HOCHSCHULE
MITTWEIDA
University of Applied Sciences

Fakultät Angewandte Computer- und Biowissenschaften

Professur Medieninformatik

Bachelorarbeit

Entwurf und Implementation einer interaktiven VR-Komponente für die Visualisierung von Entscheidungsbäumen in Unity

Andre Spittel

Mittweida, den 11. März 2020

Erstprüfer: Ritter, Marc (Prof. Dr. rer. nat)
Zweitprüfer: Vogel, Richard (Dipl.-Inf.)

Spittel, Andre
Entwurf und Implementation einer interaktiven VR-Komponente für die Visualisierung von Entscheidungsbäumen in Unity
Bachelorarbeit, Fakultät Angewandte Computer- und Biowissenschaften
Hochschule Mittweida — University of Applied Sciences, März 2020

Referat

Ziel dieser Arbeit ist der Entwurf einer Komponente, welche in der Lage ist Entscheidungsbäume in den 3D-Raum zu projizieren. Dazu ist es erforderlich, diese erweiterbar zu gestalten. Im Speziellen muss sie dafür verschiedene Arten von Knoten und Entscheidungsregeln mithilfe einer erweiterbaren Schnittstelle unterstützen. Die Schnittstelle ist als Teil der Arbeit zu definieren und zu dokumentieren. Die Komponente soll prototypisch Interaktionen des Nutzers registrieren, wie das Anwählen einzelner Bestandteile des Modells. Um die Verwendbarkeit in Folgeprojekten sicherzustellen muss zu diesem Zweck im Rahmen der Arbeit eine API definiert werden.

Name: Spittel, Andre
Studiengang: Medieninformatik und interaktives Entertainment
Seminargruppe: MI16w1-B
English Title: Design and implementation of an interactive VR-Component for visualization of Decision Trees in Unity.

Inhaltsverzeichnis

1 Einführung und Motivation **1**

1.1 Aufgabenstellung 1
1.2 Aufbau der Arbeit 2

2 Grundlagen **3**

2.1 Maschinelles Lernen 3
2.2 Entscheidungsbäume 5
2.3 Virtual Reality 6
2.4 Unity 7
2.5 Datenformate und verwendete Frameworks 9

3 Anforderungen und Spezifikation **13**

3.1 Funktionale Anforderungen 13
3.2 Nicht funktionale Anforderungen 14

4 Implementierung **15**

4.1 Übersicht der Programmstruktur 15
4.2 Darstellung des Baumes als Klasse 18
4.3 Erstellung von Testdaten 21
4.4 Importierung der externen Daten 24
4.5 Initialisierung der Knoten und Kanten 26
4.6 Platzierung der Knoten und Kanten 28
4.7 Durchfluss und Auswertung der Datensätze 32
4.8 Interaktion im virtuellen Raum 33

5 Nutzerleitfaden **37**

6 Evaluation **41**

6.1 Versuchsbeschreibung 41
6.2 Auswertung 42
6.3 Ergebnis 44

7 Zusammenfassung und Ausblick **45**

Literaturverzeichnis **I**

1 Einführung und Motivation

Software, als Lösung zu alltäglichen Problemen ist heutzutage nicht mehr wegzudenken. Ob es nun die nächste Bahnverbindung in der App der Deutschen Bahn ist, oder der Taschenrechner in der Schule. Sie dient als Erweiterung unser Selbst, sodass wir uns um viele Dinge keine Gedanken mehr machen müssen, und unsere Kraft auf wichtigeres lenken können. Vielen ist es ein Rätsel wie Programme in Computern funktionieren, so ist es für den normalen Verbraucher schon fast Magie. Und eine Thematik in der Software scheint sogar für viele Informatiker Magie zu sein, das maschinelle Lernen. Unter diesem Begriff stellt sich fast jeder etwas anderes vor. Viele haben eine künstliche Intelligenz vor Augen, die mit Menschen eine Konversation führen kann. Andere denken an die Gesichtserkennung, die heutzutage Standard auf modernen Handys ist. Doch alle haben eins gemeinsam, sie denken, dass die Algorithmen des maschinellen Lernens schwierig zu verstehen sind. Es stimmt zwar bei den genannten Beispielen, jedoch ist nicht jeder Algorithmus schwer zu verstehen. Ein verständliches Modell im Bereich des maschinellen Lernens ist der Entscheidungsbaum.

1.1 Aufgabenstellung

Aufgabe dieser Arbeit ist es, eine Komponente zu programmieren, die es ermöglicht, maschinelle Lernmodelle eines Entscheidungsbaumes aus einer JSON-Datei einzulesen, und in VR in der Engine Unity zu visualisieren. Dabei ist für die Arbeit vorgesehen, ein spezielles Format für diese JSON vorher zu definieren. Es können also keine beliebigen Modelle von Entscheidungsbäumen eingelesen werden, sondern diese müssen dem Format entsprechen. Die Visualisierung stellt den Entscheidungsverlauf eines solchen Baumes dar, sodass einfach nachvollzogen werden kann, was im Modell des Baumes eigentlich passiert, und welche Faktoren ausschlaggebend für die getroffene Entscheidung waren. Angezeigt soll dies durch eine Benutzeroberfläche sein. Der Fokus soll hier darin liegen, den Grundstein für die Weiterarbeit zu legen.

Da der Umfang dieser Arbeit eingegrenzt werden muss, wird nur eine prototypische Interaktion in VR implementiert, um die Implementierung auf weitere Funktionen vorzubereiten.

Das Ziel wird sein, das Programm im Bereich der Bildung dazu zu nutzen, hinter die Kulissen vom Entscheidungsprozess des Algorithmus zu schauen, um eine interaktive und spannende Einführung in die Welt des maschinellen Lernens zu bekommen.

1.2 Aufbau der Arbeit

Im nachfolgenden Kapitel 2 werden die Grundlagen erläutert, die benötigt werden, um die programmierte Arbeit zu verstehen. Teil davon sind das maschinelle Lernen, und eine Form dessen, die Entscheidungsbäume. Hauptfokus liegt hier vor allem auf den Aufbau von Bäumen, und den Grund, wieso in dieser Arbeit ein Entscheidungsbaum visualisiert wird. Anschließend wird der Begriff „VR“ erläutert, und dessen Vorteile. Es wird erklärt, was eine Spiele-Engine, und Unity eigentlich ist, und dessen Features erläutert, die bei der Implementierung verwendet wurden. Zum Schluss der Grundlagen werden einige Begrifflichkeiten zu den verwendeten Datenformaten geklärt.

In Kapitel 3 werden die funktionalen, und nicht funktionalen Anforderungen konkretisiert, und im Kapitel 4 wird dargelegt, wie nun diese Anforderungen implementiert wurden. Kapitel 5 gibt Einblick darin, wie dieses Tool nun eigentlich zu nutzen ist, und in Kapitel 6 wird die vollführte Evaluation beschrieben. Die Arbeit wird in Kapitel 7 durch eine Zusammenfassung mit Ausblick beendet.

2 Grundlagen

In diesem Kapitel werden die benötigten Grundlagen erläutert, um den Inhalt der Arbeit zu verstehen. Es geht vor allem auf die Aspekte der Technologien ein, welche für diese Arbeit relevant sind. In der Sektion des maschinellen Lernens wird grob erklärt was es ist und wo es angewendet wird. Auch wird auf die Lernverfahren eingegangen. Anschließend wird auf die in der Arbeit verwendete Form des maschinellen Lernens eingegangen, die Entscheidungsbäume. Der Fokus liegt hier vor allem im Aufbau und der Vorteile. Folgend wird die Technologie des *Virtual Reality* erläutert, und daraufhin die Spiele-Engine Unity, welche für die Implementation der Arbeit verwendet wurde. Zum Abschluss dieses Kapitels werden die Datenformate dargelegt, die zur Implementation verwendet werden.

2.1 Maschinelles Lernen

Dieser Abschnitt dient dazu, dem Leser grundlegende Aspekte des maschinellen Lernens klar zu machen. Die Thematik ist sehr umfangreich, sodass hier nur grob einige Funktionsweisen erklärt werden können.

Maschinelles Lernen ist einem Teilbereich der Informatik, der Künstliche Intelligenz (KI), anzusiedeln. Sie versucht bestimmte Muster oder Regeln in vorgegebenen Datenmengen automatisiert zu finden. Die gewonnene Information kann dann auf weiteren Daten mit ähnlicher Struktur und Aufbau angewendet werden, um neue oder effizientere Lösungen zu erreichen [RN12]. Anders ausgedrückt, geht es in diesem Wissenschaftsfeld also darum, durch Lernalgorithmen Modelle aus Datensätzen zu entwickeln. Diese Modelle können auf neue Datensätze derselben Art angewandt werden um beispielsweise bestimmte Voraussagungen zu treffen [WH18]. Solche Modelle zu trainieren kostet Zeit, Aufwand und vorhandene Datensätze, daher wird das maschinelle Lernen nur in bestimmten Fällen angewandt. Vor allem wenn Prozesse

zu komplex sind, um sie analytisch zu beschreiben. So wie beispielsweise die Gesichtserkennung. Es ist etwas, das jeden Tag von Menschen automatisch gemacht wird, egal in welcher Haltung, Frisur, oder Beleuchtung. Aber es ist etwas, was nur sehr schwierig beschreiben ist. Trotzdem hat das Gesicht aber bestimmte Merkmale. Wie zum Beispiel die Symmetrie, die Nase, die Augen, der Mund. Es gibt also bestimmte Muster die durch eine Kombination bestimmter Komponente erkannt werden kann. Durch Beispielbilder einer Person, können Lernprogramme eben diese Muster erkennen, und damit andere Bilder auf dieses Muster untersuchen. Dies ist ein typisches Beispiel der Mustererkennung des maschinellen Lernens [Alp19, S. 2].

Im Grunde lassen sich maschinelle Lernverfahren in drei verschiedene Arten unterteilen [BS19, S. 9].

- **Supervised Learning** (überwachtes Lernen): Dies ist ein Verfahren, mit dem ein Modell mit vielen beschrifteten Daten trainiert wird. Hier werden beispielsweise dem Algorithmus tausende Hunde- und Katzenbilder gegeben, mit der Informationen was nun Hund und was nun Katze ist. Dadurch kann der Algorithmus dann Muster erkennen, und auf andere Bilder anwenden.
- **Unsupervised Learning** (unüberwachtes Lernen): Hier werden versucht, Muster in bestehenden Daten zu finden, ohne dem Algorithmus vorher zu sagen was richtig oder falsch ist. Um das vorherige Beispiel fortzusetzen, würde es hier also bedeuten dem Algorithmus tausende Bilder von Hunden und Katzen zu geben, ohne ihm zu sagen was nun was ist. Hier muss der Algorithmus also selbst die Kategorisierung vornehmen. Wie die Bilder dann aber nun kategorisiert werden, ist Problem und Chance zugleich. So kann er es in Hund und Katze unterscheiden, oder in helles Fell oder dunkles.
- **Reinforcement Learning** (verstärkendes Lernen) Grundlage in diesem Lernverfahren ist eine Belohnungsfunktion, und sie dient dazu für ein Problem eine optimale Strategie zu finden. Dem Algorithmus wird also wieder nicht gezeigt, welche Aktion die beste ist. Er bekommt hier eine Rückmeldung für seine Aktionen, in Form einer Belohnung, oder einer Strafe. Dadurch versucht der Algorithmus die optimalen Aktionen zu finden, und kann diese mit der Rückmeldung optimieren.

Das maschinelle Lernen gibt außerordentliche Ergebnisse mit sich. Medizinische Diagnosen werden dadurch genauer, Autos lassen sich mit ihnen besser fahren.

Trotzdem gibt es aber einige Problematiken mit diesen Algorithmen, welche von Buxmann und Schmidt [BS19, S. 10] gut erklärt werden. Einerseits, obwohl sie den Menschen statistisch überlegen sind, können sie trotzdem Fehler machen. Und diese Fehlentscheidungen können, sofern man die Parameter des Algorithmus kennt, vom Menschen sogar mit Absicht hervorgeführt werden. Dies führt zu einem klaren Sicherheitsproblem. Auch herrscht bei solchen Algorithmen häufig das sogenannte Black-Box-Problem. Was bedeutet, dass oft unklar ist, wie und weshalb nun der Algorithmus genau diese Wahl getroffen hat. Zum Kontext dieser Arbeit, führt das Black-Box-Problem also auch vor allem dazu, dass solche Algorithmen schwer zu erklären sind. Jedoch gibt es eine Art und Weise, solche Entscheidungsprozesse gut erklärlich darzustellen, die Entscheidungsbäume.

2.2 Entscheidungsbäume

Diese Sektion der Arbeit behandelt weniger die technischen Aspekte, wie die Algorithmen und deren Formeln, sondern mehr den Aufbau, die Funktionsweise und Vorteile den die Entscheidungsbäume anderen Modellen gegenüber haben.

Entscheidungsbäume sind eine der einfachsten, und doch erfolgreichsten Formen des maschinellen Lernens [RN12, S. 814]. Der Entscheidungsbaum liefert zu Objekten, welche durch Mengen von Attribut/Wertpaaren beschrieben sind, jeweils eine Entscheidung, welcher Klasse das betreffende Objekt zuzuordnen ist [BKI03, S. 107]. Sie bestehen aus einem Wurzelknoten, Kanten, interne Knoten und Blätter, wie in Abbildung 2.1 aufgezeigt.

Beim Klassifizieren, stellt der Wurzelknoten den Start dar. Dann wird der Baum durchlaufen, wobei jeder Knoten eine logische Abfrage darstellt, die sogenannten Splitkriterien. Das Splitkriterium setzt sich aus einer Splitting-Variable und einem Cutoff zusammen. Dies wird solange rekursiv durchgangen, bis ein Blattknoten erreicht wird, welcher die Klasse darstellt [CIL13]. Solche Bäume können binär sein, was bedeutet, dass jeder Knoten nur zwei Kindknoten haben kann, oder auch mehrere Kindknoten haben. Dies hängt davon ab, welcher Algorithmus bei der Erstellung des Baumes genutzt wurde. Einige solcher Algorithmen sind unter anderem CART, ID3 und C4.5.

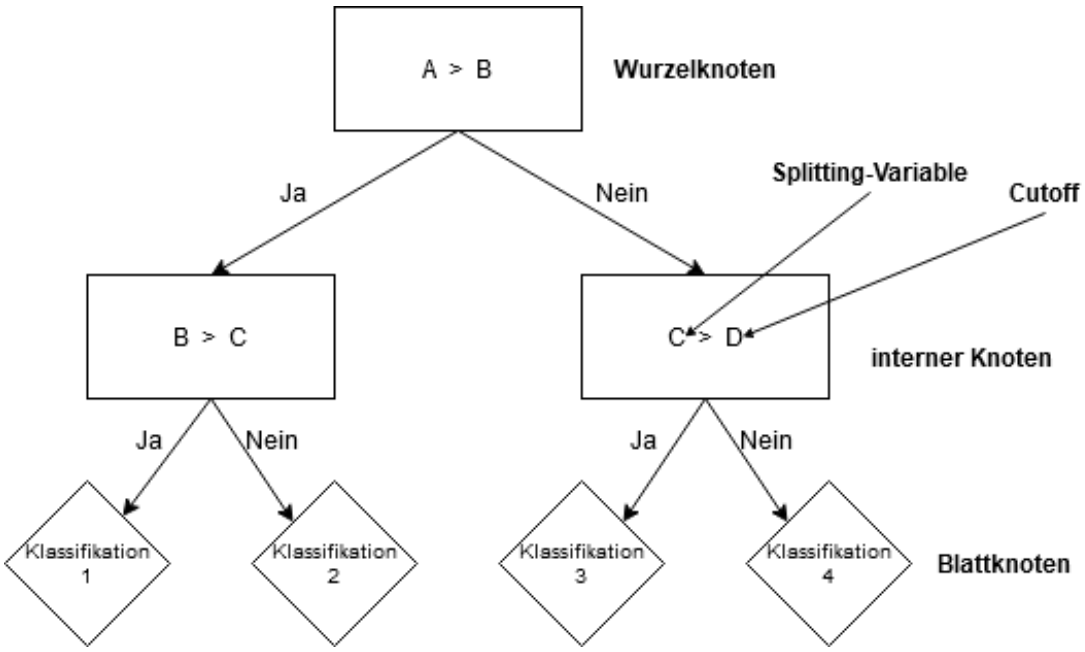

Abbildung 2.1: Visualisierung eines Entscheidungsbaums nach [CIL13]. Hervorgehoben sind hier der Wurzelknoten, interner Knoten, Blattknoten, Splitting-Variable und der Cutoff.

Dass ein Entscheidungsbaum als Projekt zur Visualisierung gewählt wurde, hat einige Hintergründe. So ist einer ihrer größten Vorteile die Transparenz. Zu jedem Punkt des Entscheidungsprozesses eines Baumes ist klar, wie, und weshalb diese Entscheidung gefällt wurde. Vor allem bei Datensätzen mit vielen Attributen können bei den Anwendern zusätzliche Erkenntnisse hinsichtlich der Bedeutung dieser Attribute für bestimmte Entscheidungen gewonnen werden. Auch kann der Entscheidungsprozess durch die Transparenz nachträglich korrigiert werden [GS10]. Das alles führt dazu, dass Entscheidungsbäume eine Form des maschinellen Lernens bieten, welche leicht zu verstehen sind. Sie formen eine exzellente Einführung in die Welt des maschinellen Lernens, und sind auch durch ihre Vorteile und Struktur leicht zu visualisieren.

2.3 Virtual Reality

Die Idee von *Virtual Reality* (weiterhin als VR abgekürzt) stand schon vor einer langen Zeit im Raum. Schon im Jahre 1965, träumte Ivan E. Sutherland von einem

„mathematischen Wunderland“, in dem „Konzepte umgesetzt werden können, die in der echten physischen Welt nicht umsetzbar sind“. Eine Technologie, welche nicht nur Eingaben von Geräten wie Tastaturen entgegennimmt, sondern unseren ganzen Körper [Iva65]. Obwohl die Vision da war, brauchte es noch viele Jahre bis die erste kommerziell brauchbare Technologie entwickelt wurde. In 2012 kam ein Kickstarter-Projekt mit dem Namen „Oculus Rift“ ans Licht, mit der Idee, eine hochqualitative *Head-Mounted-Display* (weiterhin als HMD abgekürzt) zu einem erschwinglichen Preis auf dem Markt zu bringen. Die Nachfrage war so stark. dass das Ziel des Kickstarters in unter 24 Stunden erreicht wurde. Demnach gibt es heutzutage also ein enormes Interesse an diese Technologie.

Generell, wird VR von Kuhlen als Darstellung, und gleichzeitige Wahrnehmung der Wirklichkeit und ihrer physikalischen Eigenschaften in einer Echtzeit, interaktiven Software definiert [Kuh14, S. 134]. Die am weitesten verbreitete Variante, um VR erleben zu können, ist durch die Technologie der HMDs. Eine HMD ist ein Helm, oder eine Brille, mit zwei eingearbeiteten Bildschirmen. Die Art der Bildschirme ist von Modell zu Modell unterschiedlich. Die HMD wird durch eine bestimmte Technologie im Raum verfolgt, sodass jegliche Rotation- und Translationsänderungen des Spielers im Raum auch in der Software übernommen werden können. Die Art des sogenannten *Trackings* ist hier auch von Modell zu Modell unterschiedlich[Bri09, S. 6]. Kuhlen nennt hier zwei bedeutende Elemente von VR, die Immersion, und die Interaktivität, welche für dieses Projekt ausschlaggebende Faktoren waren, um VR zu verwenden. Studien zeigten, dass VR große Vorteile im Bereich der Bildung bringt [Boy17]. Gerade durch die Visualisierung in 3D fällt es Lernenden einfacher abstrakte Themengebiete zu begreifen, und durch die Immersion und Interaktivität bleiben sie unterhalten.

2.4 Unity

Die Programmierung von Spielen, oder Software im Allgemeinen welche Rendering, Shading und Physikinteraktionen benötigt erfordert ein unglaubliches Maß an Aufwand [BEW+98]. Daher sind in den letzten Jahrzenten mehrere Frameworks entstanden, welche genau diese Problematiken erleichtern, die sogenannten Spiel-Engines. Sie vereinfachen den Prozess zur Erstellung derartiger Software durch schon eingebaute Features. Einige der bekannten öffentlich zugänglichen Engines

sind Unreal, Unity und die CryEngine. Zur Bearbeitung der Arbeit wurde sich für Unity entschieden.

Unity wurde von Unity Technologies entwickelt um interaktive Medien zu kreieren. In Unity ist als Standart die nVidia Physik-Engine, und eine eigenes 3D-Rendering integriert. Große Merkmale dieser Engine sind die sehr gute Dokumentation, die riesige Community, und das sehr intuitive Interface des Editors [CBM08]. Diese Merkmale waren ein ausschlaggebender Faktor zu Wahl der Engine.

In Unity wird mit sogenannten *Assets* gearbeitet, welche Dateien im Projekt darstellen. Szenen in Unity sind eine besondere Art von Asset, welche die gesamte Umgebung oder Level des Projektes beschreiben. Ein Objekt kann nur als sogenanntes *GameObject* in der Szene existieren, was bedeutet, dass alles in der Szene ein *GameObject* ist. *GameObjects* können verschiedene Komponenten haben, in Form von Skripten [ND17]. Wie solch eine eigene Komponente aussehen kann, ist in Abbildung 2.2 zu begutachten.

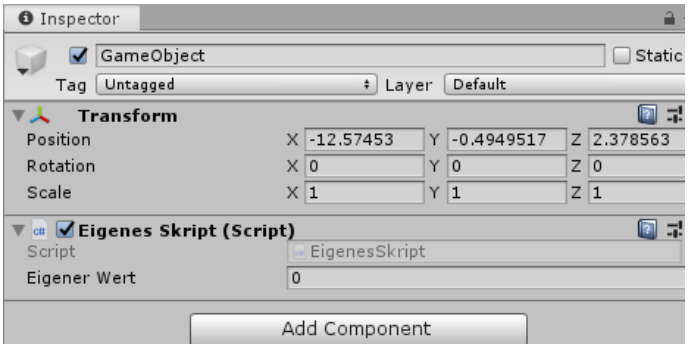

Abbildung 2.2: Beispiel einer Komponente in Unity

Zu sehen sind zwei Komponente, angehangen an einem *GameObject*. Einmal die *Transform*-Komponente, und einmal ein eigenes Skript mit dem Namen *Eigenes Skript*. Jedes *GameObject* in einer Szene hat immer eine *Transform*-Komponente, welche die Position, Rotation und Skalierung als jeweils einen dreidimensionalen Vektor widerspiegelt. In der Abbildung ist auch eine Variable *Eigener Wert* zu sehen. Diese ist im Code als eine *public* Variable definiert, sodass also innerhalb des Editors die Variablen einer Komponente geändert werden können, ohne diese nochmal im Quellcode ändern zu müssen.

Für Instanziierungen innerhalb eines Skriptes, oder um Berechnungen jeden *Frame* auszuführen, gibt es in Unity spezifische Funktionen die verwendet werden können.

Für das Ausführen von Code beim Programmstart, kann die Funktion *Start()* verwendet werden. Um Funktionen sogar vor *Start()* aufzurufen, gibt es *Awake()*. Dies wird beispielsweise benutzt, um Variablen zu instanziieren. Die *Update()*-Funktion ist dafür da, um Berechnungen in jedem *Frame* auszuführen.

Wenn ein *GameObject* mit den gewünschten Komponenten erstellt wurde, kann dieses als sogenanntes Prefab außerhalb der Szene, im Assetordner gespeichert werden. Ein Prefab stellt einen Schnappschuss eines *GameObjects* mit seinen Komponenten, Variablenwerten und Kindobjekten dar [ND17]. Sie sollten immer in Fällen benutzt werden, wenn man ein Objekt mehrfach benutzen will, was zu den meisten Objekten eines Projektes zutrifft.

Unity liefert einiges an Hilfe wenn es um VR Projekte geht, und die meisten Hersteller arbeiten eng mit Unity zusammen um Plugins zu schaffen, die das Arbeiten mit VR für Entwickler einfacher machen, wie zum Beispiel das von Valve entwickelte Plugin SteamVR. Es baut auf der OpenVR API (Application Programming Interface) auf, welches eine Schnittstelle zwischen Hardware und Anwendung bietet. Sofern die VR-Brille den OpenVR Treiber installiert hat, kann diese von SteamVR angesprochen, und identifiziert werden. Damit bietet das SteamVR Package in Unity ein Interface für Entwickler, um sehr einfach Informationen wie Position von Controller oder Buttonevents auslesen zu können [SG19].

2.5 Datenformate und verwendete Frameworks

Für dieses Projekt musste ein Format festgelegt werden, mit dem das Modell und die Datenmatrix serialisiert werden. In diesem Abschnitt wird beschrieben, was genau JSON und CSV sind, und weshalb diese verwendet wurden. Anschließend wird noch erläutert, wie mit welchen Frameworks diese Serialisierung stattfand.

JSON (*JavaScript Object Notation*) ist ein Textformat, das zum Serialisieren von strukturierten Daten genutzt wird. JSON kann sehr einfach primitive Datentypen wie *strings*, *booleans*, Zahlen, *null*, und sogar strukturierte Typen wie Arrays oder Objekte speichern [B+14]. Es ist sehr einfach von Mensch oder Maschine zu lesen, und sehr weit verbreitet, weshalb es sehr viele verfügbare Packages dafür gibt [PRS+16]. Für die Arbeit wurde das Newtonsoft „Json.Net“[1] Framework verwendet.

[1] Siehe https://www.newtonsoft.com/json

Wie in Abbildung 2.3 visualisiert, bietet es eine erstklassige Performance. Durch die Popularität des Frameworks gibt es bei Problemen auch eine Vielzahl von Online Ressourcen, und eine ausgiebige Dokumentation.

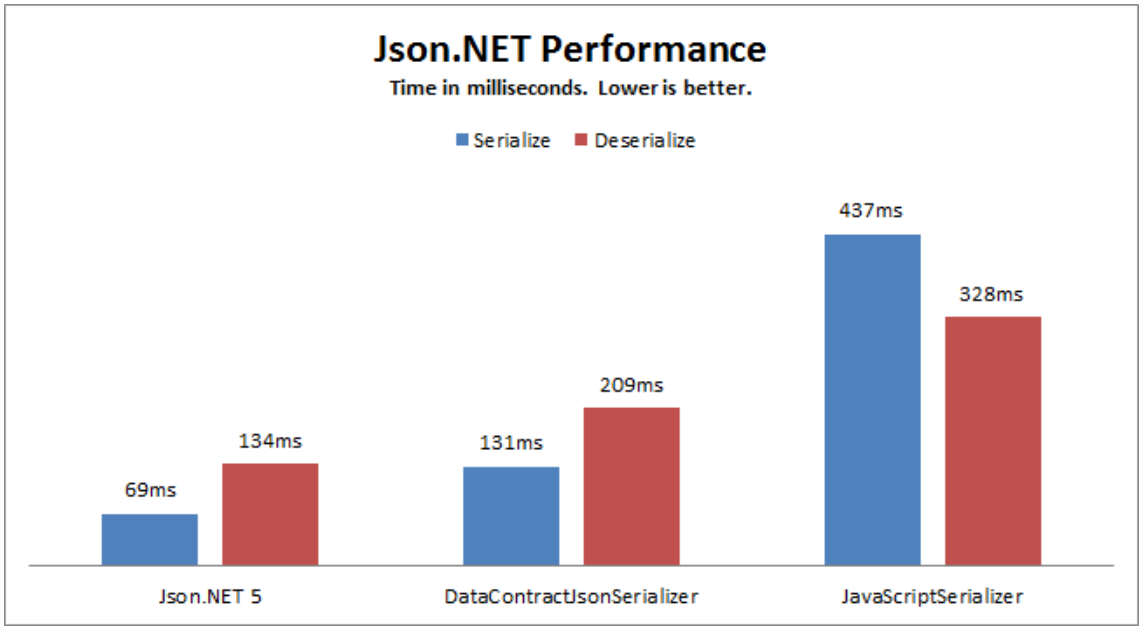

Abbildung 2.3: Vergleich von verschiedenen JSON Frameworks [New].

Für die Visualisierung war es erforderlich, Daten in großen Mengen in Echtzeit kategorisch einlesen zu können, um dann die Entscheidung des Baumes anhand dieser Daten treffen zu können. Eine einfache Lösung für solche Dateien ist das CSV-Format (Comma Seperated Values). Allgemein betrachtet, besteht eine CSV Datei aus Datensätzen, bei denen die Daten mit Kommas getrennt werden, und die Datensätze mit einem Zeilenumbruch getrennt gespeichert werden. Optional kann hier die erste Zeile als Kopfzeile fungieren, und damit ein Format mit Zeilen und Spalten erzeugen, was in Abbildung 2.4 betrachtet werden kann. Dadurch, dass es einfach im Programm zu lesen und schreiben ist, und auch sehr einfach vom Menschen zu verstehen ist, wurde sich für dieses Format entschieden [Sha05]. Für die Implementation wurde das Framework „CsvHelper“[2] verwendet. Durch dessen integrierten Funktionalitäten, können CSV-Dateien sehr einfach eingelesen, und bearbeitet werden.

[2]Siehe https://joshclose.github.io/CsvHelper/

L18

	A	B	C	D
1	age	size	IQ	weight
2	59	1,409036	62	103
3	28	1,210112	60	91
4	64	1,484329	109	51
5	20	1,849588	52	66
6	20	1,401491	98	117
7	25	1,427776	57	94
8	54	1,332166	79	105
9	60	1,268852	74	106
10	55	1,665718	52	113

Abbildung 2.4: Beispiel einer CSV-Datei für die Visualisierung.

3 Anforderungen und Spezifikation

In der Softwareentwicklung ist es essenziell vor der Produktion Anforderungen an das Programm zu setzen, und diese bei der Implementation zu beachten. In diesem Kapitel werden die Anforderungen von funktionaler und nicht funktionaler Natur beschrieben. Funktionale Anforderungen sind Anforderungen, welche Aussagen über eine Funktion treffen, und nicht funktionale Anforderungen beschreiben eine Eigenschaft des Programmes. [Bal10]

3.1 Funktionale Anforderungen

- F1. **JSON Integration**: Es muss ein Modul vorhanden sein, dass ein Modell eines Entscheidungsbaums im JSON-Format einlesen kann, und intern speichern kann.
- F2. **CSV Integration**: Es muss eine Datenmatrix in Form einer CSV-Datei eingelesen werden können.
- F3. **Initialisierung des Baumes** Jeder Knoten soll mit seinen Kanten in der Szene in Unity existieren.
- F4. **Interaktionsmöglichkeiten im virtuellen Raum** Der Baum soll mit seinen Knoten darauf vorbereitet sein, später Interaktionen mit dem Nutzer im virtuellen Raum zuzulassen. Hierfür soll also eine einfache Form der Interaktion implementiert werden, um später zusätzliche Features solcher Art einfach hinzuzufügen.
- F5. **Einheitlichkeit des Baumes** Jeder visualisierte Knoten soll genügend Platz für seine Kindknoten zur Verfügung haben. Um dies zu gewährleisten soll die Platzierung der Knoten also in einer adäquaten Form stattfinden.

- F6. **Auswertbarkeit der Daten durch das Modell** Der Baum soll die eingelesene Datenmatrix, und das eingelesenen Modell dazu nutzen, um Entscheidungen anhand dieser Datensätze treffen zu können. Knoten des Baumes müssen in der Lage sein jede Art von Splitkriterien auswerten zu können.
- F7. **Sichtbarkeit des Datenflusses** Die Knoten sollen anhand einer Nutzeroberfläche anzeigen, wie viele Daten durch sie selbst durchgeflossen sind, und wie diese Entscheidungen gefällt wurden.

3.2 Nicht funktionale Anforderungen

- NF1. **Zuverlässigkeit** Die Entscheidungen, welche durch die Knoten des Baumes gefällt werden, müssen korrekt sein. Wenn ein Datensatz, oder ein Splitkriterium eines Knotens nicht korrekt eingelesen wurde, muss der restliche Baum trotzdem weiter visualisiert werden.
- NF2. **Beachtung der Motion Sickness** Viele VR-Applikationen unterliegen der Problematik der Motion Sickness. Hierbei soll darauf geachtet werden dieses Phänomen so gut wie möglich zu dämmen.
- NF3. **Erkenntlichkeit der Informationen** Die visualisierten Informationen der Knoten und Kanten müssen dem Benutzer erkenntlich präsentiert werden. Hierbei ist Skalierung und Positionierung zu beachten.

4 Implementierung

In diesem Kapitel wird näher auf die Implementierung der Software eingegangen. Zuerst gibt es eine grobe Übersicht der Programmstruktur, welche in verständliche Funktionspackete geteilt wurde, und in Kombination in UML-Form dargestellt wird. Danach werden die Gedanken hinter der Softwarearchitektur der Baumklasse dargelegt. Es wird hier der Gedankenprozess vorgestellt, der mit der Kreierung der Klasse einherging. Folgend wird beschrieben, womit das Modell des Entscheidungsbaumes und die Datenmatrix eingelesen wird. Anschließend werden Details zur Visualisierung gegeben, und Interaktionsmöglichkeiten im virtuellen Raum geklärt.

4.1 Übersicht der Programmstruktur

Folgend werden die Funktionalitäten des Projektes als Funktionsblöcke präsentiert. Sie stellen jeweils ein Funktionspacket im Projekt dar. Die Funktionsblöcke werden später im Kapitel im Zusammenhang mit den Klassen in einem UML Diagramm dargelegt. Eine Klasse im Programm kann mehrere Funktionsblöcke beinhalten, und ein Funktionsblock kann auch durch mehrere Klassen ausgefüllt werden. Sie sind in Abbildung 4.1 je nach Bearbeitungsstatus farblich als grün, für implementiert, gelb, für teilimplementiert, und rot für nicht implementiert gekennzeichnet.

Anschließend werden die einzelnen Funktionsblöcke zusammenfassend erläutert.

- **Import des Modells:** Um einen Entscheidungsbaum zu visualisieren, muss sich darum gekümmert werden, dass das Modell in einem ordentlichen Format zur Verfügung liegt, eingelesen wird, und anderen Klassen global zur Verfügung stehen kann. Dies ist der Funktionalität F1, JSON Integration, zugeordnet.
- **Import der Datenmatrix:** Der visualisierte Entscheidungsbaum nimmt eine Datenmatrix aus einer CSV-Datei. In diesem Block wird die Funkt ionalität F2, CSV Integration, ausgeführt.

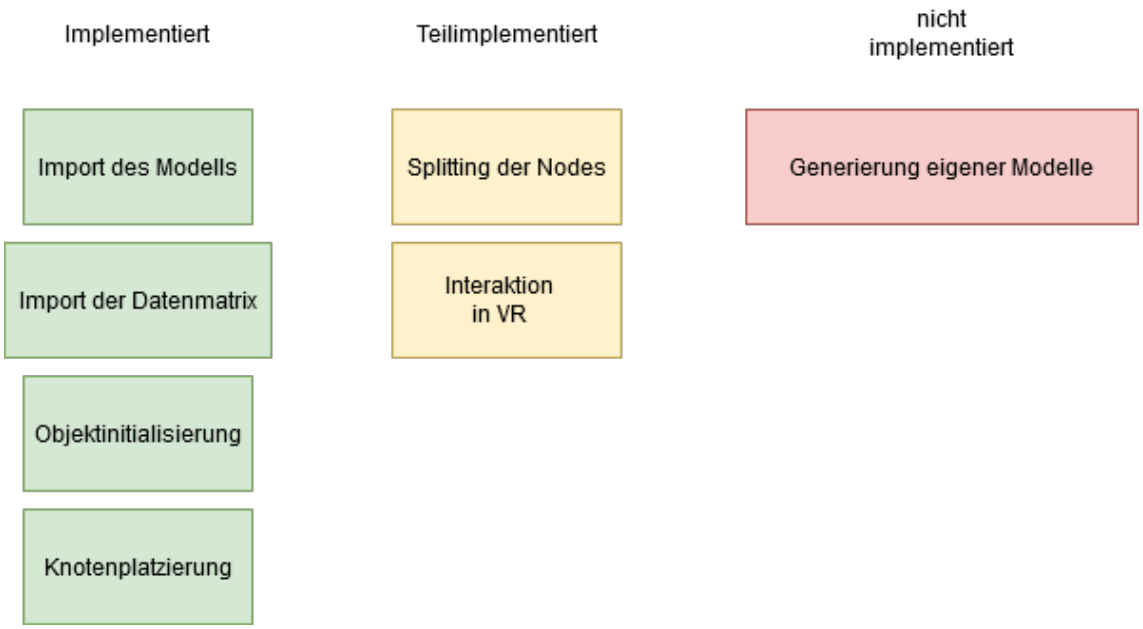

Abbildung 4.1: Übersicht der Funktionsblöcke, unterteilt in grün für implementiert, gelb für teilimplementiert, und rot für nicht implementiert.

- **Objektinitialisierung:** Dieser Block stammt aus der Funktionalität F3, Initialisierung des Baumes. Hier werden alle Knoten, mit ihren Kanten in die Unityszene platziert, und mit ihren Daten anhand des JSON Modells initialisiert. Auch wird durch die Knoteninitialisierung ebenfalls das Userinterface so eingestellt, dass die Sichtbarkeit des Datenflusses, F7, gewährleistet wird.
- **Knotenplatzierung:** Nach der Initialisierung, werden die Knoten so platziert, dass genügend Platz für die Kindknoten existiert um F5, die Einheitlichkeit des Baumes, zu garantieren. Anschließend werden die Anfangs- und Endpositionen der Kanten festgelegt.
- **Splitting der Nodes:** In diesem Block, wird die Auswertbarkeit der Daten durch das Modell, F6, implementiert. Jeder Knoten empfängt eine Datenmatrix, und wertet diese nach dem eigenen Splitkriterium aus. Anschließend wir die geteilte Datenmatrix an die Kindknoten übertragen. Dieser Block ist bisher nur teilimplementiert.
- **Interaktion in VR:** Der Funktionsblock der Interaktion in VR, wurde größtenteils durch das Packet SteamVR von Valve implementiert, und unterliegt der Funktion F4, Interaktionsmöglichkeiten im virtuellen Raum. Es wurde

eine Beispielinteraktion mit dem Baum implementiert, welche Knoten bei Berührung rot färbt. Das Hauptaugenmerk für diese Arbeit galt hier darin, das Fundament für zukünftige Interaktionen zu bilden, weshalb es daher nur teilimplementiert ist.

- **Generierung eigener Modelle:** Die Funktionalität, selber einen Algorithmus zur Modellentwicklung eines Entscheidungsbaumes zu implementieren kam während der Entwicklung in Frage, wurde jedoch aus Zeitgründen nicht implementiert.

Im UML-Diagramm in Abbildung 4.2 ist erkenntlich, wo diese einzelnen Funktionsblöcke in der Systemarchitektur eingebettet sind. Die Initialisierung, und Knotenplatzierung wird von der Manager-Klasse *VisualizationManager* übernommen. Auch übernimmt diese Klasse den Fluss und Auswertung der Daten durch die Knoten, gekennzeichnet durch den teilimplementierten Funktionsblock „Splitting der Nodes“. Das Einlesen und Bereitstellen des Entscheidungsbaummodells, und der Datenmatrix wird von der Manager-Klasse *JsonManager* übernommen. Die generellen Möglichkeiten zur Interaktion werden durch das Softwarepacket „SteamVR“ gegeben.

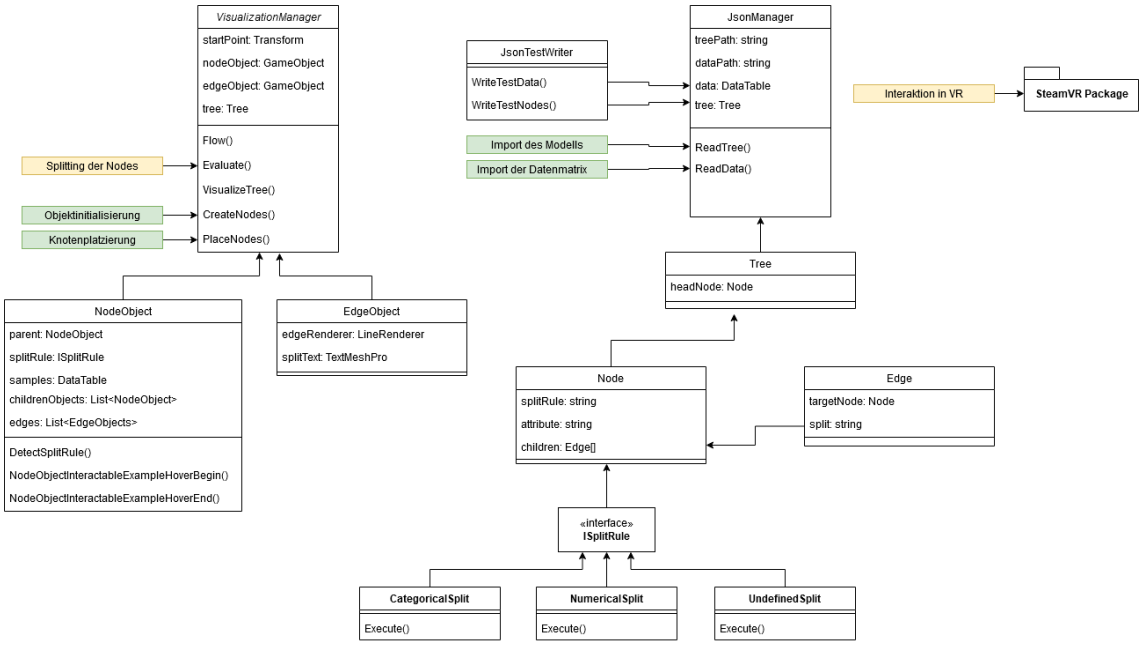

Abbildung 4.2: Klassenübersicht der Software in Form eines UML-Diagramms. Die Pfeile stehen dabei für „enthält“. Farbig gekennzeichnet sind auch die Funktionsblöcke der Software.

Die Klasse *JsonTestWriter* dient zum Erstellen von Testdaten und beliebigen Testmodellen eines Entscheidungsbaumes, welche in ihren jeweiligen Datenformaten geschrieben werden.

NodeObject und *EdgeObject* sind sogenannte „Wrapperklassen“ für die *Node* und *Edge* Klasse. Dies wird näher im nächsten Abschnitt behandelt.

4.2 Darstellung des Baumes als Klasse

Dieser Teil der Implementation ist der, der am meisten Aufwand in der Planung gekostet hat. Um die Entscheidung für die Struktur der Klasse näher zu erläutern, wird hier die Herangehensweise demonstriert, sodass die finale Struktur erst gegen Ende des Kapitels zu finden ist.

Die Hauptherausforderung in der Struktur der Baumklasse ist das Verarbeiten der Entscheidungen des Baumes, da es doch drastisch verschiedene Arten von Bäumen gibt. Eine Entscheidung die relativ einfach zu verarbeiten wäre, wäre eine boolesche Gleichung aus numerischen Attributen, wie in Abbildung 4.3 abgebildet.

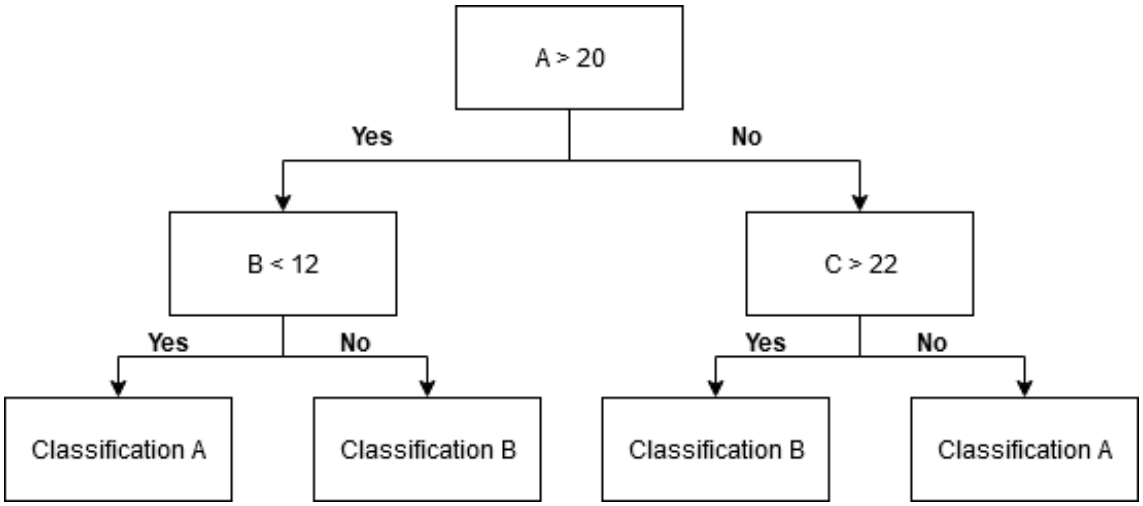

Abbildung 4.3: Entscheidungsbaum mit einfachem booleschem Vergleich, welcher im ersten Implementationsversuch als Klasse dargestellt werden soll.

Um dies zu implementieren, setzt man also einfach für das jeweilige Attribut den numerischen Wert und den booleschen Operator ein, der im Knoten definiert ist.

Wenn die Gleichung wahr ist, wird in den linken Knoten übergegangen, wenn nicht, dann in den rechten. Die Knotenklasse würde also wie in Listing 4.1 aussehen, mit zwei Variablen für die Attribute, und einen *string* für den booleschen Operator.

Listing 4.1: Einfache Knotenklasse zum Darstellen von booleschen Splitkriterien

```
1 public class Node {
2     float leftAttribute;
3     float rightAttribute;
4     string booleanOperator;
5 }
```

Für einfache Fälle wie in Abbildung 4.3 beschrieben, wäre dies auch ausreichend. Es wäre eine performante, und leicht verständliche Struktur der Knotenklasse. Jedoch soll, laut der Funktion F6, jede Art von Entscheidung durchlaufen werden können. Solch ein Splitkriterium muss nicht immer nur eine boolesche Gleichung sein. Es kann auch sein, dass der Knoten keine Attribute miteinander vergleicht, sondern beispielsweise schaut welche Art von Auto man fährt. Im weiteren Verlauf der Arbeit wird hierfür von „kategorischen Splits“ gesprochen. Mit der bisher vorgestellten Iteration der Knotenklasse wäre es unmöglich solche Art von Splits darzustellen, da nur *floats* eingelesen werden können. Also muss ein Format gefunden werden, welches verschiedene Typen von Splits entgegennehmen kann, und je nach Typ, diesen Split anders verarbeitet. Hierfür wurde das Interface *ISplitRule* mit einer *Execute()*-Methode erstellt. Dieses Interface erlaubt es die Regel für den Split zu abstrahieren, und je nachdem welches Splitkriterium für den Knoten definiert wurde, eine andere Funktion zu verwenden. Für die bisherigen zwei Fälle, den numerischen und den kategorischen Attributen, wurden also zwei *ISplitRule* Klassen erstellt, *NumericalSplit* und *CategoricalSplit*. Je nachdem welche Klasse nun für den Knoten definiert wird, wird also eine andere Variante der *Execute()*-Methode aufgerufen, welche die Daten einliest, und dann entscheidet in welchen nächsten Knoten übergegangen wird. Wie die Implementation der Splitrule aussehen kann, wird im Listing 4.2 demonstriert.

Listing 4.2: Knotenklasse, welche nun in der Lage ist verschiedene Splitkriterien auszuwerten.

```
1 public class Node {
2     string splitRule;
3     string leftAttribute;
4     string rightAttribute;
5 }
```

Durch das JSON-Format könnte man direkt ein *ISplitRule*-Objekt serialisieren, jedoch wurde sich für eine Zeichenkette entschieden, um es so allgemein wie möglich in der Modelldatei zu beschreiben, und um spätere Veränderungen zuzulassen, ohne die JSON-Dateien selbst wieder verändern zu müssen.

Ein weiteres Problem der jetzigen Iteration, ist die Annahme, dass ein Knoten nie mehr als zwei Kinder hat. Laut Funktion F4 jedoch, muss ein Knoten in der Lage sein beliebig viele Kinder haben zu können. Um dies zu lösen, ist es nichtmehr möglich von einem *rightAttribute* und *leftAttribute* auszugehen. Um von einer beliebigen Menge an Kindknoten auszugehen, muss der Knoten die Möglichkeit besitzen, beliebig viele Kanten zu anderen Knoten zu serialisieren. Hierfür muss also eine Kantenklasse erstellt werden, welche nicht nur den Zielknoten in sich speichert, sondern eben auch die Bedingung dafür, wann in den Zielknoten übergegangen werden kann. Wie dies nun final aussieht kann in Listing 4.3 betrachtet werden.

Listing 4.3: Finale implementierte Knoten- und Kantenklasse, welche verschiedenste Splitkriterien und beliebig viele Kindknoten beinhalten kann.

```
1  public class Node {
2      string splitRule;
3      string attribute;
4      Edge[] children;
5  }
6
7  public class Edge {
8      Node targetNode;
9      string split;
10 }
```

Die finale Knotenklassen speichert also nun die Art der *Splitrule* und sein Attribut als Zeichenkette und ein Array aus *Edge*-Objekten. Das *Edge*-Objekt gibt den Verweis

auf den Zielknoten, also wohin dieser Kanten nun führt, und den *split* als Zeichenkette, welcher aussagt, unter welcher Bedingung zu diesem Knoten hingeführt werden kann.

4.3 Erstellung von Testdaten

Für die spätere Nutzung der Software ist es geplant reale Modelle von Entscheidungsbäumen zu verwenden, da diese aber während der Entwicklung nicht vorlagen, aber trotzdem Modelle zum Testen benötigt werden, musste ein Skript zur automatischen Erstellung besagter Daten implementiert werden. Diese Testmodelle, und Testdatenmatrizen waren kein Teil der funktionalen Anforderungen, waren aber trotzdem benötigt, um die Software überhaupt während ihrer Entwicklung testen zu können. Diese Funktion wurde, wie im Ausschnitt des UML-Diagramms in Abbildung 4.4 gezeigt, von der Klasse *JsonTestWriter* ausgefüllt.

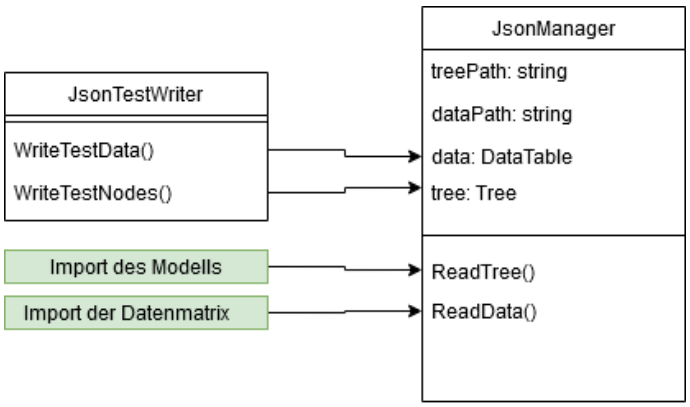

Abbildung 4.4: Ausschnitt des UML-Diagramms. Hervorgehoben wird hier die Klasse *JsonTestWriter* und dessen Bezug zum *JsonManager*.

Diese Klasse existiert als Komponente eines *GameObjects* in der Szene in Unity, und benutzt den *treePath* und *dataPath* vom *JsonManager*, um an diesem Ort die jeweiligen Dateien zu schreiben. Anschließend, so ist die Idee, sollen diese Dateien vom *JsonManager* ausgelesen werden. Hierbei ist während der Implementation aber ein typisches Problem der Ausführungsreihenfolge aufgetaucht. Um den ersten Schritt auszuführen, nämlich den Pfad vom *JsonManager* zu erhalten, muss dieser sich zuerst als Singleton instanziieren. Danach kann der Pfad aus dem Singleton ausgelesen werden und die Dateien geschrieben werden. Anschließend können diese

Dateien eingelesen werden. Dies sind drei voneinander getrennte Schritte. In Unity, gibt es aber nur zwei Funktionen mit denen man vor Beginn des Spielverlaufes Instanziierung vornehmen kann, nämlich *Start()* und *Awake()*, wobei allgemein in Unity *Awake()* vor *Start()* ausgeführt wird. Um dieses Problem also nun zu lösen, werden die Einstellung an den „Script Execution Settings“ in Unity benötigt, so wie sie in Abbildung 4.5 zu sehen ist.

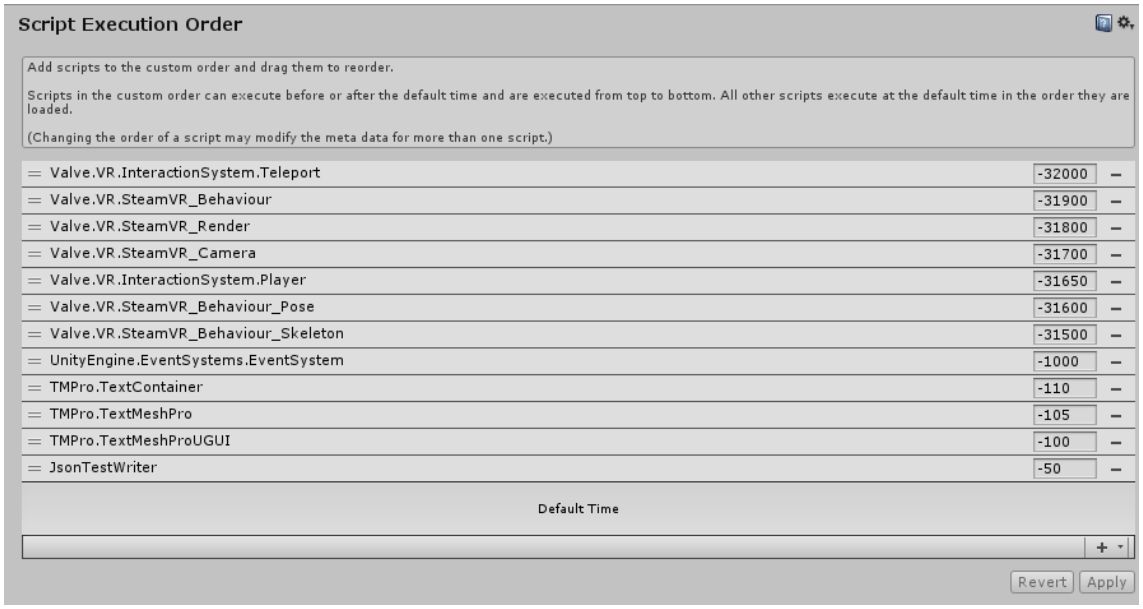

Abbildung 4.5: Die Einstellung der „Script Execution Settings“ im den „Project Settings“ in Unity.

Durch die „Script Execution Settings“ kann das Schreiben der Dateien, und das Einlesen derer beides in die *Start()*-Funktion geschrieben werden, wobei nun nur noch die Ausführungsreihenfolge vom *JsonTestWriter* so geändert werden muss, dass diese vor *JsonManager* ausgeführt wird. Um dies zu implementieren, muss nun also das Singleton-Objekt in der *Awake()*-Funktion in Unity instanziiert werden, und danach in der *Start()*-Funktion vom *JsonTestWriter* auf das Objekt zugegriffen werden, um dann den Pfad zu erhalten.

Um nun das Testmodell schreiben zu können wurde die Funktion *WriteTestNodes()* geschrieben. In dieser Funktion wurde ein Beispielbaum statisch im Code formuliert. Dieser Beispielbaum diente später zum Testen der Splitkriterien und den generellen Durchfluss der Daten. Die Funktion *WriteTestNodes()* wurde aber auch überladen,

um in den Parametern eingeben zu können, wie der gewünschte Baum aussehen soll. Die überladene Funktion kann in Listing 4.4 begutachtet werden.

Listing 4.4: Überladene Funktion zum Schreiben beliebig aussehender Bäume.

```
1  void WriteTestNodes(int layerCount, Node node,
2                      int minChildCount, int maxChildCount)
3  {
4      layerCount--;
5
6      int currentChildCount = Random.Range(minChildCount,
7                                           maxChildCount);
8      node.children = new Edge[currentChildCount];
9
10     if (layerCount != 0)
11     {
12         for(int i = 0; i < currentChildCount; i++)
13         {
14             node.children[i] = new Edge();
15             node.children[i].targetNode = new Node();
16             WriteTestNodes(layerCount,
17                            node.children[i].targetNode,
18                            minChildCount, maxChildCount);
19         }
20     }
21 }
```

Der Parameter *layerCount* beschreibt, wie tief der Baum geht, wobei die Parameter *minChildCount* und *maxChildCount* angeben wieviele Kinder ein Knoten mindestens, und maximal haben kann. Diese Funktion ist rekursiv, und erstellt solange Kindknoten und Kanten, wie es der Parameter *layerCount* erlaubt. Bei genauerer Inspektion des Funktionscodes fällt auf, dass für die Instanziierung der Knotenobjekte garkeine *ISplitRule* angegeben wird. Da keine Parameter im Knotenkonstruktor angegeben werden, wird hier der Standartkonstruktor aufgerufen, welcher den *ISplitRule* „*UndefinedSplitRule*“ vergibt. *UndefinedSplitRule* hat keine richtige implementierte *Execute()*-Funktion, weshalb die Funktionalität der Methode vorallem nur beim Testen der Platzierung der Knoten Verwendung fand.

Die Funktion *WriteTestData()* dient dazu, beliebig viele Anzahl von Werten für bestimmte Attribute zu schreiben. Diese Attribute sind statisch in der Funktion definiert, und stimmen mit den statisch definierten Attributen der *WriteTestNodes()* Funktion überein.

Um diese Testdaten schreiben zu können, muss das *JsonTestWriter*-Prefab in der Szene existieren. In Abbildung 4.6 sind die möglichen Einstellung vom Inspektor zu betrachten. Hier kann man durch die booleschen öffentlichen Attribute *WriteModel* und *WriteData* entscheiden, ob nun neue randomisierte Daten, oder Modelle beim Start der Anwendung erstellt werden sollen.

Abbildung 4.6: Ansicht der *JsonTestWriter*-Komponente des *JsonTestWriter*-Prefabs. Hier kann das Schreiben eines Modells, oder der Datenmatrix eingestellt werden.

4.4 Importierung der externen Daten

In dieser Sektion der Arbeit wird die Implementation der Funktionspackete „Import des Modells“ und „Import der Datenmatrix“ behandelt. Sie sind den funktionellen Anforderungen F1, JSON Integration, und F2, CSV Integration, einzugliedern. Es wird vor allem auf die Serialisierung, und auf die Frameworks eingegangen, mit denen bei der Importierung gearbeitet wurde. Die genannten Aufgaben der Funktionspackete werden von der Klasse *JsonManager* übernommen. Der für diesen Teil der Arbeit relevanten Ausschnitt des UML-Diagramms kann noch einmal in Abbildung 4.7 betrachtet werden.

Sie dient als global zugängliche Managerklasse, um einfach auf den Baum, den Daten und dessen Standorten zugreifen zu können. Die Funktion *ReadTree()* dient zum einlesen einer JSON, welche sich im angegeben Ort von *treePath* befindet. Diese Funktion nutzt das Framework „Json.Net“ um die JSON-Datei einzulesen, und aus dem Text die einzelnen Objekte zu deserialisieren. Die Deserialisierung findet durch die

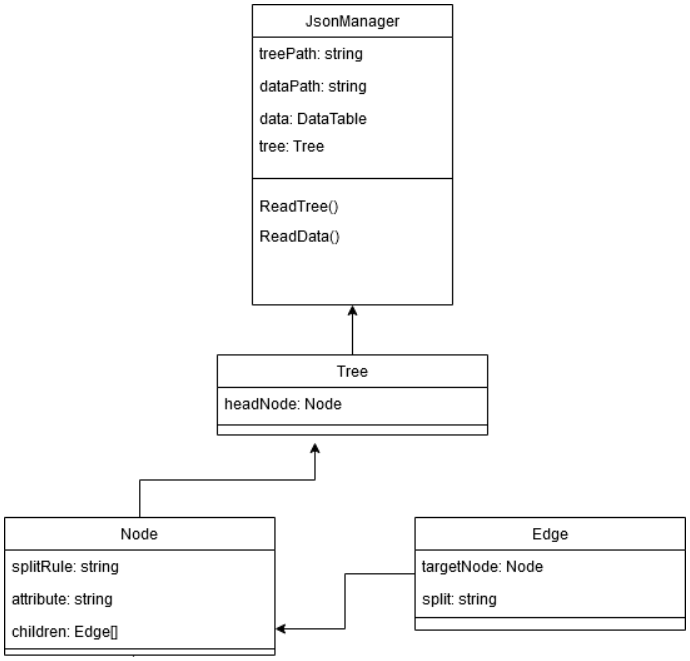

Abbildung 4.7: Ausschnitt des UML-Diagramms mit Fokus auf den *JsonManager* und den für den Baum relevanten Klassen.

Methode *JsonConvert.DeserializeObject<T>(String)* statt. Der *String*-Parameter steht hier für den Pfad, und das *T* steht für *Type*. Hier muss der Objekttyp genannt werden, zu dem das Objekt in der JSON-Datei deserialisiert werden soll, was im Fall dieser Software der *Tree*-Typ ist.

Die *ReadData()*-Funktion des *JsonManagers* übernimmt das laden der Datenmatrix aus der CSV-Datei. Dazu wurde das Framework „CsvHelper“ verwendet. Es bietet auch automatisch die Möglichkeit die eingelesene CSV-Datei, in ein *DataTable*-Objekt zu verwandeln. Eine *DataTable* ist eine Tabellenklasse, mit der auf Zeilen und Spalten sehr einfach zugegriffen werden kann. Sie implementiert Methoden wie *GetRow()* oder *NewRow()*, um die Tabelle einfach auslesen oder ändern zu können.

Das *JsonManager*-Prefab, ermöglicht die Eingabe von Parametern über den Unity-Inspektor, wie in Abbildung 4.8 zu sehen ist. *Tree Name* bildet das Eingabefeld für den Namen der Datei, welcher eingelesen werden soll, und *Data Name* den Namen für die Datenmatrix. Im *Tree*-Feld, kann der eingelesene Baum beobachtet werden. Der Baum wird aber nur beim Start der Anwendung eingelesen, daher ist der Baum

im Inspektor auch nur beim Start gefüllt. Hier kann für Debugging-Zwecke noch einmal überprüft werden, ob der Baum auch korrekt visualisiert wurde.

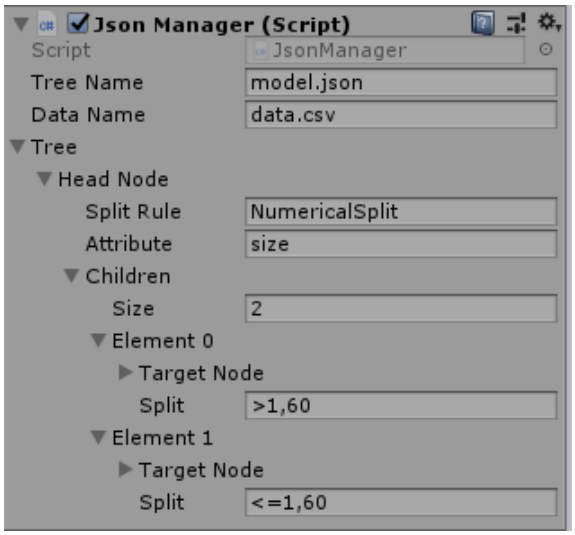

Abbildung 4.8: Ansicht der *JsonManager*-Komponente des *JsonManager*-Prefabs. Es muss hier der Name der Dateien für den Baum und die Datenmatrix eingegeben werden. Auch kann in dieser Komponente der eingelesene Baum betrachtet werden.

4.5 Initialisierung der Knoten und Kanten

Für die Visualisierung des Baumes in Unity, wurden die Initialisierung des Baumes in Unity in zwei verschiedene Bereiche geteilt. Zuerst wird jeder Knoten zusammen mit seinen benötigten Komponenten und Kanten initialisiert, und dann werden die Knoten in eine bestimme Position platziert. Das Initialisieren der Knoten und Kanten gehört der funktionalen Anforderung F3, Initialisierung des Baumes, an und wird von der Klasse *VisualizationManager* gehandhabt. Da die Initialisierung und die Platzierung beides vom *VisualizationManager* übernommen wird, wird das Prefab erst am Ende der Sektion 4.6 auf Seite 32 vorgestellt. Der für diesen Teil der Arbeit relevante Ausschnitt des UML-Diagramms ist in Abbildung 4.9 zu begutachten.

Um den Baum von der JSON-Datei, in den sichtbaren 3D-Raum von Unity zu bringen, müssen einige Dinge beachtet werden. Die Klassenstruktur, die im Kapitel 4.2 auf Seite 18 beschrieben wurde, kann in Unity so nicht direkt verwendet

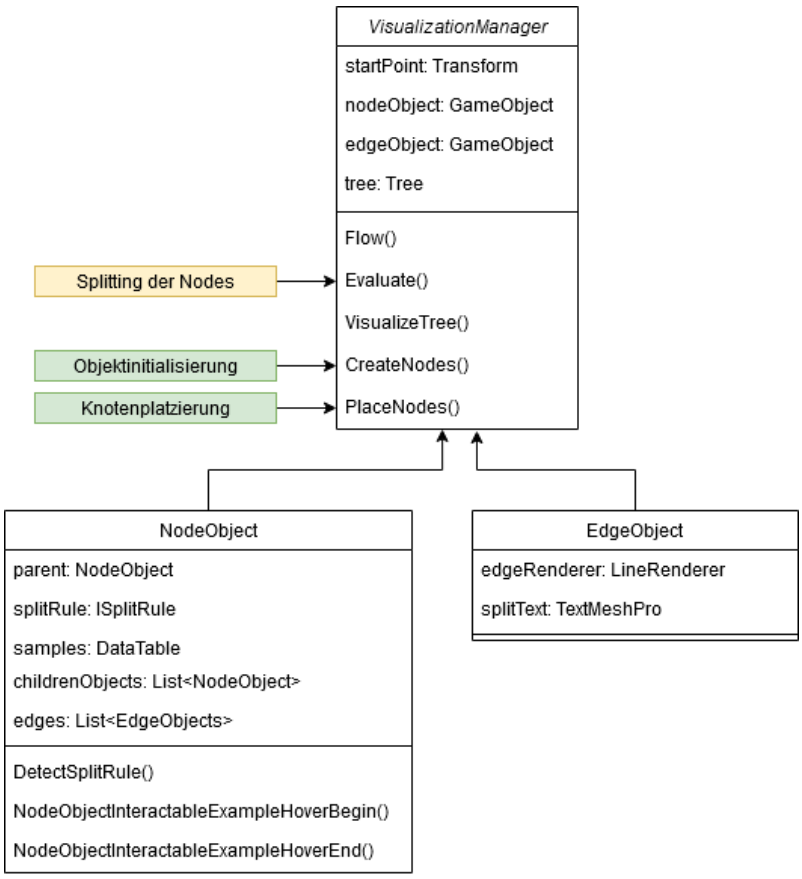

Abbildung 4.9: Ausschnitt des UML-Diagramms mit Fokus auf den *Visualization-Manager*. Die Klassen *NodeObject* und *EdgeObject* stellen jeweils den Knoten und den Kanten in Unity dar.

werden. Klassen können nur in Form von Komponenten in Unity existieren, und um eine Klasse in Unity als Komponente zu verwenden, muss diese von der Klasse *Monobehavior* erben. [Tec19] Also wurde für die Knoten- und Kantenklasse jeweils eine zusätzliche Wrapperklasse geschrieben, *NodeObject* und *EdgeObject*. *NodeObject* speichert das jeweilige normale *Node*-Objekt, die Daten die durch sie durch fließen als ein *DataTable*-Objekt, den Elternknoten und mehr in sich als Attribute. Somit sind alle relevanten Daten des Knotens nun im Knotenobjekt in Unity gespeichert. Um den Knoten dann in Unity zu initialisieren, wurde daraus ein Prefab gemacht. Das fertige Prefab kann in Abbildung 4.10 betrachtet werden.

Diese Prefabs werden dann in der Funktion *CreateNode()* mit den genannten Daten initialisiert. Zusätzlich findet hier auch die Zuteilung der Splitrule in der Funktion

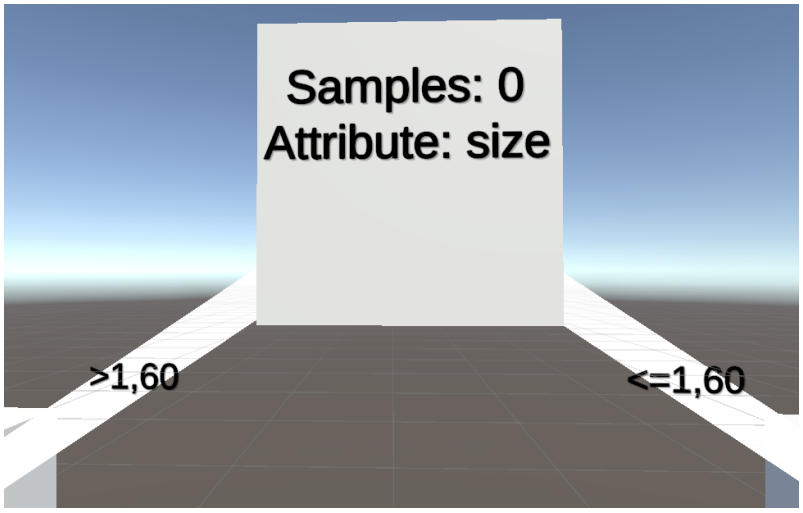

Abbildung 4.10: Das *NodeObject*-Prefab in Unity als Würfel. Die weißen Linien bilden die *EdgeObject*-Prefabs ab, welches die Kanten sind.

DetectSplitRule() statt. Wie in Kapitel 4.2 auf Seite 18 beschrieben, ist in der Klasse *Node* nur eine Zeichenkette für die Splitrule angegeben. Diese muss noch in ein Objekt der jeweiligen Klasse umgewandelt werden, um dessen *Execute()*-Funktion zu verwenden. Diese Umwandlung findet in dieser Funktion statt. Sie wurde als einfacher Zeichenkettenvergleich implementiert. *CreateNode()* ist eine rekursive Funktion, welche demnach also für alle Kinder erneut aufgerufen wird, sodass alle Knoten und Kanten nun in der Welt existieren.

4.6 Platzierung der Knoten und Kanten

In dieser Sektion wird der Algorithmus der Platzierung der Knoten erläutert, und nach welchen Regeln diese platziert wurden. Dies ist der funktionellen Anforderung F5, Einheitlichkeit des Baumes unterzuordenen. Das UML-Diagramm in Abbildung 4.9 in Sektion 4.5 auf Seite 27 kann auch für diese Sektion verwendet werden.

Nachdem alle Knoten und Kanten nun in Unity initialisiert wurden, müssen diese nun so platziert werden, dass der Baum visuell ansehnlich aussieht. John Q. Walker hatte dazu einige Regeln aufgestellt, welche bei der Implementation beachtet wurden [Wal90]. Zusammenfassend ist aus seiner Interpretation eines visuell ansprechenden Baumes also zu erwähnen, dass die Knoten derselben Tiefe auch die

selbe y-Koordinate im 3D-Raum haben müssen, der Elternknoten immer in der Mitte vom ersten und letzten Kindknoten sein muss, und der Baum immer vom Kopfknoten gespiegelt werden können sollte. Diese Vorgaben kommen mit einigen Hindernissen einher. Dadurch, dass die Anzahl der Kinder variabel ist, kann bei der Positionierung der Knoten also nur schwer vom Kopfknoten angefangen werden, weil man nur mühsam vorher einschätzen kann wieviel Platz der Knoten brauch. Die erste Implementation der Platzierung im Programm hat aber genau dies versucht. Es wurde versucht eine Einschätzung des benötigten Platzes zu generieren, indem man alle Gesamtkinder eines Knotens zusammenzählt, und in jedem Knoten speichert. Theoretisch hätte man damit also die Maximaldistanz die dieser Knoten bräuchte, und kann diese dort positionieren. Bei kleineren Bäumen sah es auch sehr gut aus, doch als dann Testbäume implementiert wurden die viel mehr Kinder hatten, wurde sehr schnell sichtbar, dass das nicht die optimale Implementation war. In Abbildung 4.11 ist zu sehen, dass bei dieser Implementierung der Zwischenplatz zwischen linken und rechten Knoten größer ist, als die eigentlich Distanz der Knoten untereinander.

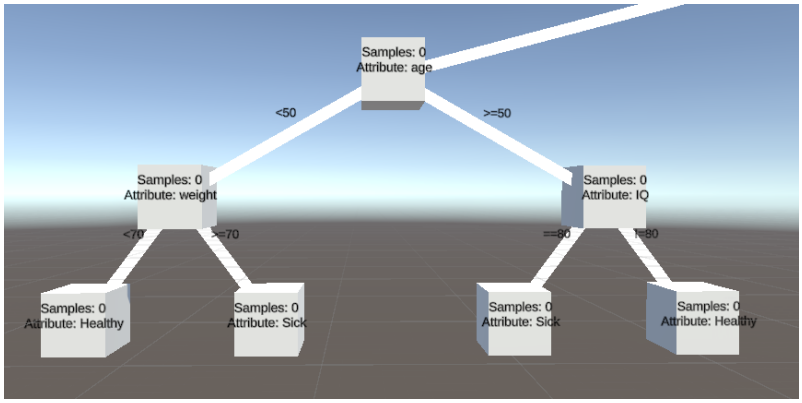

Abbildung 4.11: Implementation eines frühen Knotenplatzierungsalgorithmus. Der Abstand der Knoten ist sichtbar nicht gleichmäßig.

Die jetzige Positionierung der Knoten wird durch die Funktion *PlaceNodes()* implementiert. Die Idee dieser Implementation ist, dass schlussendlich die Distanz die ein Knoten als Abstand auf der x-Achse brauch davon abhängt, wie viele Blattknoten im untersten Layer existieren. Wenn also demnach, zuerst alle Blattknoten in der richtigen Reihenfolge positioniert werden, und danach von den Blattknoten ausgehend jeweils die Elternknoten mittig positioniert werden, bis der Wurzelknoten erreicht ist, müsste jeder Knoten gleichmäßig positioniert sein. Um dies zu implementieren, müssen also im ersten Schritt alle Blattknoten in einer bestimmten Reihenfolge in

einer Liste gespeichert werden. Es wird also zuerst der Baum mit der Funktion *SearchForBottomLeftLeaf()* zum untersten linken Knoten traversiert. Anschließend wird, von diesem Blattknoten ausgehend, versucht den jeweils nächste Blattknoten zu finden. Dies ist in der Funktion *FindLeafNodes()* implementiert, welche im Listing 4.5 zu betrachten ist.

Listing 4.5: Funktion, welche ausgehend vom untersten linken Blattknoten, alle anderen Blattknoten in einer Liste in der richtigen Reihenfolge speichert.

```
1  void FindLeafNodes(NodeObject node)
2  {
3      for (int i = 0; i < node.childrenObjects.Count; i++)
4      {
5          if(node.childrenObjects[i].childrenObjects.Count == 0)
6          {
7              leafs.Add(node.childrenObjects[i]);
8          } else if(!node.childrenObjects[i].traversed)
9          {
10             FindLeafNodes(node.childrenObjects[i]);
11         }
12     }
13     if (node.parent)
14     {
15         node.traversed = true;
16         FindLeafNodes(node.parent);
17     }
18 }
```

Diese Funktion überprüft zuerst, ausgehend von einem Knoten, ob dessen Kinder Blattknoten sind. Wenn dem so ist, dann werden sie der Blattknotenliste hinzugefügt. Wenn aber die Kinder keine Blattknoten sind, so wird die Funktion jeweils wiederholt auf dessen Kinder angewandt, bis Blattknoten zum Vorschein kommen. Ein wichtiges Attribut für die Traversierung in der Funktion ist *traversed*, ein boolesches Attribut, das widergibt ob dieser Knoten schonmal überprüft wurde. Ohne *traversed*, würde die Funktion offensichtlich eine Endschlosschleife darstellen. Wenn alle Kinder überprüft wurden, wird die Funktion auf das Elternknoten angewandt, und der jetzige Knoten auf *traversed = true* gesetzt.

Nach dieser Vorarbeit existiert nun eine Liste mit allen Blattknoten, welche in ihrer richtigen Reihenfolge von links nach rechts gespeichert wurden. Es kann nun einfach durch die Liste iteriert werden, und die Blattknoten Stück für Stück nacheinander mit der Distanz *nodeDistance* platziert werden. Zu vermerken ist hier jedoch, dass nicht jeder Blattknoten auf demselben Layer des Baumes liegt. Um dies zu lösen, wird bei der Initialisierung der Knoten, der richtige Layer in *NodeObject* gespeichert. Bei der Platzierung in der y-Achse kann hier die Layerstufe einfach verwendet werden. Nun müssen nur noch die Elternknoten der Blattknoten in der Mitte vom ersten und letzten Kind positioniert werden, und dieser Vorgang wird rekursiv für jeden Knoten ausgeführt. Mit dieser Implementation kann man die Knoten sehr einfach mit dem kleinsten Abstand positionieren, und trotzdem die Regeln von John Q. Walker befolgen. Der Unterschied zur vorherigen Implementation kann in Abbildung 4.12 betrachtet werden.

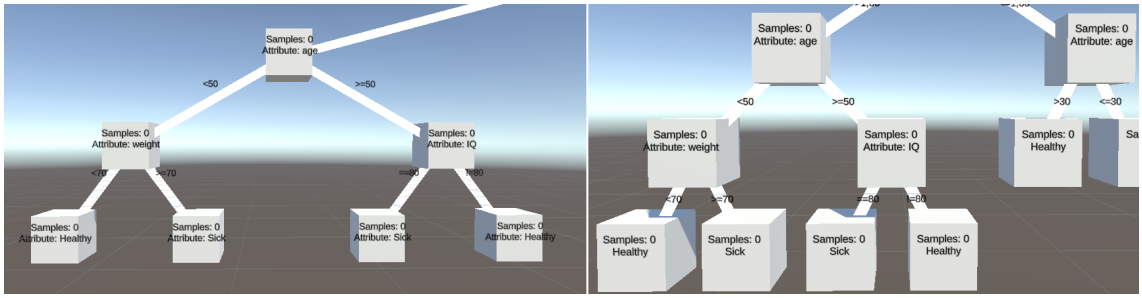

Abbildung 4.12: Vergleich der verschiedenen Implementationen des Knotenplatzierungsalgorithmus. Links im Bild ist die alte Implementierung, und rechts die neue Implementierung zu sehen. Sichtbar ist die Platzverschwendung in der alten Implementierung.

Das *VisualizationManager*-Prefab übergibt dem Nutzer im Inspektor vielerlei Möglichkeiten um Einstellung für die Visualisierung vorzunehmen. In Abbildung 4.13 kann die Komponente mit ihren Einstellungsmöglichkeiten betrachtet werden. Durch *StartPoint* kann die *Transform*-Komponente eines *GameObjects* genommen werden, um den Ort zu definieren, wo der Wurzelknoten instanziiert werden soll. Um sich dieser Einstellung vorzunehmen, reicht es hier ein *Empty GameObject* zu kreieren, zum Wunschort zu positionieren, und in das Feld von *Startpoint* reinzuziehen. Die Felder von *NodeObject* und *EdgeObject* stellen die Objekte dar, die für Instanziierung von Knoten und Kanten verwendet werden. *EdgeLength* ist die Kantenlänge, und *NodeDistance* ist die horizontale Distanz zwischen den Knoten. Zum Schluss, um den Datenfluss einzuleiten dient die Box *FlowData*. Es ist intern ein öffentlicher

boolescher Wert, welcher, wenn *true* geschalten wird, den Datenfluss einleitet. Da er öffentlich ist, und sich in einer Managerklasse befindet, kann hier also sehr einfach durch eine andere Klasse der Datenfluss eingeleitet werden.

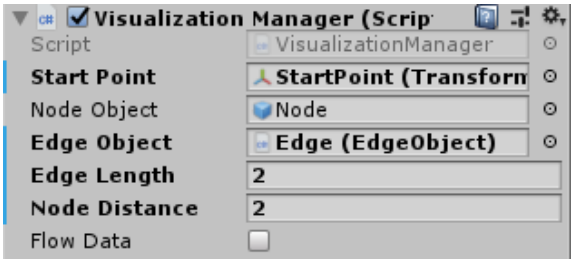

Abbildung 4.13: Ansicht der *VisualizationManager*-Komponente des *VisualizationManager*-Prefabs. Es muss hier der Name der Dateien für den Baum und die Datenmatrix eingegeben werden. Auch kann in dieser Komponente der eingelesene Baum betrachtet werden.

4.7 Durchfluss und Auswertung der Datensätze

In diesem Abschnitt der Arbeit wird die Implementation der Funktion des Programmes behandelt, welche die Daten nacheinander aus der Datenmatrix ausliest, und dann bei jedem Datensatz entscheidet, ob es in den Kindknoten weitergeleitet werden darf oder nicht. Die Entscheidung, ob der Datensatz in den Kindknoten weitergeleitet werden darf, wird durch die schon in Kapitel 4.2 angesprochene *Execute()*-Funktion entschieden, welche, je nach angegebener Splitrule, einen anderen Algorithmus anwendet. Diese Sektion implementiert die funktionale Anforderung F6, Auswertbarkeit der Daten durch das Modell.

Um zu entscheiden, ob ein Datensatz überhaupt in den nächsten Knoten weitergeleitet werden darf oder nicht, brauch es dafür erst einmal eine Funktion, welche diesen Datenfluss einleitet. Dafür ist die *Evaluate()*-Funktion des *VisualizationManager* da. Sie fängt beim Kopfknoten an, macht eine Kopie aller Datensätze, und speichert diese in der *NodeObject*-Komponente. Diese Kopie der Datenmatrix wird später genutzt, um zu visualisieren welche Daten überhaupt durch den Knoten gelaufen sind. Anschließend wird durch jeden Datensatz in der Matrix iteriert, und die *Execute()*-Funktion des Knotens an diesen Datensatz angewandt. Wenn durch die *Execute()*-Funktion evaluiert wurde, dass dieser Datensatz nicht in den nächste

Knoten weitergeleitet werden darf, wird dieser Datensatz aus der lokalen Kopie des Knotens aus der Matrix gelöscht. Diese Funktion wird für jeden Kindknoten angewandt.

Die *Execute()*-Funktion, ist eine Methode welche den booleschen Wert *true* zurückgibt, wenn das Splitkriterium erfüllt wurde. Beim *NumericalSplit*, wird davon ausgegangen, dass das Attribut und der Splitstring beides Zahlenwerte sind. Durch reguläre Ausdrücke wird gefiltert, welcher boolescher Operator sich im Splitstring befindet, und dann wird diese boolesche Operation zwischen diesen beiden Zahlenwerten ausgeführt. Die andere Splitrule, der*CategoricalSplit*, ist aus Zeitgründen noch nicht implementiert, würde aber ähnlich implementiert werden. Für Debuggingzwecke, und für den Fall, dass der Split in der JSON nicht definierbar ist, gibt es den *UndefinedSplit*. Wenn die *Execute()*-Funktion nun erfolgreich bestimmt hat, dass dieser Split *true* ist, wird der Datensatz an den jetzigen behandelten Kindknoten übergeben. Der ganze Datenfluss wird derzeit durch eine boolesche Variable im *VisualizationManager* eingeleitet, sodass nicht automatisch beim Programmstart alle Daten evaluiert werden. Diese Variable ist *public*, und somit auch im Unity-Inspektor zugreiflich.

4.8 Interaktion im virtuellen Raum

Für diese Arbeit galt es, grundlegende Einstellungen und Skripte so vorzubereiten, sodass Interaktionen später einfach hinzugefügt werden können. In diesem Kapitel werden die Hilfsmittel beschrieben, die benötigt werden, um Interaktionen in VR in Unity möglich zu machen. VR ist ein aufblühendes Geschäft, weshalb es deshalb auch sehr viele verschiedene HMDs und Controller gibt, welche auch komplett verschieden fungieren. So haben beispielsweise die Controller der Oculus Rift ganz andere Buttons als die von der HTC Vive. Dies führt zu einigen Problemen bei der Arbeit mit VR. Selbst die Features können voneinander unterscheiden, so bieten Controller der Valve Index beispielsweise die Möglichkeit, eine Ästimation der Position der Finger auszugeben. Damit man nicht für jeden Controller ein eigenes Skript schreiben muss, muss hier die Eingabe also abstrahiert werden. Das heißt, dass man also nur ein *Event* definiert, beispielsweise ein „Benutzen“ *Event*, welches eine Tür öffnet wenn ein Knopf des Controllers gedrückt wird. Anschließend definiert man nur noch welcher Knopf nun das „Benutzen“ *Event* auslöst. Um diese Abstraktion des Inputs zu

implementieren, wurde das Unitypackage SteamVR von Valve benutzt. Mit Hilfe von SteamVR, kann man für solch eine Abstraktion sogenannte *Actionsets* verwenden, welche eine Reihe von Events beschreibt. Diese können auch in Echtzeit geändert werden, was nützlich ist um beispielsweise die Steuerung zu ändern wenn ein neuer Gegenstand in einem Spiel aufgehoben wird. Wie solch ein *Actionset* aussieht, kann in Abbildung 4.14 beobachtet werden.

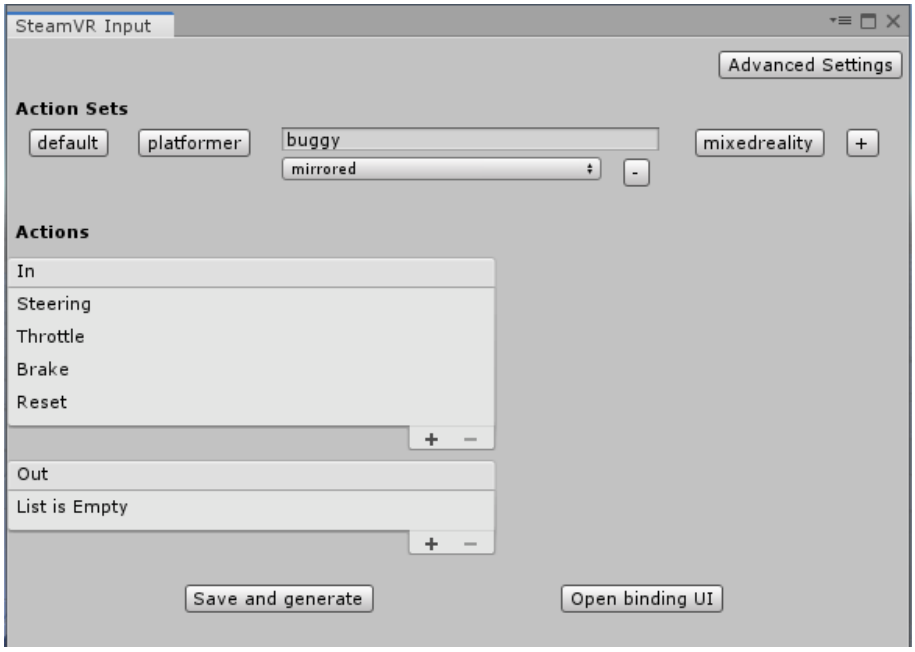

Abbildung 4.14: Actionsets in SteamVR

In der Abbildung wurde ein *buggy* Actionset definiert, mit dem der Spieler einen Controller bedienen kann, welches ein Spielzeugauto fährt. Es beinhaltet Actions wie lenken, fahren und bremsen. Sie können verschiedene Werte ausgeben, wie zum Beispiel einen booleschen Wert, für einen Knopf am Controller, oder einen 2D Vektor, für beispielsweise ein Touchpad. Diese Actionsets werden als eine Json-Datei gespeichert. Mit Hilfe des in der selben Abbildung aufgezeigten Buttons *Open Binding UI*, kann ein Menü geöffnet werden, in dem diese Actions dann den Buttons der Controller zugewiesen werden kann. Dies muss für alle relevanten Controllertypen getan werden.

SteamVR erlaubt es auch sehr einfach Interaktionen des Spielers mit Objekten in der Welt zu implementieren. Die wichtigsten Komponenten die von SteamVR zur Verfü-

gung gestellt werden, sind die Player-Klasse, die Hand-Klasse und die Interactable-Klasse.

Die *Player*-Klasse ist eine Singleton-Klasse, welche global erreichbar ist um verschiedene vitale Informationen für sämtliche Komponenten in der Szene bereit zu stellen. So wären beispielsweise die Hände, und das HMD des Spielers hier zu finden. Durch diese Klasse wird auch der 2D-Modus implementiert. Es wird zeitgleich beim Start des Programmes geschaut, ob ein verfügbares HMD angeschlossen ist, oder nicht, und entsprechend der benötigte Modus aktiviert. Diese Klasse ist schon fertig eingestellt im Player Prefab des SteamVR Packetes anzufinden, welches auch für die Implementation des Programmes verwendet wurde. Diese Klasse stellt auch eine Estimation der Fußposition zur Verfügung, durch die Variable *feetPositionGuess*. Dies ist darauf basierend, wo sich das HMD befindet, kann aber sehr inakkurat sein, da es keine sichere Möglichkeit gibt dies einzuschätzen. Darauf aufbauend kann auch die Bewegungsrichtung des Spielers aus dieser Klasse mit *bodyDirectionGuess* ausgelesen werden.

Die *Hand*-Klasse ist die komplizierteste Klasse vom SteamVR Packet. Sie überprüft jederzeit, ob eine Hand des Spielers über ein Objekt in der Szene schwebt, und sendet diesem Nachrichten zu. Diese Nachrichten lösen je nach Status spezifische Events aus. Sie regelt auch das Anfügen von Objekten an den Händen des Spielers, was gebraucht wird, wenn der Spieler beispielsweise einen Gegenstand aufheben möchte. Die *Hand*-Klasse ist im Player-Prefab schon enthalten, und es musste bei der Implementierung auch nichts weiter daran eingestellt werden. Wobei die *Hand*-Klasse diese Nachrichten aussendet, ist die *Interactable*-Klasse die Klasse, welche die Nachrichten erhält. Sie muss an alle Objekte als Komponente angefügt werden, welche diese Nachrichten erhalten sollen, weshalb sie auch an das Node-Prefab angehangen werden musste. Dadurch konnte prototypisch eine eigene Interaktion implementiert werden. Mit Hilfe der Komponente *InteractableHoverEvents* aus dem SteamVR Packet, wurden bei spezifischen Ereignissen eigene Events in Form von *UnityEvents* ausgelöst. Dies wird in Abbildung 4.15 veranschaulicht.

Zu sehen ist, dass jeweils bei den Ereignissen *On Hand Hover Begin* und *On Hand Hover End* eine Funktion der Knotenlasse aufgerufen wird, welche beim Berühren des Knotens mit der Hand den Knoten rot färbt, und wieder weiß, wenn man sie nichtmehr berührt.

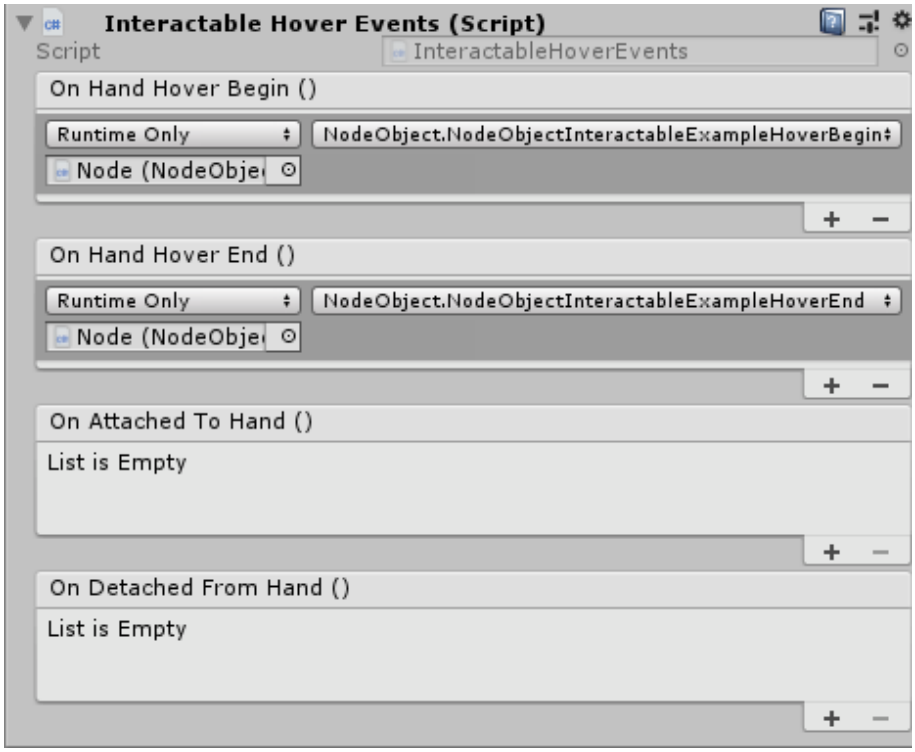

Abbildung 4.15: *InteractableHoverEvents*-Komponente des Node Prefabs

5 Nutzerleitfaden

Um das Projekt testen zu können, muss die Unity-Version 2019.2.1f1 installiert sein. Abweichungen von der Unity-Version können zu vorher nicht dagewesenen Fehlern führen. Optional, um das Projekt in VR zu starten, muss die Software *SteamVR*[1] installiert sein, und es muss eine VR-Brille zur Verfügung stehen. Das Projekt kann auch in einem 2D Modus gestartet werden, bei dem die Maus als Ersatz für die Controller bedient werden kann, und die Kamera mit den Tasten *WASD* bewegt werden kann. Es kann entweder das ganze Projekt über Github[2], oder das Unitypackage über den Downloadlink im *Readme* des Githubs runtergeladen und gestartet werden.

Um das Unitypackage in ein Unityprojekt zu integrieren, muss zuerst das Zielprojekt geöffnet, oder falls noch keins vorhanden ist, ein neues Projekt erstellt werden. Um ein neues Projekt zu erstellen, muss einmal zu *Projects* navigiert werden, und auf *New* geklickt werden, um das Fenster in Abbildung 5.1 zu öffnen.

Hier muss das 3D *Template* ausgewählt werden, ein Projektname ausgesucht, und ein Speicherort definiert werden. Wenn das Projekt erstellt wurden, gilt es nun das Unitypackage zu importieren. Hierfür muss, wie in Abbildung 5.2 abgebildet ein *Custom Package* importiert werden. Dies ist unter *Assets* > *Import Package* > *Custom Package* zu finden. Als nächster Schritt muss nur noch der Import bestätigt werden.

Nach dem erfolgreichen Import ist nun im *Assetfolder* ein Ordner *DT Visualizer* aufgetaucht. In diesem Ordner befinden sich alle *Assets* und Daten vom *Package*. Falls nach dem abgeschlossenen Import eine Fehlermeldung *Missing compiler required member 'Microsoft.CSharp.RuntimeBinder.CSharpArgumentInfo.Create'* auftaucht, muss hier unbedingt die *API Compatibility* des Projektes auf *.NET 4.x* geändert werden. Diese Einstellung ist unter *Edit* > *Project Settings* > *Player* > *Configuration*

[1] Siehe https://store.steampowered.com/app/250820/SteamVR/
[2] Siehe https://github.com/Geeses/Visualization-Decision-Trees

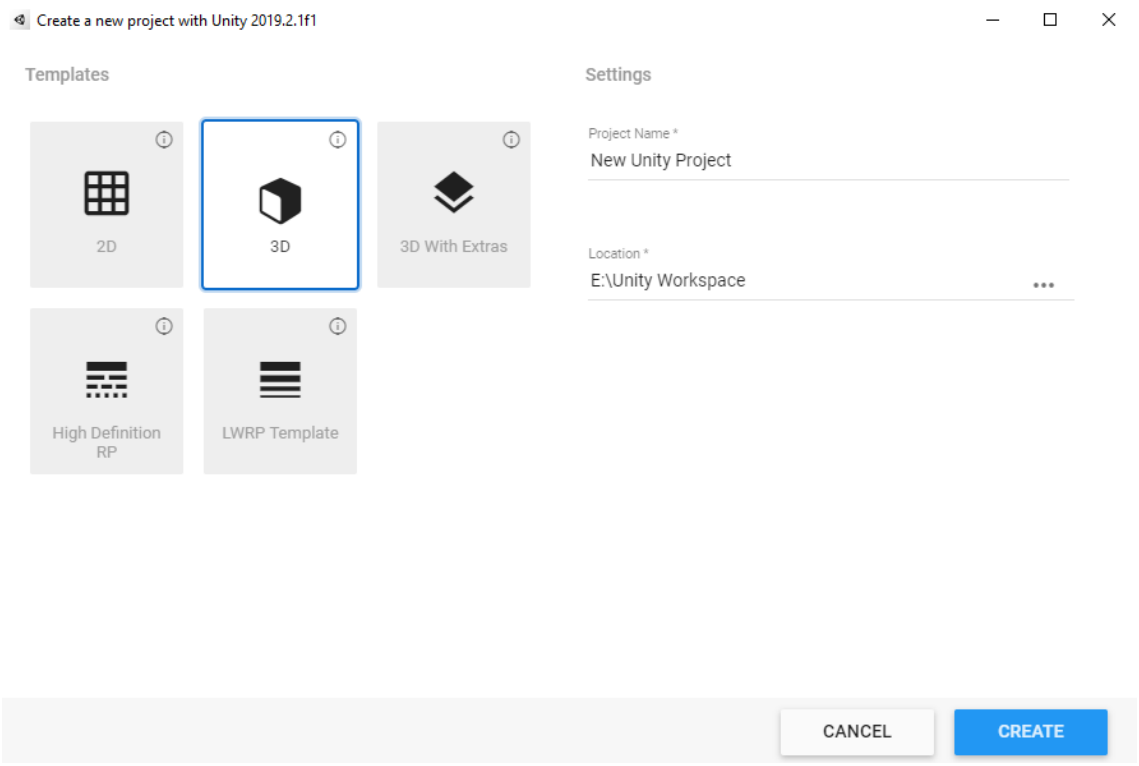

Abbildung 5.1: Projekterstellungsfenster

zu finden. Dies liegt daran, dass bei der Implementation einige *Features* der *.NET 4.x API* verwendet wurden, jedoch standartgemäß im Projekt nur *.NET Standart 2.0* ausgewählt wird.

Um nun die Funktionalitäten in eine eigene Szene zu integrieren, brauch es die Prefabs *Player, JsonManager, JsonTestWriter* und *VisualizationManager*, welche im Kapitel Implementation in ihren Abschnitten erläutert wurden. Sie befinden sich im *Prefab* Ordner und müssen nun einfach in die Szene gezogen werden um sie dort zu platzieren. Anschließend können Änderungen in den Einstellungen der Prefabs vorgenommen werden, falls dies so gewollt wird.

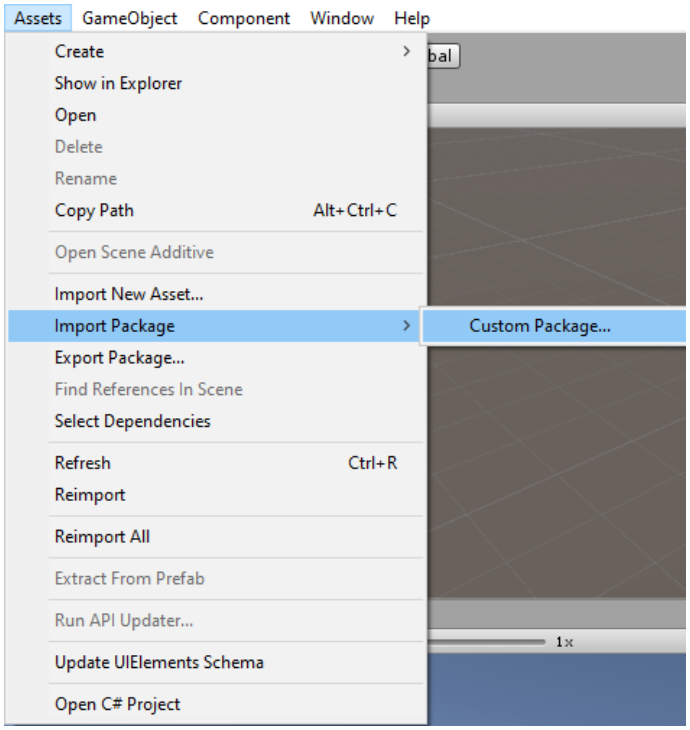

Abbildung 5.2: Fenster zur Packetimportierung

6 Evaluation

Dieses Kapitel gibt Auskunft über den Versuchsablauf der Evaluation, und deren Ergebnisse. Die Evaluation diente zum Testen mehrere Aspekte. Zum einen sollte die technische Sicht getestet werden, also wie die Tester mit der VR Technik zurechtkamen. Dann ob die jetzige Steuerung verständlich ist. Und zu danach, ob nach der Benutzung der Software verständlicher ist, wie solch ein Entscheidungsbaum nun durchläuft, und wie dieser aussieht. Das Erreichen von Testern hat sich als eine große Schwierigkeit ergeben, da wenig Leute eine VR-Anlage bei sich zu Hause besitzen. Daher konnten keine Tests Online durchgeführt werden, was dazu führte, dass alle Tests mit eigener Hardware durchgeführt wurden. Bei der Auswahl der Testpersonen war es entscheidend Menschen unterschiedlicher Natur zu nehmen. Insgesamt wurden 6 verschiedene Personen befragt, einige mit einem groben Wissen vom maschinellem Lernen, und einige ohne.

6.1 Versuchsbeschreibung

Vor dem direkten Testen wurden allgemeine Daten zur Gruppierung der Teilnehmer entgegengenommen, wie Alter, Geschlecht, Hintergrund zu VR, und Wissensstand zum maschinellen Lernen. Daraufhin gab es eine kurze Einführung, wie mit der VR-Hardware umzugehen ist, und wie diese aufgesetzt wird. Es wurde für den Versuch die Valve Index mit den Index-Controllern in einem rund 1m*1m großem Feld verwendet. Es wurde auch die Steuerung der Controller vorher vorgeführt. Anschließend kam es zum Durchlauf der Applikation.

Die Teilnehmer mussten nun die VR Geräte aufsetzen. Wenn sie bereit waren, wurde eine Frage zur Verständlichkeit der UI gestellt, ohne ihnen vorher erklärt zu haben was Entscheidungsbäume sind. Anschließend wurde ihnen nun grob erläutert, was maschinelles Lernen ist, und in welchem Zusammenhang der Entscheidungsbaum dazu steht.

Anschließend durften die Teilnehmer den Knopf berühren, mit dem sie die Daten durch die Knoten laufen lassen konnten. An diesem Punkt wurde den Probanden nun die nächsten Fragen zum User-Interface gestellt. Nach Beendigung des Versuches, gab es noch ein paar zusammenfassende Fragen.

6.2 Auswertung

Die Überzahl der Befragten befand sich im älteren Spektrum. Insgesamt gab es 4 über 40-Jährige, und 2 Leute welche jeweils 22 Jahre alt waren. Da in dieser Evaluation vor allem auch die Verständlichkeit getestet werden sollte, ist es hier nur von Vorteil mit so vielen älteren Leuten zu testen.

Bevor das maschinelle Lernen und Entscheidungsbäume noch einmal erläutert wurden, konnten 3 der Befragten das User-Interface der Knoten erläutern, wobei 2 von den Personen vorheriges grobes Wissen zu Entscheidungsbäumen und maschinelles Lernen besaßen. Dies sagt also aus, dass wenn generelles Wissen über die Thematik bestand, das User-Interface so in der jetzigen Form hinreichend war. Im Gegenzug dazu konnte jedoch nur eine Person der restlichen 4 das Interface deuten, ohne vorheriges Wissen zu haben. Wenn diese Software also zur Visualisierung für Experten benutzt wird, sollte es in der Form hinreichend sein. Diese Erkenntnis wird dadurch auch wiederholt bestärkt, dass nach der Aufklärung zum maschinellen Lernen und Entscheidungsbäumen jeder einzelne Teilnehmer das User-Interface deuten konnte. Bei Lernenden, sollte also das User-Interface weniger formal und besser verständlich dargestellt werden, um einen optimalen Effekt zu erzielen.

Zu der Frage nach neuen wünschenswerten Interaktionsmöglichkeiten kamen hier viele Fragen, und auch Anregungen von den Teilnehmern. Oft wurde hier zuerst die Frage der Sinnhaftigkeit gestellt, also in welchem Szenario man denn einen Entscheidungsbaum gerne in VR sich anschauen möchte. Sie waren oft ein wenig verwirrt was sie denn nun hier machen sollten, und wo die Vorteile des relativ aufwändigen VR nun hier lagen.

Anregungen waren beispielsweise das eigene Ändern der Splitkriterien der Knoten, um dessen Auswirkungen und den neuen Fluss der Daten zu beobachten. Auch gab es die Rückmeldung, eventuell neue Knoten kreieren zu können, mit eigenen Splitkriterien. Die Teilnehmer wollten also die Möglichkeit haben ein wenig kreativ

und experimentell mit dem Baum umgehen zu wollen, einfach um zu beobachten was passiert. Auch wollte eine Person gerne die Möglichkeit, die Skalierung und Rotation des Baumes zu ändern, da der Baum doch relativ statisch in der Luft fliegt, sodass man also den ganzen Baum in der Hand halten kann, und mit der anderen Hand größer und kleiner ziehen kann. Alles in allem war die Meinung zum bisherigen Interaktionsfaktor so, dass sich viel mehr Möglichkeiten erwünscht werden. Dies war zu erwarten, da es bisher nur geringfügige prototypische Interaktionen in VR gibt.

Ein Anforderung, nach NF2, an die Software war es eine schwindelfreie VR Erfahrung zu erzeugen. Es hatten nur zwei der Teilnehmer vorherige, kurze, Erfahrung mit VR. Trotzdem hatte keiner bis zum Ende der Evaluation Probleme mit Schwindelgefühlen. Generell war einigen älteren Teilnehmern, die Plattform auf der sie in VR standen ein wenig zu klein.

Die zwei Befragten, welche schon Wissen zum maschinellen Lernen hatten, hatten gemeldet, dass sie ihr Wissen dazu nicht sonderlich erweitern konnten. Jedoch konnten diese sich nun dafür die Funktionsweise und den Aufbau eines Entscheidungsbaumes besser vorstellen. Zusätzlich konnten die anderen Teilnehmer, welche nichts über die Thematik wussten, sich nun ein besseres Bild über beide Gebiete machen. Die Kombination aus verbaler Erklärung, und der Abbildung eines Baumes in VR hatte demnach eine gute Resonanz zum Lernfaktor für die Teilnehmer.

Anschließend gab es auch eine Menge Ideen zur Verbesserung der Visualisierung. Vor allem wurde hier generell die noch sehr prototypische und spartanische Ästhetik kritisiert. Es wurde angemerkt, hier ein generelles Thema für die Umgebung zu suchen. Beispiele waren ein Klassenzimmer oder ein Labor, um einer schulenden Atmosphäre entgegenzukommen. Auch wurde der Wunsch nach mehr Farben geäußert. Ein Wunsch eines Experten war es auch den generellen Fluss der Daten zu visualisieren. Diese sollten Stück für Stück durch die Knoten laufen, und gegebenenfalls animiert werden. Auch sollte sich die Knotengröße je nach Datengröße anpassen. Ein wichtiger Punkt war auch die Differenzierung von normalen Knoten zu einem Blattknoten in der Visualisierung, welche noch nicht wirklich außerhalb des User-Interfaces gegeben ist.

Zu der Frage, wie die Teilnehmer die Splitkriterien leichter repräsentieren würden, gab es auch einige Anmerkungen. Vor allem, dass das User-Interface nur von einer Seite des Baumes zu sehen war, schien einigen Testern zu stören. Auch gab es die

Rückmeldung, das Splitkriterium nicht zu trennen, sondern an jeder Kante auszuschreiben, da es sonst bei Leuten ohne Hintergrundwissen zu Verwirrungen führen könnte. Die Meinung der Experten war hier jedoch, dass die Annotation korrekt dargestellt wurde, und so in ihrer Form hinreichend war.

6.3 Ergebnis

Zusammenfassend hatten die Teilnehmer sehr viele Verbesserungsvorschläge, was bei einer Software welche noch mitten in ihrer Entwicklung ist zu erwarten, und auch sehr erwünscht war. Es gab sehr viel Kritik zu fehlender Interaktion, und zu der sehr einfachen Ästhetik. Es wurde sich jedoch sehr positiv über den Lehrfaktor ausgesprochen, den solch eine Software bieten könnte. Die nicht funktionalen Anforderungen *NF2*, Beachtung der Motion Sickness, und *NF3*, Erkenntlichkeit der Informationen, wurden jedoch laut der Rückmeldung der Teilnehmer gut beachtet.

7 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde ein Unity-Package zur Visualisierung eines Entscheidungsbaumes programmiert und anschließend evaluiert. Es wurde der Grundstein gesetzt, um dieses Projekt noch weiter ausbauen zu können. Der erste Anlaufpunkt für die Erweiterung des Projektes sind die teilimplementierten Funktionsblöcke „Splitting der Nodes und „Interaktion in VR. Es muss hier also die *Execute()*-Funktion für den Split *CategoricalSplit* geschrieben, und getestet werden. Anschließend sollten zusätzliche Interaktionen in VR implementiert werden. Als Quelle für Ideen kann hier vor allem die Evaluation der Arbeit genutzt werden. Zusätzliche zu implementierende Interaktionen können sein ...

- ... eine Größeneinstellung des Baumes mit den Controllern.
- ... laden von anderen Bäumen während der Laufzeit. Hier muss es die Möglichkeit geben gut zu erkennen wie der Baum vorher aussieht.
- ... Änderung des Cutoff der Knoten durch den Nutzer während der Laufzeit, mit entsprechender Änderung des Datendurchfluss. Hierfür müsste ein passendes User-Interface kreiert, und implementiert werden.
- ... die Möglichkeit zusätzliche Knoten zu kreieren, und von diesen Knoten aus Kanten auf andere Knoten zu ziehen. Ebenfalls müsste hier ein passendes User-Interface kreiert werden, um diese Knoten mit einem Splitkriterium zu versehen.

Auch ist der Punkt der Software erreicht, um sich um eine schönere Ästhetik zu kümmern. Blattknoten sollten visuell anders dargestellt werden und die Umgebung sollte vom Standart-Unityhintergrund abweichen. Knoten sollten ein visuelles Feedback geben, wenn mehr oder weniger Datensätze durch sie durchgeflossen sind, und vielleicht können die die Kanten auch ein anderes aussehen bekommen, als eine weiße Linie. Aber auch sollten Implementationen, welche im nach hinein als suboptimal oder nicht fertig eingeschätzt wurden, noch einmal überholt werden. Beispielsweise

kann eine eigene De- und Serialisierung der Knoten geschrieben werden, um unabhängig von der Serialisierung eines *.Net*-Objektes zu werden.

Zum Abschluss lässt sich sagen, dass während der Arbeit an diesem Projekt einiges neues Wissen gesammelt wurde. Vor allem im Bereich des maschinellen Lernens, und der Entscheidungsbäume. Die Arbeit, ein Fundament eines größeren Projektes zu setzen, stellte sich als sehr komplex heraus. Am Anfang weiß man wohl kaum wo man anfangen soll, da doch noch sehr vieles zu implementieren gilt. Doch alles in allem, war dies eine lehrreiche Erfahrung.

Literaturverzeichnis

[Alp19] Ethem Alpaydin: *Maschinelles Lernen*, Walter de Gruyter GmbH & Co KG, 2019.

[B+14] Tim Bray *et al.*: *The javascript object notation (json) data interchange format*, 2014.

[Bal10] Helmut Balzert: *Lehrbuch der softwaretechnik: Basiskonzepte und requirements engineering*, Springer-Verlag, 2010.

[BEW+98] Lars Bishop, Dave Eberly, Turner Whitted, Mark Finch und Michael Shantz: *Designing a PC game engine*, *IEEE Computer Graphics and Applications*, Bd. 18(1):S. 46–53, 1998.

[BKI03] Christoph Beierle und Gabriele Kern-Isberner: *Methoden wissensbasierter Systeme*, Springer, 2003.

[Boy17] Brian Boyles: *virtual reality and augmented reality in Education*, *Center For Teaching Excellence, United States Military Academy, West Point, Ny*, 2017.

[Bri09] Manfred Brill: *Virtuelle Realität*, Springer, 2009.

[BS19] Peter Buxmann und Holger Schmidt: *Künstliche Intelligenz*, Springer, 2019.

[CBM08] Jeff Craighead, Jennifer Burke und Robin Murphy: *Using the unity game engine to develop sarge: a case study*, in *Proceedings of the 2008 Simulation Workshop at the International Conference on Intelligent Robots and Systems (IROS 2008)*, 2008.

[CIL13] Swaantje Wiarda Casjens, Katja Ickstadt und JProf Dr Uwe Ligges: *Adaption und Vergleich evolutionärer mehrkriterieller Algorithmen mit Hilfe von Variablenwichtigkeitsmaßen*, Dissertation, 2013.

[GS10] Günther Görz und Josef Schneeberger: *Handbuch der künstlichen Intelligenz*, Walter de Gruyter, 2010.

[Iva65] E Sutherland Ivan: *The ultimate display*, in *Proceedings of the IFIP Congress, Sutherland*, 1965.

[Kuh14] Torsten Kuhlen: *Virtuelle Realität als Gegenstand und Werkzeug der Wissenschaft*, in *Exploring Virtuality*, S. 133–147, Springer, 2014.

[ND17] Vinh T Nguyen und Tommy Dang: *Setting up virtual reality and augmented reality learning environment in unity*, in *2017 IEEE International Symposium on Mixed and Augmented Reality (ISMAR-Adjunct)*, S. 315–320, IEEE, 2017.

[New] Newtonsoft: *Abbildung Vergleich Json Frameworks*, URL: https://www.newtonsoft.com/content/images/jsonperformance.png, besucht am 23.01.2020.

[PRS+16] Felipe Pezoa, Juan L Reutter, Fernando Suarez, Martín Ugarte und Domagoj Vrgoč: *Foundations of JSON schema*, in *Proceedings of the 25th International Conference on World Wide Web*, S. 263–273, 2016.

[RN12] Stuart Russell und Peter Norvig: *Künstliche Intelligenz*, Pearson Deutschland GmbH, 2012.

[SG19] Steven Schürstedt und Andreas Geiger: *Einsatz von VR-Technologien in BIM/GIS*, in *31. Forum Bauinformatik: 11.–13. September 2019 in Berlin. Proceedings*, S. 283, Universitätsverlag der TU Berlin, 2019.

[Sha05] Yakov Shafranovich: *Common format and MIME type for comma-separated values (CSV) files*, 2005.

[Tec19] Unity Technologies: *MonoBehaviour*, 2019, URL: https://docs.unity3d.com/ScriptReference/MonoBehaviour.html, besucht am 03.02.2020.

[Wal90] John Q Walker: *A node-positioning algorithm for general trees*, *Software: Practice and Experience*, Bd. 20(7):S. 685–705, 1990.

[WH18] Stefan Wrobel und Dirk Hecker: *Fraunhofer-Allianz Big Data*, in *Digitalisierung*, S. 261–273, Springer, 2018.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Mittweida, den 11. März 2020

Andre Spittel