschrieben, wobei zuerst auf die grafische Umsetzung und anschließend auf Steuerung und Skripte eingegangen wird. Dem Spieler ist frei überlassen ob er zuerst die Insel erkundet, aber da er bereits in der Nähe des Dorf NPCs beginnt, ist es wahrscheinlich, dass er früh den TMT startet.

Um zur TMT Szene oder Fischer Szene zu wechseln, muss der Spieler den entsprechenden NPC ansehen und ihm nahe genug sein, damit dieser hervorgehoben wird. Wird dann die Greifgeste ausgeführt, öffnet sich ein Dialog mit einer rudimentären Erklärung der Aufgabe. Dieser wird wie erwähnt mit der Swipe-Geste bestätigt. Bei Maus- und Tastatursteuerung wird er mit **E** aufgerufen und mit einem Klick auf den *Weiter* Button oder nochmaligem Druck von **E** bestätigt.

Nach einem Countdown von fünf Sekunden, die der Spieler nutzen kann um sich zu Orientieren, startet das Spiel und die Zeiterfassung. Wenn TMT-A gelöst wurde, wird direkt im Anschluss TMT-B geladen. Sind beide Varianten gelöst, kehrt der Spieler automatisch in das Dorf zurück.

4.4.1 Grafische Umsetzung

Den Hintergrund bildet eine große Wasserfläche, in der die Schwärme und ein Wal kreisen. Die Fischschwärme stellen dabei die Kreise des Trail Making Tests dar und sind mit den entsprechenden Nummern oder Buchstaben versehen. Der Wal hat keine spielerische Funktion und soll nur die Szene beleben.

Die Schwärme sind mit gelben Fischen unterschiedlicher Größe gefüllt, welche in kreisförmigen Bahnen um das Zentrum animiert sind, was in Abschnitt 4.4.3.3 näher beschrieben und in Abbildung 4.17 gezeigt wird.
Durch die konstante Bewegung im Vordergrund und den starken Kontrast zum Hintergrund, sind die Grenzen der Kreise auch ohne zusätzliche Begrenzungslinien gut zu erkennen.

Abbildung 4.17: Die TMT Szene beim ersten Start

Eigentlich müssten sich die Fische unter der Wasseroberfläche befinden, was sich durch optische Verzerrung und einen starken Blaustich bemerkbar machen würde. Zugunsten eines hohen Kontrastes wird darauf allerdings verzichtet.

Links oben befindet sich die Zeitanzeige, welche nach dem Countdown *5... 4... 3... 2... 1.. Los* zu zählen beginnt. Aufgrund der Leserichtung unserer Sprache, von links nach rechts und oben nach unten, schaut ein Spieler auf der Suche nach Information häufig zuerst in diese Ecke.

Auffällig ist der schwarze Rand oben und unten, welcher die Szene umgibt. Da die Kreise nicht komplett zufällig auf der Fläche positioniert, sondern auf festgelegten Positionen verteilt werden, muss die Sichtbarkeit der Kreise bei unterschiedlichen Bildschirmauflösungen sichergestellt sein.
Durch den bei Bedarf eingeblendeten Rahmen, wird ein festes Seitenverhältnis garantiert, weswegen kein Kreis außerhalb des Sichtfelds liegen kann. Der Code hierfür kann in Adrian Lopez *Theory and Principles of Game Design* Blog nachgelesen werden [unib].

Eine Alternative wäre gewesen, abhängig von der Auflösung nur sichtbare Kreise zu aktivieren, allerdings wird deren Anzahl im Startmenü festgelegt. Sind zu viele Kreise

nicht mehr sichtbar, kann nicht mehr garantiert werden, dass die eingestellte Anzahl auch möglich ist.
Sonst wäre eine Tabelle nötig gewesen, die für jede Auflösung die maximal mögliche Kreisanzahl speichert oder die Szene hätte im Hintergrund geladen und ausgewertet werden müssen.

Keine dieser Varianten wurde als so praktikabel wie die Lösung mit schwarzen Balken eingestuft, welche in Abbildung 4.18 in drei verschiedenen Auflösungen zu sehen ist.

Abbildung 4.18: Verschiedene Auflösungen der TMT Szene

Das Wasser ist ein animiertes Unity Standard Asset namens *Daylight Simple Water*, dass per Drag & Drop auf ein beliebiges GameObject gezogen wird. Üblicherweise wird dafür ein *Plane* Objekt, also eine Ebene verwendet. Das Wasser wird aber nur dann korrekt angezeigt, wenn die Ebene komplett horizontal liegt.

Nun spielt es für den User keine Rolle wie die Szene intern ausgerichtet ist, da dieser die Szene nur aus Sicht der Kamera sieht, weshalb die Ausrichtung beliebig angepasst werden kann. Es ist allerdings sehr aufwendig, die Objekte einer Szene neu auszurichten und alle entsprechenden Werte in den Skripten anzupassen.

Soll wie in der vorliegenden Szene das Wasser in einer anderen Ebene angezeigt werden, kann z.B. ein Quader mit Dicke/Tiefe als verwendet werden, wodurch das Wasser auf den Seiten ebenfalls korrekt gerendert wird.

4.4.2 Leap Steuerung - TMT_Leap.cs

Beim analogen Trail Making Test werden die Kreise mit von Hand aufgemalten Linien verbunden. Die Stiftspitze des analogen TMT, wird in der digitalen Version durch den Zeiger in Form eines Schiffs ersetzt, da der Spieler konstante Rückmeldungen benötigt, wo sich seine Fingerspitze gerade befindet.

Abbildung 4.19 zeigt eine Handhaltungsvariante mit der der TMT gesteuert werden kann.

Abbildung 4.19: Steuerung des TMT per Leap

Der Code für die Steuerung findet sich in der *TMT_Leap.cs* Klasse, welche eine Komponente des `LEAP Pointer` Objekts und in Listing 4.13 zu sehen ist. Die genaue Funktionsweise wird im nächsten Abschnitt näher erläutert.
Die eigentliche Überprüfung, ob sich der Zeiger über/in den Kreisen befindet, geschieht per Kollisionsabfrage der Objekte und wird in Abschnitt 4.4.3.2 behandelt.

Listing 4.13: Leap Steuerung des TMT

```
1 Leap.Pointable pointable;
2 if(fingerId != -1){
3     pointable = frame.Pointable(fingerId);
4 }
5 else {
6     pointable = GetForeMostPointable(frame);
7     fingerId = pointable.Id;
8 }
```

```
9  Leap.Screen screen =
       controller.CalibratedScreens.ClosestScreenHit(pointable);
10
11 Leap.Vector intersection = screen.Intersect(pointable, true);
12 float x = intersection.x * screenw;
13 float y = intersection.y * screenh;
14
15 if(!float.IsNaN(x) && !float.IsNaN(y)){
16     Vector3 pos = cam.ScreenToWorldPoint(new Vector3(x,y,zPos));
17     float x_clamped = Mathf.Clamp(pos.x, leftBorder, rightBorder);
18     float y_clamped = Mathf.Clamp(pos.y, topBorder, bottomBorder);
19
20     transform.position = Vector3.Lerp(lastPosition, new Vector3(x_clamped,
           y_clamped, zPos), 0.5f);
21     lastPosition = transform.position;
22 }
```

In dieser Klasse wird immer `Pointable` ausgewertet, was neben Fingern auch Tools wie Stifte umfasst. Damit ist die Umstellung von analog zu digital einfacher, da der Spieler weiterhin einen Stift verwenden kann, sofern er dies möchte und die Steuerung dadurch sogar präziser ist.

Die im vorherigen Beispiel zur Berechnung verwendeten Ränder des Sichtfeldes und die Bildschirmgröße, können wie in Listing 4.14 gezeigt ausgelesen werden.

Listing 4.14: Ränder und Auflösung auslesen

```
1 float dist = (transform.position - Camera.main.transform.position).y;
2
3 leftBorder = Camera.main.ViewportToWorldPoint(new Vector3(0,dist,0)).z;
4 rightBorder = Camera.main.ViewportToWorldPoint(new Vector3(1,dist,0)).z;
5 topBorder = Camera.main.ViewportToWorldPoint(new Vector3(0,1,1)).x;
6 bottomBorder = Camera.main.ViewportToWorldPoint(new Vector3(1,0,0)).x;
7
8 screenw = UnityEngine.Screen.width;
9 screenh = UnityEngine.Screen.height;
```

Ebenfalls kommt in Listing 4.13 in Zeile 6 eine angepasste Methode der *LeapFirstPersonGrab* Demo von *Pohung Chen* zum Einsatz, womit das `Pointable` ermittelt wird, welches sich am nächsten an der Leap befindet [Leal].
Dazu wird die Position der (Finger)Spitze ausgewertet, was in dem folgendem Beispiel zu sehen ist.

Listing 4.15: Der vorderste Finger wird ermittelt

```
1  private Pointable GetForeMostPointable(Frame f) {
2      Pointable foremostPointable = null;
3
4      float zMax = -float.MaxValue;
5      for(int i = 0; i < f.Pointables.Count; ++i) {
6          float palmZ = f.Pointables[i].TipPosition.ToUnityScaled().z;
7          if (palmZ > zMax) {
8              zMax = palmZ;
9              foremostPointable = f.Pointables[i];
10         }
11     }
12
13     return foremostPointable;
14 }
```

4.4.2.1 Umrechnung der Koordinatensysteme

Um die Szene per Leap zu steuern wird in Listing 4.13, Zeile 9 der Bildschirm ausgelesen, welcher sich am nächsten am `Pointable` befindet. Da das Leap SDK mehrere Bildschirme unterstützt, kann so derjenige ermittelt werden, auf den sich die Bewegung am wahrscheinlichsten bezieht.

Die Zeigerposition auf dem Bildschirm entspricht dabei der Fingerposition vor dem Bildschirm. Damit dies möglich wird, ist eine Umrechnung von Fingerkoordinaten der Leap, zu Bildschirmkoordinaten, zu Unity Kamera Koordinaten notwendig.

Dazu wird in Zeile 11 zuerst der Schnittpunkt (engl. *Intersection*) des `Pointable` mit dem Bildschirm in normalisierten Koordinaten ausgelesen. Das heißt, die Koordinaten

liegen im Bereich 0 bis 1, wobei 0/0 untypisch in der linken unteren Ecke und 1 jeweils ganz oben oder ganz rechts liegt. Um einen Punkt auf dem Bildschirm in Pixeln zu erhalten, müssen diese Koordinaten mit der Bildschirmhöhe und -breite in Pixeln multipliziert werden, was in Zeile 12 und 13 passiert.
Zeigt das `Pointable` Objekt nicht in Richtung des Bildschirms, sondern parallel dazu, gibt es keinen Schnittpunkt und `.Intersect()` gibt *NaN (not-a-number)* zurück, weswegen dies überprüft werden muss.

Mithilfe der `ScreenToWorldPoint()` Methode wird der Punkt in Pixeln in das Koordinatensystem der Szenen Kamera in Unity umgewandelt.

Mit `Mathf.Clamp()` in Zeile 17 und 18 wird sichergestellt, dass der Zeiger den sichtbaren Bereich des Bildschirms auch bei fehlerhaften Daten nicht verlassen kann.

Letztendlich wird noch mit `Vector3.Lerp()` eine Lineare Interpolation der aktuellen und der Position aus dem letzten Frame ausgeführt, um die Zeigerbewegung gleichmäßiger zu gestalten.

4.4.2.2 Spielererfahrungen der Leap Steuerung

Erste Tests haben ergeben, dass diese Art der Steuerung leicht verständlich ist, denn es ist allen Spielern nach wenigen Sekunden klar, dass sie den Zeiger mit ihren Bewegungen steuern. Was einigen Testern anfangs Schwierigkeiten bereitet hat, war dass sie mit nur mit einem Finger anstatt der ganzen Hand steuern können.

Allerdings hat sich wieder gezeigt, dass ein flüssiges Spielerlebnis für diese Art der Steuerung Grundvoraussetzung ist. Stimmt die Bewegung des Fingers durch Einbrüche der FPS nicht mehr mit der Bewegung des Zeigers überein, sind die Spieler aufgrund des fehlenden Feedbacks nicht mehr sicher, ob sie noch den Zeiger bewegen oder aufgrund von (Programm)Fehlern die Kontrolle verloren haben.

Außerdem muss die Bewegungen noch manuell ausgeglichen werden, wie es in dieser Arbeit mit `Vector3.Lerp()` geschieht, da die Zeigerbewegungen sonst abgehackt wirken, was sich negativ auf die empfundene Präzision auswirkt.

4.4.3 Skripte

In diesem Abschnitt werden Skripte behandelt, welche die Schwärme erzeugen, sie kreisen lassen, sich um die Kollisionsabfrage mit dem Zeiger kümmern und die Daten zur Auswertung speichern.

Die Objekte der Schwärme sind hierarchisch wie folgt organisiert:

```
Circles // mit TMT_CircleParent.cs Skript
  Swarm1 // mit TMT_Circle.cs Skript
    FishFlock (Clone)
      Fish // mit TMT_FishCircle.cs Skript
        Fish
      ...
    FishFlock (Clone)
    FishFlock (Clone)
  Swarm2
  ...
  Swarm4
```

Dabei haben die einzelnen Objekte die folgenden Skripte:

- Das Elternobjekt `Circles` hat das *TMT_CircleParent.cs* Skript aus dem nächsten Abschnitt, dem vier `Swarm` Gruppen untergeordnet sind.
- Jede `Swarm` Gruppe hat das *TMT_Circle.cs* Skript, das in Abschnitt 4.4.3.2 erläutert wird und darunter die von *TMT_CircleParent.cs* erzeugten `FishFlock` Einzelschwärme.
- Die Einzelschwärme enthalten einzelne Fische mit dem *TMT_FishCircle.cs* Skript aus Abschnitt 4.4.3.3, denen weitere Fische untergeordnet sein können.

TMT_CircleParent.cs erzeugt und aktiviert dabei die einzelnen Schwärme, *TMT_Circle.cs* behandelt die Kollisionsabfrage und *TMT_FishCircle.cs* animiert die Fische.

4.4.3.1 Aktivierung der Schwärme - TMT_CircleParent.cs

Um die Generierung der gewünschten Anzahl Fischschwärme mit dem TMT-A oder TMT-B Text, kümmert sich das *TMT_CircleParent.cs* Skript aus Listing 4.16.

Dabei werden solange einzelne Kreise durch die Methode `GetComponent<TMT\_Circle>().init()` aktiviert und ihnen mit der `GetComponent<TMT_Circle>().name` Methode der entsprechende String zugewiesen, bis die gewünschte Anzahl erreicht ist. Um die Mischung aus Ziffern und Buchstaben des TMT-B möglichst simpel umzusetzen, ist ein festes String Array namens `circleNames` mit der Abfolge [1, A, 2, B, ...] angelegt.

Listing 4.16: TMT_CircleParent.cs - Aktivierung einzelner Kreise

```
1  void Start () {
2      int circleCount = TMT_Main.Get().maxNumber;
3      bool numbersMode = TMT_Main.Get().numbersMode;
4
5      while(counter <= circleCount){
6          foreach (Transform childgroup in transform){
7              foreach (Transform child in childgroup){
8                  if(counter <= circleCount){
9                      TMT_Circle c = child.GetComponent<TMT_Circle>();
10                     if(randomBoolean() && !c.isActive){
11                         c.isActive = true;
12                         c.number = counter;
13                         if(numbersMode){
14                             c.name = ""+counter;
15                         }
16                         else {
17                             c.name = circleNames[counter-1];
18                         }
19                         c.init();
20                         counter++;
21                     }
22                 }
23                 else  { return; }
24             } // [...]
```

4.4.3.2 Kollisionsabfrage - TMT_Circle.cs

Die einfachste Art zu überprüfen, ob sich der Zeiger in einem Kreis befindet, ist es Unitys Kollisionssystem zu nutzen. Dafür enthält das `LEAP Pointer` Objekt einen *Sphere Collider* sowie eine *RigidBody* Komponente und ist darauf als `isKinematic` markiert.

Die Kreise enthalten jeweils einen *Capsule Collider* und das *TMT_Circle.cs* Skript, in dem mittels `OnTriggerEnter()` und `OnTriggerExit()` die Kollisionen abgefragt werden. In Listing 4.17 ist Code der `OnTriggerEnter()` Methode zu sehen.

Listing 4.17: TMT_Circle.cs - Kollisionshandler der Kreise

```
1  // Der aktuelle Kreis entspricht dem naechsten der Reihe
2  if(tmt_main.lastHovered == (number-1)){
3      Transform lastHoveredObject = tmt_main.lastHoveredObject;
4      // Nicht mehr relevante Kreise ausgrauen
5      if(lastHoveredObject != null){
6          lastHoveredObject.GetComponent<TMT_Circle>().isBlackedOut = true;
7          Transform[] children =
               lastHoveredObject.transform.GetComponentsInChildren<Transform>();
8          foreach(Transform t in children){
9              if(t.renderer != null && t.tag != "TMT_Circle"){
10                 t.renderer.material.color = Color.black;
11             }
12         }
13     }
14     // Kreisposition speichern
15     tmt_main.vertexList.Add(new Vector3(transform.position.x,
           transform.position.y, 890));
16     tmt_main.lastHovered = number;
17     // [...]
18
19     if(number == tmt_main.maxNumber){
20         tmt_main.wasSolved = true;
21     }
22 }
23 else if {
24     // [...]
25
```

```
26 // Die Knoten des LineRenderers setzen
27 lr.SetVertexCount(tmt_main.vertexList.size);
28 for(int i=0; i<tmt_main.vertexList.size; i++){
29     lr.SetPosition(i, tmt_main.vertexList[i]);
30 }
31 if(!isBlackedOut){  // Den Kreis hervor heben
32     oldColor = renderer.material.color;
33     renderer.material.color = Color.green;
34 }
```

Der Code überprüft ob der aktuell überfahrene Kreis dem nächsten in der Reihe entspricht und setzt entsprechend die Farben der Fische. Ist ein Kreis dauerhaft verbunden, wird die Farbe der Fische auf schwarz gesetzt.

Die Methode in Listing 4.18 wird von *TMT_CircleParent.cs* aufgerufen und instantiiert drei Einzelschwärme, die in Blickrichtung der Kamera verschoben sind. Somit liegen diese in mehreren Ebenen übereinander und haben dadurch mehr räumliche Tiefe. Mit dem `transform.parent = gameObject.transform` Aufruf wird das erzeugende Objekt als Elternobjekt zugewiesen.

Das erzeugte `fishflock` Objekt ist ein vorbereiteter *Prefab*, der mehrere kreisförmig angeordnete Fische mit *TMT_FishCircle.cs* Skripten enthält die den Schwarm bilden.

Listing 4.18: TMT_Circle.cs - Initialisierung der Schwärme

```
1  public void init(){
2      if(isActive){
3          Vector3 vec = new Vector3(
4              transform.position.x-1.15f,
5              transform.position.y+1.3f,
6              transform.position.z
7          );
8
9          vec.z = transform.position.z+1.1f;
10         GameObject ff = (GameObject) Instantiate(fishflock, vec,
               Quaternion.identity);
11         ff.transform.parent = gameObject.transform;
12
```

```
13          vec.z = transform.position.z+1.3f;
14          GameObject ff2 = (GameObject) Instantiate(fishflock, vec,
                Quaternion.identity);
15          ff2.transform.parent = gameObject.transform;
16
17          vec.z = transform.position.z+1.5f;
18          GameObject ff3 = (GameObject) Instantiate(fishflock, vec,
                Quaternion.identity);
19          ff3.transform.parent = gameObject.transform;
20      }
21      // [...]
22  }
```

4.4.3.3 Kreisbewegung der Fische - TMT_FishCircle.cs

Die eigentliche Kreisbewegung der Fische wird in *TMT_FishCircle.cs* mit Hilfe des Codes in Listing 4.19 animiert.

Listing 4.19: TMT_FishCircle.cs - Kreisbewegung der Fische

```
1  public class TMT_Fishcircle : MonoBehaviour {
2      public GameObject rotateTarget;
3      public float speed;
4      public bool isZaxis = true;
5
6      void Start(){
7          speed = Random.Range(20.0f, 80.0f);
8          var parent = transform.parent;
9
10         while(parent != null) {
11             if(parent.tag == "TMT_Circle") {
12                 rotateTarget = parent.gameObject as GameObject;
13                 return;
14             }
15             parent = parent.transform.parent;
16         }
17     }
```

```
18      void Update () {
19          if(isZaxis){
20              transform.RotateAround (rotateTarget.transform.position,
                    Vector3.forward, speed * Time.deltaTime);
21          }
22          else {
23              transform.RotateAround (rotateTarget.transform.position,
                    Vector3.up, -speed * Time.deltaTime);
24          }
25      }
26  }
```

Jeder Fisch mit diesem Skript bewegt sich mit randomisierter Geschwindigkeit, kreisförmig um einen Mittelpunkt `rotateTarget`. Unity bringt hierfür die Methode `Transform.RotateAround()` mit. Durch den Zufallsfaktor bewegt sich jeder Schwarm leicht anders, was das Gesamtbild lebendiger wirken lässt.

Als Ziel der Rotation wird ab Zeile 9 das Kreis Objekt gesetzt, was lediglich einem leeren GameObject entspricht, wodurch die Fische sich um dessen Mittelpunkt drehen.

Wenn sich jeder Fisch eines Schwarms selbstständig bewegen würde, bräuchte er ein eigenes Skript und hätte dadurch eine eigene `Update()` Methode, was für die Performance ein Problem wäre.
Außerdem hätte es dann passieren können, dass an manchen Stellen des Kreises, kurzzeitig keine Fische sind. Dadurch wären die Ränder und Ausmaße der Kreise nicht immer klar zu erkennen, wodurch der Spieler kein deutliches Feedback mehr erhalten würde.

Beide Probleme werden durch geschickte Schachtelung der Fische umgegangen. Sind mehrere Fische unter einem Elternobjekt samt Skript gruppiert, bewegen sich diese zwar für den Betrachter, behalten zueinander aber immer den gleichen Abstand. Durch passende Verteilung wird die Anzahl der benötigten `Update()` Aufrufe verringert und eine gleichmäßige Füllung des Kreises garantiert.

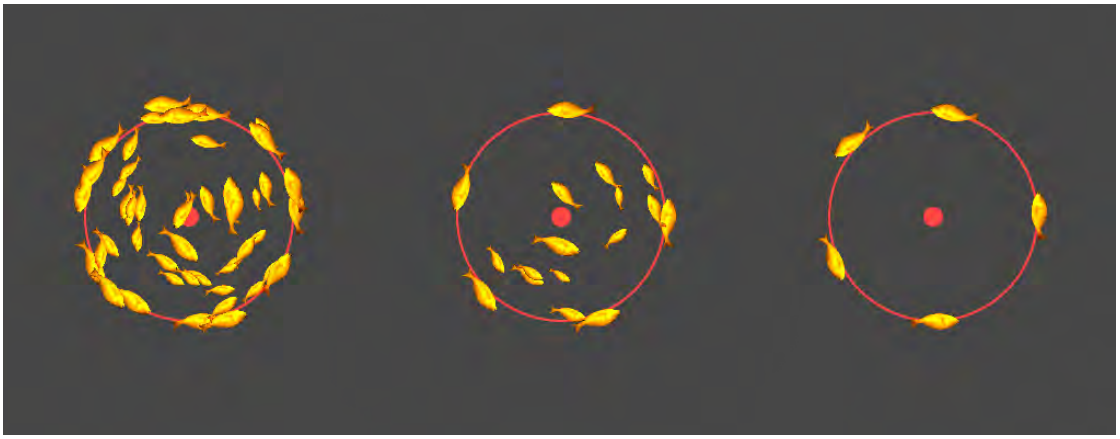

Abbildung 4.20: v.l.n.r. kompletter Schwarm, Einzelschwarm, Fischgruppe. Rot eingezeichnet der Mittelpunkt und Umfang des Kreises.

Wie in Abbildung 4.20 zu sehen ist, enthält jeder Schwarm z.B. eine fünfergruppe Fische, die sich nur auf dem Rand bewegt, um die Ausmaße des Kreises klar abzugrenzen.

Es wäre vermutlich auch hier performanter die `Update()` Methoden zentral zu kapseln. Allerdings ist mit obigen Maßnahmen die Performance kein Problem in dieser Szene, weshalb auf Überoptimierung verzichtet wird.

4.4.3.4 Auswertung

Da bereits in Abschnitt 4.3.3.2 darauf eingegangen wurde, wie mit Unity in Dateien geschrieben wird, soll an dieser Stelle nur kurz der Datenaufbau erläutert werden.
Die Auswertungsdaten werden in *tmt_stats.txt* gespeichert, ein Beispiel des Inhaltes ist in Listing 4.20 zu sehen.

Die erste Spalte beinhaltet den Zeitstempel der Erzeugung des Eintrags, gefolgt von der zur Lösung benötigten Zeit in Sekunden (gezählt ab dem Ende des Countdowns) und der Unterscheidung ob TMT-A oder TMT-B gelöst wurde. Dann folgen die Koordinaten der Kreise in der gelösten Reihenfolge, zusammen mit ihrer Nummerierung.

Listing 4.20: Zwei Einträge von tmt_stats.txt

```
1 12/31/2013 6:22:59 PM | 6.971352 | TMT-A | (-16.5, -7.3, 900.0)=1>(8.8,
      -2.7, 900.0)=2>(-3.0, 5.2, 900.0)=3>(-8.3, 16.7, 900.0)=4>(-21.2,
      13.0, 900.0)=5>(-1.0, 13.2, 900.0)=6>(-23.4, -6.9, 900.0)=7>(2.9, 7.6,
      900.0)=8>(-5.3, -1.4, 900.0)=9>(14.3, 8.3, 900.0)=10>
2 12/31/2013 6:23:15 PM | 9.964573 | TMT-B | (11.6, 2.9, 900.0)=1>(8.8,
      -2.7, 900.0)=A>(-9.1, -6.1, 900.0)=2>(-24.9, 6.0, 900.0)=B>(-12.6,
      12.9, 900.0)=3>(-25.6, 17.6, 900.0)=C>(-21.2, 13.0, 900.0)=4>(-18.1,
      18.6, 900.0)=D>(-1.0, 13.2, 900.0)=5>(-23.4, -6.9, 900.0)=E>
```

4.5 Aufbau der Fischer Szene in Unity

Dieser Abschnitt beschäftigt sich mit der Fischer Szene, welche in Abbildung 4.21 gezeigt wird. Das Ziel dieses Teilspiels ist es, genügend gelbe und lila Fische mit der richtigen Netzart zu fangen. Sind beide Netze zu 100% gefüllt, kehrt das Spiel automatisch zur Insel Szene zurück.

Abbildung 4.21: Die fertige Fischer Szene

Um die Fischer Szene zu starten, muss der NPC im Hafen mithilfe der Greifgeste angesprochen werden, woraufhin sich das Dialogfeld mit den rudimentären Erklärungen des Spielprinzips aus Abbildung 4.22 öffnet.

Abbildung 4.22: Der Fischer Dialog

Im nächsten Abschnitt wird dazu auf die grafische Umsetzung eingegangen, danach in Abschnitt 4.5.2 die Umsetzung der Leap Steuerung beleuchtet und anschließend in Abschnitt 4.5.3 die zusätzlich verwendeten Skripte erläutert.

4.5.1 Grafische Umsetzung

Wird die Fischer Szene das erste Mal gestartet, ist nur das Schiff mit gelbem Netz und die GUI zu sehen, wobei ein Balken jeweils den Füllstand eines Netzes angibt. Der gewählte Lila- und Gelbton haben dabei einen hohen Kontrast und sind auch für Farbenblinde gut unterscheidbar, wie bereits in Kapitel 4.1.3 thematisiert wurde.

Zuerst weiß der Spieler nicht genau wofür die Balken stehen, aber sobald nach ein paar Sekunden die Fischschwärme auf ihn zu schwimmen und er anfängt diese zu fangen, füllt sich der Balken der entsprechenden Farbe und die Zahl zählt hoch. Damit ist für den Spieler der Zusammenhang zwischen den Balken und den Fischen klar, weswegen ihm nicht noch explizit die GUI erklärt werden muss.
Da der Füllstand zusätzlich in Prozent angegeben ist und der sich füllende Balken eine feste Länge hat, erschließt sich dem Spieler selbstständig, dass die Balken zu 100% gefüllt sein müssen.

Nachdem mit dem Netz Fische gefangen werden, würde es seltsam anmuten, wenn es sich mit der Zeit nicht auch optisch füllen würde. Aus diesem Grund werden die in Abbildung 4.23 gezeigten Stufen für den Füllstand implementiert. Bei einer Füllung von 5% wird die erste, ab 50% die zweite und bei 100% die dritte Stufe aktiviert. Das entsprechende Skript wird in Abschnitt 4.5.3.4 erläutert.

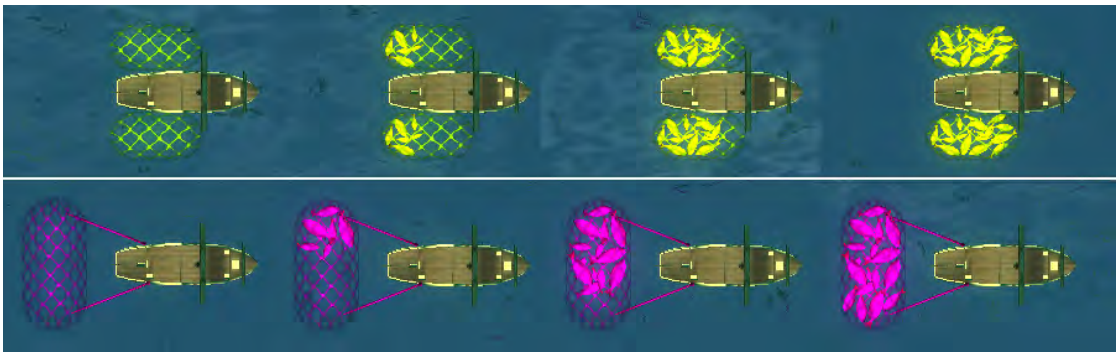

Abbildung 4.23: Die verschiedenen Füllstände der beiden Netzarten

Die Schwellenwerte spiegeln damit nicht den optischen Füllstand wieder, sondern sind so gewählt, dass der Spieler durch die niedrige untere Schwelle, schnell ein erstes Erfolgserlebnis hat. Würde die untere Schwelle z.B. erst ab 33% aktiviert, wie es optisch passender wäre, müsste der Spieler zu lange ohne diese Rückmeldung spielen.

Es wäre auch möglich, noch mehr Stufen einzubauen oder das Netz dynamischer zu befüllen, allerdings wird dies grafisch schnell zu feingliedrig und der Aufwand würde in keinem Verhältnis zum Nutzen stehen.

Auf die GUI Balken und Prozentangaben kann grundsätzlich nicht verzichtet werden, da der Spieler sonst bei Werten, die zwischen den Stufen liegen, kein klares Feedback erhält, ob sein aktuelles Verhalten positiv oder negativ für den Spielverlauf ist.

Es gibt drei Arten von Schwärmen, die für beide Farben gleich sind und von rechts auf den Spieler zu schwimmen, siehe Abbildung 4.24. Die einzelnen Schwärme unterscheiden sich in ihrem Bewegungsmuster und stellen damit unterschiedliche Anforderungen an den Spieler, was das Einsammeln oder Ausweichen angeht.

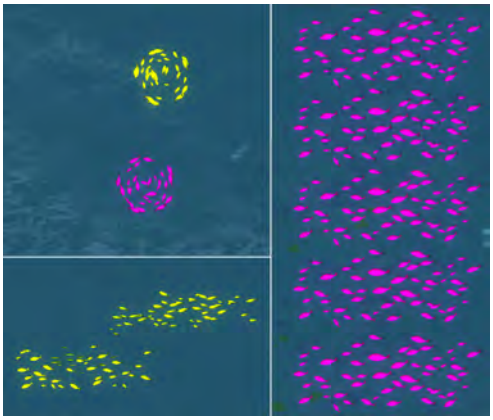

Abbildung 4.24: Die drei Schwarmarten

Die kreisförmigen Schwärme bewegen sich von rechts diagonal nach unten oder oben auf den Rand des Sichtfelds zu und wechseln an diesem die vertikale Richtung.

Die länglichen Schwärme bewegen sich ebenfalls diagonal auf den Rand zu, allerdings in einem flacheren Winkel und verschwinden, wenn sie aus dem Bild schwimmen.

Die dritte Art füllt die ganze Bildschirmhöhe und bewegt sich gleichmäßig nach links, wodurch ihr der Spieler nicht ausweichen kann, sondern das Netz zur richtigen Farbe ändern muss.

4.5.2 Leap Steuerung - L_PlayerMovement.cs

Als Steuerung für die Fischer Szene bietet es sich an, den Zeiger wie beim TMT absolut zu steuern. Damit konnten in der TMT Szene bereits positive Erfahrungen gesammelt werden und nachdem bei der Entwicklung auf Wiederverwendbarkeit von Code geachtet wird, ist es einfach die Steuerung zu adaptieren.

Die Leap Steuerung der Fischer Szene ist in der *L_PlayerMovement.cs* Klasse zu finden und dabei gleich aufgebaut, wie die in Kapitel 4.4.2 beschriebene Steuerung des TMT. Der Hauptunterschied ist die zusätzliche Notwendigkeit den Farbwechsel zu steuern.

Da die vertikale Y-Achse und horizontale X-Achse bereits mit den Zeigerbewegungen belegt sind, ist es naheliegend den Farbwechsel über die Z-Achse abzuwickeln. Dazu

wird in Listing 4.21 ausgewertet, ob der Finger über den festgelegten Schwellenwert zurück gezogen wird und entsprechend die Farbe geändert. Um Fehler zu vermeiden, kann die Farbe erst dann erneut gewechselt werden, wenn der Finger wieder weit genug nach vorne bewegt wird.

Listing 4.21: L_PlayerMovement.cs - Farb-/Netzwechsel per Leap

```
1 if(pointable.TipPosition.z > 80 && !fingerChanged) {
2     statics.switchColor();
3     fingerChanged = true;
4 }
5 else if(pointable.TipPosition.z < 70){
6     fingerChanged = false;
7 }
```

Wichtig ist, dass sich der Schwellenwert im sichtbaren Bereich der Leap befindet, damit beim Zurückziehen des Fingers noch Kontrolle über den Zeiger erfolgen kann. Wie groß dieser Wert gewählt wird, hängt vom Abstand des Spielers zur Leap und von der Leap zum Monitor ab. Im Falle dieser Arbeit hat sich 80 mm als geeignet erwiesen.

Ein Farbwechsel wird bei der Maussteuerung über die mittlere Maustaste, wie in Listing 4.22 initialisiert.

Listing 4.22: L_PlayerMovement.cs - Farb-/Netzwechsel per Maus

```
1 if(Input.GetMouseButtonDown(0) && !fingerChanged){
2     statics.switchColor();
3     fingerChanged = true;
4 }
5 else {
6     fingerChanged = false;
7 }
```

4.5.3 Skripte

Im Folgenden werden die zusätzlich verwendeten Skripte der Szene erläutert, angefangen mit der Erzeugung von Gegnern in Abschnitt 4.5.3.1, gefolgt von den Bewe-

gungsmustern der Schwärme in Abschnitt 4.5.3.2 und dem einsammeln von Fischen in Abschnitt 4.5.3.3 und 4.5.3.4. Anschließend wir noch der Inhalt der Auswertungsdatei in Abschnitt 4.5.3.5 besprochen.

Da die Fischer Szene ursprünglich als eine Art *Shoot 'em up* geplant war, spiegeln die Klassen und Dateinamen dies zum Teil noch wieder.

4.5.3.1 Schwarm Erzeugung - EnemySpawn.js

Die Schwärme sind wie beschrieben in drei Arten eingeteilt, die alle mit dem gleichen Skript in randomisierten Abständen erzeugt werden. Für den Zufallsfaktor bei der Erzeugung werden die Grenzen manuell festgelegt. Wären diese komplett zufällig, würde die Kontrolle über den Schwierigkeitsgrad verloren gehen und die Vergleichbarkeit zwischen mehreren Durchgängen, wäre nicht gegeben.
Die einzelnen Schwärme sind dazu als *Prefab* angelegt und erhalten ihr Bewegungsmuster, über das im nächsten Abschnitt beschriebene Skript.

In Listing 4.23 ist der vollständige Code der *EnemySpawn.js* Klasse zur Schwarm Erzeugung zu sehen. Die Prefabs werden dem Skript per Drag & Drop, mithilfe der öffentlichen `enemyToSpawn` Variablen in der 2. Zeile zugewiesen. Über `spawnDelayMin` und `spawnDelayMax` wird der Grenzbereich für die Zufallszahlen festlegt.
Mit `isBottom` wird bei Schwärmen mit vertikaler Bewegung festgelegt, ob sich der Schwarm zuerst nach unten oder oben bewegt und `firstSpawn` legt fest, wann der Schwarm zum ersten mal erzeugt wird.

In der ersten Skript Version, wurden die Schwärme jedes mal durch `Instantiate()` in Unity angelegt und beim Einsammeln durch `Destroy()` wieder zerstört. Da die Schwärme aus sehr vielen kleinen Objekten bestehen, die dazu noch eigene Collider besitzen, hat dies die in Kapitel 2.1.7.3 beschriebenen Performance-Probleme mit sich gebracht.

Um die Performance zu verbessern wird auf Pooling zurückgegriffen, damit jedes Objekt nur einmal instantiiert werden muss. Danach wird das Objekt somit lediglich deaktiviert und bei erneuter Aktivierung wieder an seine Ursprungsposition zurück versetzt.

Listing 4.23: EnemySpawn.js - Erzeugung von Schwärmen

```
#pragma strict
function Start () {
    spawnPoint = Vector3(offsetX, 0, 20);
    nextSpawnTime = Time.time + firstSpawn;
}
function Update () {
    if((spawnedEnemy == null || spawnedEnemy.active == false) && Time.time
        > nextSpawnTime){
        nextSpawnTime = Time.time + Random.Range(spawnDelayMin,
            spawnDelayMax);

        if(spawnedEnemy != null){
            spawnedEnemy.transform.position = spawnPoint;
            spawnedEnemy.SetActive(true);
            spawnedEnemy.SetActiveRecursively(true);
        }
        else {
            spawnedEnemy = Instantiate(enemyToSpawn, spawnPoint,
                Quaternion.identity);
        }

        if(!isBottom){
            spawnedEnemy.transform.rigidbody.velocity.x *= -1;
        }
    }
}
```

Es ist ersichtlich, dass dadurch jeweils ein *EnemySpawn.js* Skript einen Schwarm verwaltet und pro Skript auch nur ein Schwarm gleichzeitig sichtbar ist. In der vorliegenden Arbeit sind diese so verteilt, dass maximal 10 Schwärme gleichzeitig sichtbar sein können: Vier Runde, vier Längliche und zwei Bildschirmfüllende Schwärme, jeweils zu gleichen Teilen gelb und lila.

Die `firstSpawn` Werte sind so gewählt, dass zuerst nur ein einzelner runder Schwarm erscheint, danach ein zweiter runder, danach ein länglicher usw., so dass der Spieler sich langsam an die neuen Bewegungsmuster gewöhnen kann und die Schwierigkeit

stetig steigt. So soll die Szene auch optisch immer fordernder werden.
Durch den Zufallsfaktor kann keine perfekte Lösung auswendig gelernt werden, wodurch sich jede Runde anders spielt.

4.5.3.2 Schwarm Bewegung - EnemyMovementCS.cs

Die einfachste Art Objekte per Skript zu bewegen ist, in der `Update()` Methode die Position über `transform.position` zu ändern, weshalb dies auch zuerst so umgesetzt wurde.

Die Performance der Fischer Szene ist allerdings eine ebenso große Herausforderung, wie die Performance der Insel Szene. Dies mag angesichts der schlichteren Grafik überraschen, hat aber wie im vorigen Abschnitt und in Kapitel 2.1.6.2 erwähnt, mit den vielen kleinen Objekten und ihren Collidern zu tun.
Werden diese direkt über die Positionsänderung bewegt, muss Unity ständig Neuberechnungen des Physiksystems vornehmen. Die Lösung liegt in der Verwendung der *RigidBody* Komponente.

In Falle dieser Implementierung enthält das Elternobjekt eines Schwarmes einen Rigidbody, der nicht von Gravitation beeinflusst wird und nicht kinematisch ist. Wie in Listing 4.24 Zeile 5-8 sichtbar, erhält dieser beim Start Bewegungsenergie nach links und gegebenenfalls nach oben oder unten.

Listing 4.24: EnemyMovementCS.cs - Bewegung eines Schwarms

```
1  void Start() {
2      Vector3 vel = transform.rigidbody.velocity;
3      vel.x = xSpeed;
4      vel.z = -zSpeed;
5      transform.rigidbody.velocity = vel;
6  }
7  void Update () {
8      if(transform.position.x > statics.topBorder || transform.position.x <
           statics.bottomBorder){
9          Vector3 vel = transform.rigidbody.velocity;
10         vel.x *= -1;
```

```
11          transform.rigidbody.velocity = vel;
12
13          if(transform.position.x > statics.topBorder){
14              transform.position = new Vector3(transform.position.x - 0.1f,
                    transform.position.y, transform.position.z);
15          }
16          else if(transform.position.x < statics.bottomBorder){
17              transform.position = new Vector3(transform.position.x + 0.1f,
                    transform.position.y, transform.position.z);
18          }
19      }
20      if(transform.position.z < statics.leftBorder-5){
21          gameObject.SetActive(false);
22      }
23  }
```

Durch die gleichmäßige, von Unity gesteuerte Bewegung, können intern Voraussagen über die nächste Position getroffen und Neuberechnungen der Physik-Engine optimiert werden.

Erreichen Schwärme mit vertikaler Bewegung den Bildschirmrand, wird dies in `Update()` überprüft und deren vertikale Bewegung umgekehrt. Zusätzlich wird die vertikale Position noch etwas verschoben, um Fehler bei Objekten die sich genau auf dem Rand bewegen, zu vermeiden. Diese bleiben sonst durch eine ungünstige Position „hängen“ ohne sich vom Rand wieder lösen zu können.

Verlässt das Objekt links das Bild, wird es deaktiviert und später durch EnemySpawn.js wieder zurückgesetzt und aktiviert.

Prinzipiell wäre es performanter gewesen, die einzelnen `Update()` Funktionen der Schwärme zentral zu kapseln. Allerdings haben sich die Performance-Einbußen im Profiler als vernachlässigbar gegenüber anderen Optimierungen, wie dem Einsatz von Rigidbodies, heraus gestellt.

Optimal ist die Performance dieser Szene aber dennoch nicht, da es bei leistungsschwächeren PCs trotz Optimierungen zu FPS Einbrüchen beim Zurücksetzen der Position kommen kann.

4.5.3.3 Fische fangen - Shooter_PlayerCollision.cs

Fische werden mit dem *Shooter_PlayerCollision.cs* Skript aus Listing 4.25 gefangen, weswegen die Netze jeweils eine eigene Skriptinstanz und einen eigenen *Box Collider* in der Größe der Netzgrafik enthalten. Damit werden Fische auch nur in dem Bereich gefangen, in dem das Netz zu sehen ist.

Da die Fische jeweils einen Collider mit `isTrigger = true` als Komponente haben, kann über `OnTriggerEnter()` die Kollision erfasst und über das *Tag* die Art des Fisches festgestellt werden. Anschließend wird die Füllstandsvariable in *Shooter_GameStatics.cs* geändert, der einzelne Fisch deaktiviert und die Netzgrafiken bei Bedarf, über die im nächsten Abschnitt erläuterte `.handleNetFish()` Methode geändert.

Listing 4.25: Shooter_PlayerCollision.cs - Fische fangen

```
1  void OnTriggerEnter (Collider other) {
2      if(other.tag == "EnemyLight"){ // purple
3          if(statics.isPlayerLight && statics.netFilledPurple < 100){
4              statics.netFilledPurple += 0.1f;
5              other.gameObject.SetActive(false);
6          }
7          else if(!statics.isPlayerLight){
8              if(statics.netFilledYellow > 0){
9                  statics.netFilledYellow -= 0.15f;
10             }
11             other.gameObject.SetActive(false);
12         }
13     }
14     else if(other.tag == "EnemyDark"){ // yellow
15         // [...]
16     }
17     statics.handleNetFish();
18 }
```

4.5.3.4 Netzfüllung - Shooter_GameStatics.cs

Die *Shooter_GameStatics.cs* Klasse verwaltet gemeinsame Variablen der Szene, wie den Füllstand der Netze, zeichnet die GUI, speichert die Auswertungsdaten und ändert die Netzfarbe sowie deren Füllung.

Listing 4.26 zeigt, wie in Zeile 2-5 die GUI aktualisiert und anschließend die Grafik der Netzfüllung geändert wird. Die Fischgruppen, welche die Netze füllen, nehmen optisch jeweils ein Drittel des Raums ein und werden in `toggleXThirdNet()` durch `.SetActive(true/false)` aktiviert oder deaktiviert.

Da die NGUI Balken jeweils einen Slider haben, der für die optische Befüllung zuständig ist und sich stufenweise von 0 (leer) bis 11 (voll) regeln lässt, werden die Werte in Zeile 2 und 3 entsprechend angepasst.

Listing 4.26: Shooter_GameStatics.cs - Netzfüllung ändern

```
1  public void handleNetFish(){
2      purpleSlider.sliderValue = Mathf.Floor(netFilledPurple/10)/10;
3      yellowSlider.sliderValue = Mathf.Floor(netFilledYellow/10)/10;
4      purpleLabel.text = Mathf.Floor(netFilledPurple) + " %";
5      yellowLabel.text = Mathf.Floor(netFilledYellow) + " %";
6
7      if(netFilledYellow >= 100){
8          toggleOneThirdNet(true, true);
9          toggleTwoThirdNet(true, true);
10         toggleThreeThirdNet(true, true);
11     }
12     else if(netFilledYellow >= 50) {
13         toggleOneThirdNet(true, true);
14         toggleTwoThirdNet(true, true);
15         toggleThreeThirdNet(false, true);
16     }
17     else if(netFilledYellow >= 5) { /* [...] */ }
18     else { /* [...] */ }
19
20     if(netFilledPurple >= 100){  // [...]
21 }
```

```
22 private void toggleOneThirdNet(bool status, bool isYellow){
23     if(isYellow){
24         foreach(GameObject net in oneThirdNetYellow){
25             net.SetActive(status);
26         }
27     }
28     else {
29         oneThirdNetPurpleObj.SetActive(status);
30     }
31 }
```

Ebenfalls in der *Shooter_GameStatics.cs* findet der Farb- und Netzwechsel mithilfe des Codes in Listing 4.27 statt. Dazu haben die Elternobjekte der Netze jeweils ein eigenes Tag, über das sie gefunden und aktiviert bzw. deaktiviert werden. Ebenfalls wird der Zeitpunkt des Farbwechsels zwischengespeichert, um bei Spielende in die Auswertungsdatei geschrieben zu werden.

Listing 4.27: Shooter_GameStatics.cs - Farb-/Netzwechsel

```
1  void Start(){
2      darkNet = GameObject.FindWithTag("DarkNet");
3      lightNet = GameObject.FindWithTag("LightNet");
4      // [...]
5  }
6  public void switchColor(){
7      writeToFileString.Add("["+System.DateTime.Now+"]switch");
8      if(isPlayerLight){
9          isPlayerLight = false;
10         darkNet.SetActive(true);
11         lightNet.SetActive(false);
12     }
13     else {
14         isPlayerLight = true;
15         darkNet.SetActive(false);
16         lightNet.SetActive(true);
17     }
18 }
```

4.5.3.5 Auswertungsdatei - Shooter_GameStatics.cs

Die Auswertungsdatei dieser Szene heißt *fischer_stats.txt* und wird, sehr ähnlich wie in Abschnitt 4.3.3.2 Listing 4.8 gezeigt, mit Informationen zum Farbwechsel, dem Füllstand der Netze und der benötigten Gesamtzeit befüllt. Der Füllstand beider Netze wird jeweils in 5% Schritten gespeichert.

Ein exemplarischer Auszug dieser Datei ist in Listing 4.28 zu sehen. Die Werte werden während des laufenden Spiels zwischengespeichert und erst am Ende in die Datei geschrieben, um die Performance nicht zu beeinträchtigen.

Dabei steht `switch` für den Farbwechsel, `netfill` für den Füllstand beider Netze und `win` für die gewonnene Runde.

Listing 4.28: Inhalt der Fischer Auswertungsdatei

```
1  //new session//
2  [1/6/2014 6:51:59 PM]netfill|purple:0|yellow:5
3  [1/6/2014 6:52:03 PM]switch
4  [1/6/2014 6:52:05 PM]switch
5  [1/6/2014 6:52:09 PM]switch
6  [1/6/2014 6:52:09 PM]switch
7  [1/6/2014 6:52:10 PM]netfill|purple:2|yellow:10
8  [1/6/2014 6:52:11 PM]switch
9  [...]
10 [1/6/2014 6:53:05 PM]netfill|purple:74|yellow:95
11 [1/6/2014 6:53:06 PM]netfill|purple:74|yellow:100
12 [1/6/2014 6:53:08 PM]netfill|purple:74|yellow:95
13 [1/6/2014 6:53:08 PM]netfill|purple:74|yellow:90
14 [1/6/2014 6:53:09 PM]switch
15 [...]
16 [1/6/2014 6:53:27 PM]netfill|purple:90|yellow:100
17 [1/6/2014 6:53:28 PM]netfill|purple:95|yellow:100
18 [1/6/2014 6:53:28 PM]win|time:92.81628
```

5

Fazit der Anforderungen

In der folgenden Tabelle wird für die Anforderungen aus Kapitel 3 betrachtet, ob diese erfolgreich umgesetzt werden konnten.

Es wird unterschieden, ob die Anforderungen *Erfüllt* bzw. *Teilweise Erfüllt* wurden oder dies weiterhin *Offen* ist.

Tabelle 5.1: Ergebnis der Anforderungen

Name	**Titel**	**Status**	**Beschreibung**
FA 1	Exekutivfunktionen	Offen	Konnte nicht untersucht werden
FA 2	Leap Motion	Erfüllt	Alle Szenen im Spiel werden komplett per Leap gesteuert
FA 3	3D-Grafik	Erfüllt	Die Grafik aller Szenen ist 3D

FA 4	Auswertung	Erfüllt	Jede Szene speichert Daten
FA 5	TMT Spiel	Erfüllt	Die TMT Szene ist implementiert
FA 6	Fischer Spiel	Erfüllt	Die Fischer Szene ist implementiert und auf Switching ausgelegt
FA 7	Freie Welt	Erfüllt	Die Insel wird frei erkundet
FA 8	Szenario	Erfüllt	Das Szenario ermöglicht fließende Übergänge zwischen den einzelnen Szenen
FA 9	Aufträge	Erfüllt	Die Teilspiele werden per Dialog gestartet
NFA 1	Landschaft	Erfüllt	Die Insel hat Landmarken die die Orientierung erleichtern und bietet verschiedene Grafiksets wie Wald, Strand oder Wiese
NFA 2	Geschichte	Teilweise Erfüllt	Pflanzen zu sammeln wird als Aufgabe nicht erklärt
NFA 3	Teilspieldauer	Erfüllt	Keines der Teilspiele dauert im Durchschnitt länger als 5 Minuten
NFA 4	Teilspielart	Erfüllt	Sowohl die TMT als auch Fischer Szene sind einfach zu verstehen
NFA 5	Teilspiele	Erfüllt	Die TMT und Fischer Szene haben klar kommunizierte Ziele
NFA 6	Benutzeroberfläche	Erfüllt	Die GUI ist für die Tester hilfreich
NFA 7	Steuerung	Teilweise Erfüllt	Wie flüssig die Steuerung ist, hängt maßgeblich von den FPS der Szene ab
NFA 8	Performance	Teilweise Erfüllt	Die Fischer Szene hat noch Performance-Einbrüche

6

Ausblick

In diesem Kapitel wird auf zukünftige Möglichkeiten und Erweiterungen dieses Projektes eingegangen und anschließend die Arbeit kurz zusammengefasst.

6.1 Erweiterungsmöglichkeiten

Nachdem das Anwendungsszenario eine grundlegende Unterweisung durch einen Psychologen vorsieht, gibt es relativ wenig Erklärungen zur Steuerung oder dem gesamten Spielziel. Sollte **Where are my Pirates** daher separat veröffentlicht werden, sind mehr Erklärungen sowohl zum Inhalt und der Steuerung als auch zu den erfassten Daten nötig. Zudem sollte das Spiel dann eine Highscore enthalten, damit der Trainingseffekt selbstständig festgestellt werden kann.

Ebenfalls ist in den Szenen bisher kein Sound vorhanden, lediglich auf der Insel verursachen die Möwen ein Geräusch, weil es in ihrer Skriptvorlage bereits fertig implementiert war. Hier kann das Spielerlebnis dementsprechend noch stark über Hintergrundmusik und Soundeffekte bereichert werden. Auch lassen sich positive und negative Aktionen in der TMT oder Fischer Szene, durch entsprechende Soundeffekte noch deutlicher anzeigen und als zusätzliches Feedback nutzen.

Bei der Umsetzung in Unity ist ebenso noch Verbesserungspotenzial im Bereich der Grafik und Performance vorhanden. Gerade die Fischer Szene ist auf schwächeren Systemen noch nicht dauerhaft flüssig spielbar.

6.1.1 Testmöglichkeiten

Wie im Grundlagen Kapitel, den Anforderungen und in der Implementierung dargestellt, wurde **Where are my Pirates** auf ein Training der Exekutivfunktionen hin entwickelt. Es war aber leider nicht möglich zu testen, wie hoch der Trainingseffekt des digitalen Trail Making Test, gerade im Vergleich zur analogen Version tatsächlich ist. Dafür wären zur Untersuchung mehrere Testpersonen und zur Betreuung, Befragung und Auswertung mindestens ein Psychologe nötig gewesen, was den Rahmen dieser Arbeit überstiegen hätte.

Dennoch sind mögliche Unterschiede des Trainingseffektes zwischen der digitalen und analogen Variante sowie dem digitalen TMT mit Maus- oder mit Leap Motion Steuerung, ein nach Meinung des Autors interessantes Forschungsgebiet.
Gerade wenn sich eine Version als geeigneter erweisen würde, könnten weitere Untersuchungen der zugrunde liegenden Faktoren lohnenswert sein.

6.1.2 Leap Motion

Bereits angekündigt für zukünftige Versionen des Leap Motion Treibers ist eine Verbesserung des Trackings, indem die Finger nicht mehr einzeln, sondern die Hand als ganzes Objekt erfasst wird [leab]. Damit kann die Leap anhand der übrigen Daten auf die Position von Fingern schließen, auch wenn sie diese nicht einzeln erkennen

kann. Dadurch sollen eine höhere Präzision und neue Gesten, wie überkreuzte Finger ermöglicht werden.

Wie im Grundlagen Kapitel erwähnt, ist die Kalibrierung und feste Position der Leap im Verhältnis zum Monitor für eine hohe Genauigkeit des Trackings nötig, was mit einem separaten Gerät schwierig umzusetzen ist. Inzwischen kooperiert Leap Motion allerdings mit Herstellern wie *HP* und *ASUS*, um unter anderem Notebooks, mit fest eingebauter Leap zu vertreiben, wobei erste Modelle sind bereits auf dem Markt [leac] [leaa]. Dies könnte die Notwendigkeit manueller Kalibrierung verringern und eröffnet neue Möglichkeiten der absoluten Steuerung, da die eingebaute Leap bereits auf die Monitorgröße vorkalibriert verkauft werden könnte. Zusätzlich könnte dies versehentlicher Bedienung der Leap vorbeugen, da auch die Position der Tastatur im Verhältnis zur Leap bekannt ist.
Generell wäre auch ein Einbau der Leap Sensorik in eine Tastatur denkbar, um beide Steuerungskonzepte in einem Gerät zu verbinden.

6.1.3 Oculus Rift

Ebenfalls könnte der Einfluss von stereoskopischem 3D einer Videobrille auf das Training der Exekutivfunktionen untersucht werden. Mit der *Oculus Rift* von *Oculus VR* steht ein Modell mit besonders intensiver Inversion bereit. Nach Meinung des Autors, welcher in seiner Nebentätigkeit an einem Oculus Rift Projekt arbeitet, ist diese bereits in der Entwicklerversion sehr ausgereift und übertrifft bisherige Videobrillen durch die niedrige Latenz bei Kopfbewegungen und das große Sichtfeld.

Für Unity ist ein fertiges Asset Paket von Oculus VR verfügbar, welches alle nötigen Skripte und Objekte enthält. Es müssen dafür einige Grafik Einstellungen angepasst und das Oculus Kamera Prefab in die Szene integriert werden, um die stereoskopische Ansicht zu erhalten. Abbildung 6.1 zeigt das Dorf in der von der Rift benötigten Ansicht.

Zu beachten ist, dass einige Standard Assets zusammen mit der Oculus Rift nicht mehr funktionieren. So haben das Wasser der Insel Szene sowie die die Szene umschließende

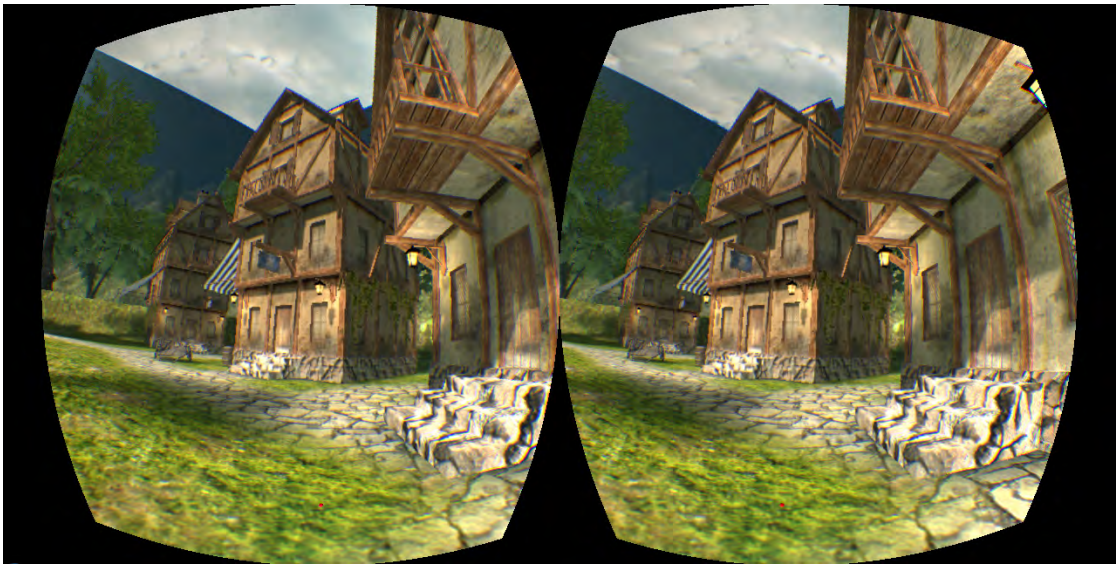

Abbildung 6.1: Das Dorf mit angepasster Oculus Rift Optik

Skybox[1] Grafikfehler und alle GUI Elemente müssen speziell für die Verwendung angepasst und aufbereitet werden, da diese nicht mehr nur eine Kamera als Referenzpunkt für die Ausrichtung haben. Für diese Probleme sind aber bereits Lösungen entwickelt worden, so gibt es z.B. spezielle Skybox Assets für die Oculus Rift Verwendung und Anleitungen um NGUI anzupassen.

Ebenfalls fallen falsche Größenverhältnisse der eigenen Figur zu den umliegenden Gebäuden und NPCs sehr stark auf, wenn die Rift verwendet wird. Dies könnte das größere Hindernis bei einer Anpassung darstellen, da dadurch die Größe von Objekte aller Szene überprüft werden muss.

Ein weiter Hindernis, ist eine stabile Framerate, die durchgehend bei 60 FPS liegen muss, da sonst die Verzögerungen zwischen Kopfbewegung und Bild vom Spieler wahr genommen wird [ocu]. Wie der Autor aus eigener Erfahrung berichten kann, führt dies rasch zu starker Bewegungsübelkeit.

[1] Ein die Szene umschließender Würfel, auf den nahtlose Grafiken projiziert werden, die als Himmel fungieren.

Ein Test mit der Oculus Rift würde deshalb genug Herausforderungen für eine eigene Abschlussarbeit bereitstellen, könnte aber neue Erkenntnisse im Bereich der Videobrillen und Exekutivfunktionen ermöglichen.

6.2 Zusammenfassung

Im Rahmen dieser Diplomarbeit ist ein Unity Spiel entstanden, dass die Leap Motion zur Steuerung eines Serious Games einsetzt und auf ein Training der Exekutivfunktionen hin entwickelt wurde, auch wenn die Frage wie hoch der Trainingseffekt ist weiterhin offen bleiben muss. Dabei wurde eine grafisch abwechslungsreiche Landschaft durch ein glaubwürdiges Szenario mit den beiden Teilspielen verknüpft, die ihre Anforderungen fast vollständig umsetzen konnten. Lediglich die Performance ist stellenweise noch ein Problem.

Es hat sich gezeigt, dass die Leap zwar noch Verbesserungspotenzial bietet, aber bereits produktiv eingesetzt werden kann, um neue Wege in der Mensch-Computer-Interaktion zu gehen. Ob die Bedienung dabei einfacher oder schwieriger ist, hängt weniger von der Leap selbst, sondern maßgeblich von einer geschickten Bewegungs- und Gestenwahl ab.

Unity hat sich als wertvolles Entwicklungswerkzeug herausgestellt, das durch umfangreiche Möglichkeiten, eine ausführliche Dokumentation und große Community besticht und dem Autor viele Arbeitsschritte stark vereinfacht hat.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literaturverzeichnis

[Cha10] CHARSKY, Dennis: From Edutainment to Serious Games: A Change in the Use of Game Characteristics. In: *Games and Culture* 5 (2010)

[cre] *Creative Commons - Attribution-NonCommercial-ShareAlike 4.0 International - CC BY-NC-SA 4.0*. http://creativecommons.org/licenses/by-nc-sa/4.0/, Abruf: 2014-01-10

[Csi10] CSIKSZENTMIHALYI, Mihaly: *Das flow-Erlebnis: Jenseits von Angst und Langeweile: im Tun aufgehen*. 11. Stuttgart : Klett-Cotta, 2010

[DFS01] DEMIRIS, George ; FINKELSTEIN, Stanley M. ; SPEEDIE, Stuart M.: Considerations for the design of a Web-based clinical monitoring and educational system for elderly patients. In: *Journal of the American Medical Informatics Association* 8 (2001), Nr. 5, S. 468–472

[GPSR13] GEIGER, Philip ; PRYSS, Rüdiger ; SCHICKLER, Marc ; REICHERT, Manfred: Engineering an Advanced Location-Based Augmented Reality Engine for Smart Mobile Devices / University of Ulm. Version: October 2013. http://dbis.eprints.uni-ulm.de/972/. Ulm : University of Ulm, October 2013 (UIB-2013-09). – Technical Report

[Huh04] HUHN, Dr. G.: *Evolutionäres Management*. Wien : Verlag für Systemisches Management, 2004

[leaa] *Asus Laptops and Desktops to Ship With Gesture Control Thanks to New Deal With Startup Leap Motion | MIT Technology Review*. http://www.technologyreview.com/news/509491/

asus-laptops-and-desktops-to-ship-with-leap-motions-gesture-control/, Abruf: 2014-01-10

[leab] *How Leap Motion tracks what it can't see | Internet & Media - CNET News*. http://news.cnet.com/8301-1023_3-57613090-93/how-leap-motion-tracks-what-it-cant-see/, Abruf: 2014-01-10

[leac] *HP Envy 17 Leap Motion Special Edition review (Wired UK)*. http://www.wired.co.uk/reviews/laptops/2014-01/hp-envy-17, Abruf: 2014-01-10

[lead] *Leap Motion*. https://www.leapmotion.com/product, Abruf: 2013-08-05

[leae] *Leap Motion | 3D Motion and Gesture Control for PC & Mac*. https://www.leapmotion.com/product, Abruf: 2014-01-05

[leaf] *Leap Motion | Press for Our Gesture Control Technology*. https://www.leapmotion.com/press_releases, Abruf: 2013-11-19

[leag] *Leap Motion Developer | Creating Leap apps with Unity (standard license) < Articles*. https://developer.leapmotion.com/articles/creating-leap-apps-with-unity-standard-license, Abruf: 2013-11-03

[leah] *Leap Motion Developer | Developer App Documentation*. https://developer.leapmotion.com/docs, Abruf: 2013-11-03

[leai] *Leap Motion Developer | Leap Overview < SDK Documentation*. https://developer.leapmotion.com/documentation/Languages/CSharpandUnity/Guides/Leap_Overview.html, Abruf: 2013-11-03

[leaj] *Leap Motion Developer | Setting up a Unity project < SDK Documentation*. https://developer.leapmotion.com/documentation/Languages/CSharpandUnity/Guides/Setup_Unity.html, Abruf: 2013-11-03

[leak] *leapjs*. http://js.leapmotion.com/tutorials/leapSpace, Abruf: 2013-11-19

[Leal] *pohung / LeapFirstPersonGrab — Bitbucket*. https://bitbucket.org/pohung/leapfirstpersongrab, Abruf: 2014-01-10

[leam] *Using the Leap Motion Diagnostics Test : Leap Motion Support*. http://support.leapmotion.com/entries/

24900981-Using-the-Leap-Motion-Diagnostics-Test, Abruf: 2014-01-10

[NTT+12] NOUCHI, Rui ; TAKI, Yasuyuki ; TAKEUCHI, Hikaru ; HASHIZUME, Hiroshi ; AKITSUKI, Yuko ; SHIGEMUNE, Yayoi ; SEKIGUCHI, Atsushi ; KOTOZAKI, Yuka ; TSUKIURA, Takashi ; YOMOGIDA, Yukihito ; KAWASHIMA, Ryuta: Brain Training Game Improves Executive Functions and Processing Speed in the Elderly: A Randomized Controlled Trial. In: *PLOS ONE* 7 (2012), 01, Nr. 1. http://dx.doi.org/10.1371/journal.pone.0029676. – DOI 10.1371/journal.pone.0029676

[ocu] *Oculus Rift developer on required system specs, potential new functionality | PC Gamer*. http://www.pcgamer.com/2013/01/11/oculus-rift-developer-on-required-system-specs-potential-new-functionality/, Abruf: 2014-01-10

[RNMR10] RÖTHLISBERGER, Marianne ; NEUENSCHWANDER, Regula ; MICHEL, Eva ; ROEBERS, Claudia M.: Exekutive Funktionen: Zugrundeliegende kognitive Prozesse und deren Korrelate bei Kindern im späten Vorschulalter. In: *Zeitschrift für Entwicklungspsychologie und Pädagogische Psychologie* 42 (2010), Nr. 2, S. 99–110

[spaa] *Leap Motion Teardown - Learn.SFE*. https://learn.sparkfun.com/tutorials/leap-motion-teardown/you-got-guts-kid, Abruf: 2013-08-05

[spab] *Leap Motion Teardown - Learn.SFE*. https://learn.sparkfun.com/tutorials/leap-motion-teardown/you-got-guts-kid, Abruf: 2013-08-05

[SSP+13] SCHOBEL, Johannes ; SCHICKLER, Marc ; PRYSS, Rüdiger ; NIENHAUS, Hans ; REICHERT, Manfred: Using Vital Sensors in Mobile Healthcare Business Applications: Challenges, Examples, Lessons Learned. In: *9th Int'l Conference on Web Information Systems and Technologies (WEBIST 2013), Special Session on Business Apps*, 2013, 509–518

[STK07] SEIFERTH, N. Y. ; THIENEL, R. ; KIRCHER, T.: Funktionelle MRT in Psychiatrie und Neurologie. (2007), S. 266

[SW05] SWEETSER, Penelope ; WYETH, Peta: GameFlow: A Model for Evaluating Player Enjoyment in Games. In: *Computers in Entertainment (CIE)* 3 (2005)

[Tom04] TOMBAUGH, Tom N.: Trail Making Test A and B: Normative data stratified by age and education. In: *Archives of Clinical Neuropsychology* 19 (2004), Nr. 2, 203-214. http://dx.doi.org/10.1016/S0887-6177(03)00039-8. – DOI 10.1016/S0887–6177(03)00039–8

[unda] *Unity - 3D formats*. http://docs.unity3d.com/Documentation/Manual/3D-formats.html, Abruf: 2013-08-02

[undb] *Unity - Physics*. http://docs.unity3d.com/Documentation/Manual/Physics.html, Abruf: 2013-08-02

[undc] *Unity - Practical Guide to Optimization for Mobiles - Optimizing Scripts*. http://docs.unity3d.com/Documentation/Manual/iphone-PracticalScriptingOptimizations.html#Object%20Pooling%20Example, Abruf: 2013-08-02

[undd] *Unity Script Reference:*. http://docs.unity3d.com/Documentation/ScriptReference/CharacterController.OnControllerColliderHit.html, Abruf: 2013-08-02

[unia] *Asset Store - 3D Models*. https://www.assetstore.unity3d.com/#/category/0, Abruf: 2013-07-27

[unib] *Theory and Principles of Game Design: Controlling Aspect Ratio in Unity*. http://gamedesigntheory.blogspot.de/2010/09/controlling-aspect-ratio-in-unity.html, Abruf: 2013-12-31

[unic] *Unity - Asset Store Terms of Service and EULA*. http://unity3d.com/company/legal/as_terms, Abruf: 2014-01-10

[unid] *Unity - Draw Call Batching*. http://docs.unity3d.com/Documentation/Manual/DrawCallBatching.html, Abruf: 2014-01-10

[unie] *Unity - Getting Started with iOS Development*. http://docs.unity3d.com/Documentation/Manual/iphone-GettingStarted.html, Abruf: 2014-01-10

[unif] *Unity - Physics - Built-in physics for immersive, action-packed games*. http://unity3d.com/unity/quality/physics, Abruf: 2014-01-10

[unig] *Unity - Physics overview*. http://docs.unity3d.com/Documentation/Components/DynamicsGroupOverview.html, Abruf: 2014-01-10

[unih] *Unity - Special Folders and Script Compilation Order*. http://docs.unity3d.com/Documentation/Manual/ScriptCompileOrderFolders.html, Abruf: 2014-01-10

[unii] *Unity - Submitting your content to the Asset Store*. http://unity3d.com/asset-store/sell-assets, Abruf: 2014-01-10

[unij] *Unity - Trees*. http://docs.unity3d.com/Documentation/Components/terrain-Trees.html, Abruf: 2014-01-10

[unik] *Unity Script Reference:*. http://docs.unity3d.com/Documentation/ScriptReference/Collider.html, Abruf: 2014-01-10

Name: Alexander Fürgut Matrikelnummer: 640046

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Alexander Fürgut