Hochschule Kempten
University of Applied Sciences

Bachelorarbeit im Studiengang Informatik - Game Engineering

PROZEDURAL GENERIERTE VEGETATION IN EINER AUGMENTED REALITY UMGEBUNG

Philipp Vidal

Kurzbeschreibung

Diese Arbeit beschreibt das Design und die Implementierung einer Augmented Reality-Anwendung, in der ein Raum glaubwürdig durch prozedural generierte Vegetation überwachsen wird. Die Anwendung läuft hierbei auf der HoloLens 2 und wurde in Unity erstellt. Jegliche Vegetation der Anwendung wird entweder prozedural platziert oder vollkommen prozedural erstellt, um sie an die Umgebung des Spielers anzupassen. Die Anwendung besitzt zudem einfache spielerische Elemente, wie etwa das Erhalten von Punkten für das Platzieren von Pflanzen und ein Highscore-System, wodurch größeres Interesse bei dem Benutzer geweckt werden soll.

Aufgabensteller/Prüfer:	Prof. Dr. Christoph Bichlmeier
Arbeit vorgelegt am:	16.12.2020
Durchgeführt an der:	Fakultät für Informatik

VORWORT:

Ich möchte mich recht herzlich bei Prof. Dr. Christoph Bichlmeier für die Unterstützung bei Fragen, die Betreuung während der Bachelorarbeit und das Ausleihen der HoloLens 2 bedanken.

INHALTSVERZEICHNIS

1 EINLEITUNG

Viele populäre Videospiele setzen heutzutage nicht nur Wert darauf, sich durch unterhaltsames und innovatives Gameplay auszuzeichnen, sondern auch auf die Darstellung einer lebendigen Spielwelt, in die der Spieler mit Leichtigkeit eintauchen kann. Dabei liegt der Fokus oft auf einer stimulierenden Handlung und auf dem Erschaffen einer immersiven Atmosphäre, um eine glaubhafte virtuelle Welt zu kreieren. Für die Atmosphäre der Spielwelt stellt hier die Vegetation eine sehr wichtige Rolle dar. Durch die Vegetation kann der Spieler, häufig unterschwellig, Informationen über seine Umgebung erhalten, die ihm nicht direkt durch die Handlung des Spiels oder andere Quellen, wie etwa NPCs, zur Verfügung gestellt werden. Palmen und andere tropische Pflanzen können den Spieler so zum Beispiel direkt vermuten lassen, dass er sich in der Nähe des Äquators, z.B. auf einer tropischen Insel, befindet. Genauso können kahle Bäume und Sträucher sofort den Eintrug beim Spieler erwecken, dass die Handlung des Spieles in den Wintermonaten oder in einer kälteren Region der Spielwelt stattfindet. Die Vegetation eines Videospiels kann daher einen wesentlichen Bestandteil des Game-Designs und der Entwicklung repräsentieren, obwohl diese bei den Spielern häufig nicht im Vordergrund steht oder direkt wahrgenommen wird und stattdessen das Spielerlebnis eher indirekt bereichert.

Aufgrund der hohen Immersion, die Spieler bei dem Konsum von Videospielen erleben können, eigenen sich diese auch ausgezeichnet gut dafür, dem Spieler eine Flucht aus der Realität zu ermöglichen. Das kann weiter verstärkt werden, indem anstatt eines gewöhnlichen Monitors mit Tastatur und Maus bzw. Kontroller, Technologien wie Augmented- und Virtual Reality (kurz AR und VR) verwendet werden. Bei diesen Technologien trägt der Spieler üblicherweise eine Brille mit integrierten Monitoren auf seinem Kopf, ein sogenanntes Head-Mounted Display (HMD), dass ihm den Eindruck gibt, mitten im Spielgeschehen zu sein. Bei Augmented Reality gibt es zwar alternative Darstellmöglichkeiten, wie z.B. durch den Bildschirm eines Smartphones, jedoch kann mit dem Nutzen eines HMDs leichter eine höhere Immersion erreicht werden. AR und VR haben in den letzten Jahren sehr stark an Popularität gewonnen, was bei VR insbesondere der Spieleindustrie zu danken ist. Ähnlich ist es bei Augmented Reality, jedoch fokussiert sich die Spieleindustrie bei AR vor allem auf das Entwickeln von Spielen und Anwendungen für Smartphones, da diese bereits weit verbreitet und günstiger sind, als die meisten HMDs. HMDs finden im Gebiet der Augmented Reality besonders in anderen Bereichen, wie etwa dem Gesundheitswesen, Anwendung. In diesem Bereich wird AR auch häufig in der Form von Videospielen genutzt, aber es handelt sich hierbei oft um eine spezielle Art von Videospielen, nämlich die sogenannten Serious Games. Serious Games sind Videospiele, deren Hauptfokus nicht nur die Unterhaltung des Nutzers ist, sondern auch ein tieferer Sinn. Im Gesundheitswesen werden sie zum Beispiel in der Therapie von posttraumatischen Belastungsstörungen und anderen Störungen genutzt. Augmented Reality kann in diesem Bereich auch als Hilfsmittel bei Operationen und zur visuellen Darstellung von Patientendaten verwendet werden. Augmented Reality Serious Games können zusätzlich in anderen Industrien, wie etwa der Automobilindustrie, Verwendung finden und auch zur Schulung von Personal verwendet werden.

Ein allgemeiner Trend in der Spieleindustrie ist es, jedes Spiel größer und besser zu machen als das Vorherige, um es von den Vorgängern und etwaiger Konkurrenz abzuheben und den Spieler dazu zu animieren, das neue Spiel zu kaufen. In vielen Fällen bedeutet das, dass die Spielwelt und der Bereich, der durch die Spieler betreten und erkundet werden kann, auch an Umfang zunimmt. Ein Nachteil, der sich oft aus solch einem „größeren“ und „besseren“ Videospiel ergibt, ist daher, dass mehr Aufwand in das Erstellen der Spielwelt gesteckt werden muss, um diese glaubwürdig und lebendig wirken zu lassen. Um immer größere virtuelle Welten zu erstellen, werden natürlich auch immer mehr Objekte benötigt, mit denen die Welt des Spieles

gefüllt werden kann. Das führt dazu, dass in das Erschaffen dieser Objekte und anderer Spieleinhalte mehr Zeit und Ressourcen investiert werden müssen. Eine Lösung für dieses Problem ist das Nutzen von prozeduraler Generation, um die Spielwelt mit Vegetation und anderen Objekten zu füllen, was zum Beispiel bei der Entwicklung des Videospiels „Horizon Zero Dawn“ von Guerilla Games eingesetzt wurde. Wie in einer Präsentation von Jaap van Muijden[1] bei der GDC[2] im Jahr 2017 zu sehen ist, wurde in diesem Spiel ein prozedurales Platzierungssystem, dass auf der Grafikkarte läuft, implementiert, um einen Großteil der Flora und anderer Objekte in der Welt des Spieles prozedural zu platzieren. Prozedurale Generation eignet sich zusätzlich hervorragend dafür, Variationen in die Modelle von Pflanzen zu bringen und diese an ihre Umgebung anzupassen.

1.1 THEMENVERWANDTE ARBEITEN

Das Erstellen und Anpassen eines Augmented Reality-Spieles an die Umgebung des Spielers und das Platzieren von Spieleinhalten an geeigneten Flächen dieser Umgebung kann eine schwierige Aufgabe sein. Ein Ansatz hierfür wird in der Arbeit „Procedural Level Generation for Augmented Reality Games“ von Sasha Azad et al. (2016) vorgestellt. In der Arbeit wird ein AR-Plattformer namens „MR Lemmings“ beschrieben, bei dem der Spieler versucht, virtuelle Charaktere durch das Platzieren von Objekten zu einem Ziel zu steuern. Die Objekte, die der Spieler dafür benutzt, sind Hindernisse, die die Richtung der Charaktere beeinflussen und Hilfsobjekte, die sie über Lücken in der Umgebung katapultieren. (S. 247). Der Raum wird bei der Anwendung, ähnlich wie bei der HoloLens 2, durch eine spezielle Kamera abgetastet, um ein verwendbares 3D-Modell zu erhalten. (S. 248). Die Anwendung identifiziert die vertikalen und horizontalen Flächen des Spielbereichs und analysiert diese, um den weiteren Spielablauf zu ermöglichen. (S. 248). Am Ende der Arbeit machen Sasha Azad et al. die folgende Aussage (Azad et al., 2016, S. 248):

> Most home and office environments are unlikely to be perfectly suitable game playing environments. The MR Lemmings game demonstrates how procedural content generation algorithms can constrain the gameplay to those portions of the physical world that are most conducive to gameplay.

Wie diese Aussage schon anmerkt, kann es sein, dass nicht alle Bereiche eines Raumes in AR für prozedural erstellte Inhalte geeignet sind. Es kann also eine gute Idee sein, die prozedurale Generation nur auf die Stellen im Raum zu beschränken, die sich auch wirklich dafür anbieten.

In Bezug auf das prozedurale Erstellen der Vegetation gibt es auch zwei themenverwandte Arbeiten, die sich mit dem prozeduralen Erstellen von Kletterpflanzen, ähnlich wie es in dieser Arbeit getan wird, beschäftigen. Zum einen gibt es hier die Arbeit von Hädrich et al. (2017) und zum anderen die Arbeit von Johan Knutzen (2009). Hädrich et al. beschreiben in der Arbeit „Interactive Modeling and Authoring of Climbing Plants“ das prozedurale Generieren von Kletterpflanzen durch das Verwenden von verbundenen anisotropen Partikeln. (S. 50), was dem Nutzer, im Vergleich zu anderen Ansätzen, sehr große Kontrolle über das konkrete Wachstum der Pflanzen verspricht. (S. 50). Die Kletterpflanzen können bei diesem Ansatz zur Laufzeit erstellt werden und verhalten sich weitestgehend physikalisch korrekt. (S. 50). Bei der Arbeit, die Johan Knutzen im Jahr 2009 veröffentlicht hat, werden die Kletterpflanzen stattdessen unter Verwendung eines L-Systems erstellt. Der Fokus der Arbeit liegt laut Johan Knutzen nicht darin, biophysiologisch akkurat zu sein, sondern ein Werkzeug für das potenzielle Erreichen der Ziele

[1] Präsentation von Jaap van Muijden auf dem Youtube-Kanal der GDC veröffentlicht am 27.12.2019 - https://www.youtube.com/watch?v=ToCozpl1sYY

[2] Games Development Conference (GDC) - https://gdconf.com/

von Designern zu liefern. (Knutzen, 2009, S. 7). Ein relevantes Problem, auf das Knutzen in der Arbeit gestoßen ist, ist das Anfangs versucht wurde, die Kletterpflanzen biophysiologisch akkurat zu halten, jedoch hat das laut ihm zu einem langsamen Ergebnis geführt, das keinen signifikanten Mehrwert in Hinsicht auf den Realismus geboten hat. (Knutzen, 2009, S. 42).

1.2 ZIELSETZUNG UND AUFBAU DER ARBEIT

Das Ziel dieser Bachelorarbeit ist es, eine Augmented Reality-Anwendung für die HoloLens 2, eine Mixed Reality-Brille von Microsoft, zu erstellen, in der der Raum, in dem sich der Benutzer befindet, langsam durch die Natur erobert wird. Die Anwendung wird mit der Unity Game Engine erstellt und die unterschiedliche Vegetation, die in der Anwendung Verwendung findet, um die Oberflächen des Raumes zu überwachsen, soll prozedural generiert werden. In der Anwendung muss daher ein Weg gefunden werden, die verschiedenen Pflanzenarten prozedural zu erstellen oder prozedural zu platzieren und dabei auf die jeweiligen Eigenschaften der Oberflächen des Raumes zu achten. Bei der prozeduralen Generation der Vegetation müssen grundsätzlich die folgenden Ziele erfüllt werden:

- Die verschiedenen Pflanzenarten können nur an Oberflächen des Raumes platziert werden, die für sie geeignet sind.
- Sollte die jeweilige Pflanzenart prozedural erstellt werden, muss diese ihre Umgebung bei der Erstellung berücksichtigen.
- Der Raum muss durch die generierte Vegetation auf eine glaubhafte Weise überwachsen wirken.

ABBILDUNG 1: EINE SPIELSZENE AUS DER ERSTELLTEN ANWENDUNG.

Zu Beginn der Arbeit wird ein kurzer Überblick über die Software und Hardware gegeben, die bei der Entwicklung der Anwendung verwendet wurde. In Kapitel 3 wird auf die Grundlagen der prozeduralen Generation und Vegetation in 3D-Anwendungen eingegangen. Anschließend wird in Kapitel 4 das Design der erstellen Anwendung und ihrer einzelnen Komponenten erläutert und allgemeine Anforderungen an diese festgelegt. Im fünften und sechsten Kapitel der Arbeit

wird die konkrete Umsetzung des Designs in der Form der individuellen Komponenten erklärt und die Verbesserung der Anwendungsperformance durch Optimierung an verschiedenen Stellen beschrieben. Im Anschluss werden die Ergebnisse der Arbeit diskutiert. Hierbei werden die bei der Entwicklung aufgetretenen Probleme angesprochen, die Mängel der fertigen Anwendung offengelegt und unterschiedliche Ideen für zukünftige Arbeiten in verschiedenen Bereichen geliefert. Das letzte Kapitel liefert eine detailreiche Zusammenfassung der gesamten Arbeit.

2 VERWENDETE TECHNOLOGIEN

Bei der Entwicklung der Anwendung wurden mehrere Software- und Hardware-Technologien verwendet, zu denen in diesem Kapitel kurz einige grundlegende Information gegeben werden, um einen grundsätzlichen Überblick zu verschaffen.

2.1 AUGMENTED REALITY UND HOLOLENS 2

Für den Begriff „Augmented Reality“ gibt es viele Definitionen, die sich oft auf verschiedene Aspekte dieser Technologie fokussieren. Die bei Weitem am häufigsten verwendete Definition ist jedoch die, die sich auf das visuelle Überlagern von Information über die reale Welt konzentriert. Mit dem Begriff „Augmented Reality“ ist damit in den meisten Fällen, wie von Carmigniani et al. definiert, das direkte und indirekte Erweitern der Realität durch computergenerierte Informationen in Echtzeit gemeint. (Carmigniani et al., 2010, S. 342). Das Ganze kann auch wie von Azuma et al. definiert werden. Sie definieren den Begriff „Augmented Reality“ anhand der folgenden Punkte (Azuma et al., 2001, S. 34):

- Es kombiniert reale und virtuelle Objekte in einer realen Umgebung
- Das Ganze ist interaktiv und läuft in Echtzeit
- Virtuelle Objekte werden anhand realer Objekte ausgerichtet

Mit Augmented Reality sehr eng verwandte Bereiche sind Virtual Reality und Mixed Reality. Virtual Reality und Augmented Reality sind in der Regel leicht zu unterscheiden, da sich Virtual Reality rein auf den virtuellen Raum bezieht. Der grundsätzlich größte Unterschied ist dabei, dass die Inhalte bei Virtual Reality in einer virtuellen Spielwelt gerendert werden, hingegen werden die Inhalte bei Augmented Reality lediglich über die reale Welt gelegt. Der Begriff „Mixed Reality“ kann hierbei als Sammelbegriff für sowohl AR als auch VR verwendet werden, aber er wird heutzutage oft als Synonym für Augmented Reality genutzt, obwohl die beiden Begriffe technisch gesehen nicht die gleichen Dinge beschreiben.

Augmented Reality kann auf vielen verschiedenen Gerätetypen genutzt werden, jedoch sind die am weitesten verbreiteten Gerätearten herkömmliche Smartphones und sogenannte Head-Mounted Displays (HMDs). Hierbei können die Qualität und Nutzererfahrung stark mit dem Preis und der Qualität des Gerätes variieren. Head-Mounted Displays haben den großen Vorteil, dass sie den Nutzer stärker in das Geschehen der Anwendung vertiefen können, ohne dabei unnatürlich zu wirken. Das ist vor allem der Fall, da man bei der Benutzung eines HMDs die Hände im Normalfall frei hat und diese oft auf natürliche Art zum Steuern der Anwendung nutzen kann.

ABBILDUNG 2: DIE HOLOLENS 2, EIN AR-HMD VON MICROSOFT.

Die HoloLens 2[3] von Microsoft[4] ist ein drahtloses high-end HMD, dass nicht nur das grundsätzliche Nutzen der Hände für den Gebrauch in Augmented Reality ermöglicht, sondern zusätzlich auch das Erkennen verschiedener Handgesten, Eyetracking und Sprachbefehle anbietet. Im Vergleich zu ihrem Vorgänger, der HoloLens 1, hat die HoloLens 2 ein höheres Sichtfeld (Field of View), neue Funktionen und sie besitzt einen leistungsstärkeren Prozessor, was zu höherer Rechenleistung führt. Da es nicht kabelgebunden ist, eignet sich dieses HMD besonders dafür, dem Benutzer uneingeschränkte Bewegung innerhalb seiner Umgebung zu ermöglichen.

Die wichtigste Funktion der HoloLens 2, die in der Anwendung dieser Arbeit genutzt wird, ist das Spatial Mapping. Spatial Mapping ist das Abtasten der realen Umgebung des Benutzers und das Erstellen eines Meshes aus den erhaltenen Daten mithilfe verschiedener Algorithmen. Das erhaltene Mesh ist eine relativ akkurate, virtuelle Darstellung der realen Umgebung des Benutzers, die im späteren Anwendungsverlauf weiter genutzt und verarbeitet werden kann. Am häufigsten wird dieses Spatial Mapping-Mesh für die Kollisionserkennung mit anderen Objekten verwendet, jedoch können damit auch Objekte, die in der Szene platziert wurden, verdeckt werden, was die Illusion, dass es sich um ein reales Objekt handeln könnte, verstärkt.

[3] Microsoft HoloLens 2 - https://www.microsoft.com/en-us/hololens/
[4] Microsoft Corporation - https://www.microsoft.com

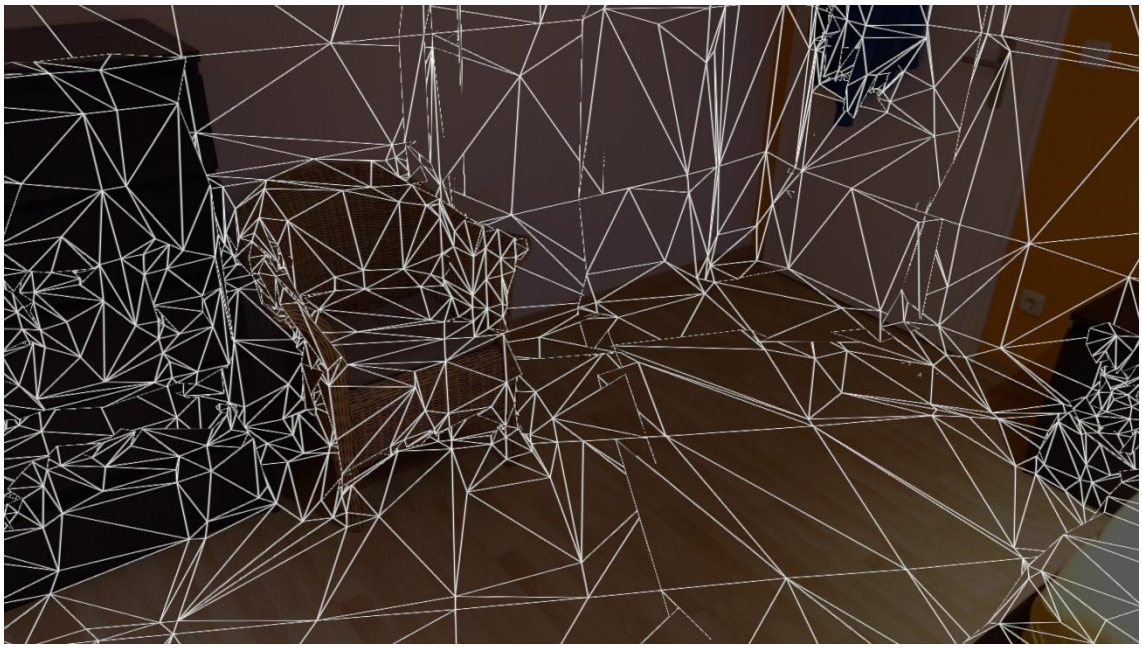

ABBILDUNG 3: DAS ABTASTEN EINES RAUMES IN DER ANWENDUNG AUS DER SICHT DES NUTZERS.

2.2 UNITY ENGINE

Für das Erstellen der Anwendung wurde die Game Engine Unity[5] verwendet. Diese Game Engine ist sehr populär in der Entwicklung von Anwendungen für mobile Geräte, insbesondere werden damit häufig vor allem Videospiele für Smartphones entwickelt. Die Engine selbst ist größtenteils komponenten-basiert, dass heißt, es werden verschiedene Komponenten an sogenannte Game-Objects gehangen. Game-Objects stellen hierbei abstrakte Objekte in der Spielwelt dar, die grundsätzlich nur eine Transformations-Komponente besitzen, diese speichert wiederrum die Position, Rotation und Skalierung des Objektes in der Spielwelt. Erst durch das Hinzufügen weiterer Komponenten wird die spezifische Rolle des Game-Objects näher definiert. Diese hinzugefügten Komponenten können zum Beispiel für das Rendern des Meshes des Objekts verantwortlich sein (z.B. Mesh-Filter, Mesh-Renderer) oder auch als Schnittstelle zum Physik-System der Engine dienen (z.B. Physics-Collider). Der Entwickler kann jedoch auch eigene Komponenten in der Form von Skripten hinzufügen. Skripte werden normalerweise in der Programmiersprache C# geschrieben und stellen eine eigene Klasse dar, die über verschiedene, von Unity vorgegebene und eigene Funktionen verfügen kann. Die wichtigsten Funktionen hierbei sind die Update-Funktion, die bei jedem Tick der Anwendung gerufen wird, sowie die Awake- und Start-Funktionen, welche jeweils zu verschiedenen Zeitpunkten bei der Instanziierung und dem Aktivieren des Game-Objects aufgerufen werden. Da Skripte in den meisten Fällen Komponenten von Spieleobjekten sind und diese die erstellten Klassen darstellen, werden in dieser Arbeit die Begriffe „Skript“, „Komponente“ und „Klasse“ weitestgehend synonym verwendet.

[5] Unity Technologies - https://unity.com/

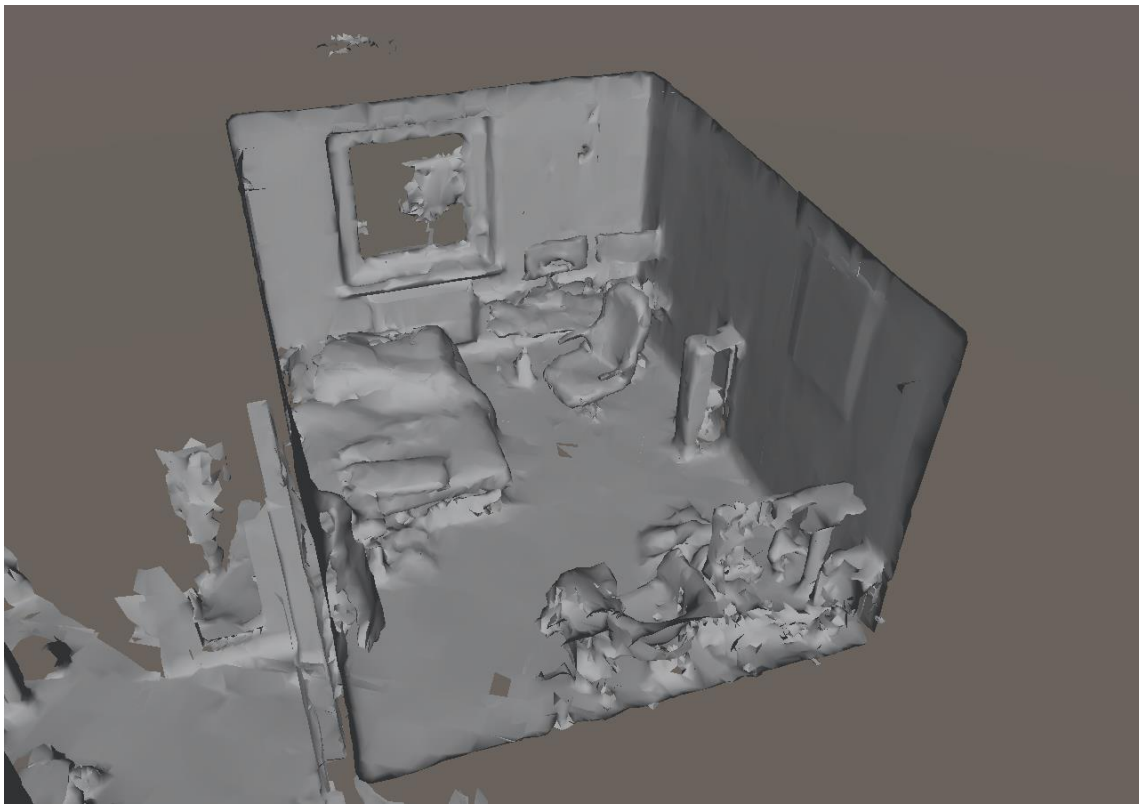

ABBILDUNG 4: DAS SPATIAL MAPPING-MESH EINES RAUMES IN UNITY.

2.3 MIXED REALITY TOOL KIT V2

Das Mixed Reality Tool Kit v2[6] (MRTK) ist ein Software-Toolkit, dass von Microsoft für das Entwickeln von Mixed Reality Anwendungen zur Verfügung gestellt wird und den Entwicklungsprozess vereinfachen soll. Mit dem MRTK können AR-Anwendungen für die HoloLens 2 programmiert werden, aber es werden auch andere Augmented Reality- und Virtual Reality-Plattformen unterstützt. Das MRTK v2 ist der Nachfolger des Mixed Reality Tool Kits (v1) und kann auch als Nachfolger des HoloToolKits angesehen werden, welches für die Entwicklung mit der ersten HoloLens verwendet wurde. Laut Microsoft[7] bietet das MRTK v2, im Vergleich zu seinen Vorgängern, ein modulares Design, Cross-Platform Support und bessere Performance. (Microsoft, 2019). Ein weiterer Vorteil ist, dass der Großteil der Einstellungen für die einzelnen Komponenten des Software-Toolkits innerhalb des Editors der Unity Game Engine mithilfe von Profilen angepasst werden kann, was das grundsätzliche Entwickeln stark erleichtert. Als größter Nachteil kann das Fehlen von mehreren nützlichen Funktionen der Vorgänger angesehen werden. So werden zum Beispiel die Spatial Understanding-Algorithmen, die das Erkennen und Kategorisieren der Umgebung des Benutzers ermöglichen, momentan nicht in der zweiten Version des Mixed Reality Tool Kits angeboten. Mit diesen Algorithmen können verschiedene Möbel innerhalb eines Raums, sowie Wände, Decken und Böden erkannt werden. Eine weitere Funktion, die fehlt, ist das Weiterverarbeiten (Mesh-Processing) des, von der HoloLens 2, erhaltenen Meshes um ggf. Löcher und Unebenheiten im Mesh zu entfernen und zu glätten.

---

[6] Mixed Reality Toolkit - https://github.com/microsoft/MixedRealityToolkit-Unity

[7] Microsoft: Getting started with MRTK for Unity - https://docs.microsoft.com/en-us/windows/mixed-reality/develop/unity/mrtk-getting-started

3 GRUNDLAGEN

3.1 PROZEDURALE GENERATION

Der Bereich der prozeduralen Generation, auch *Procedural Content Generation* (kurz PCG) genannt, umfasst viele Gebiete und kann daher auf unterschiedliche Weise definiert werden. Bidarra et al. definieren prozedurale Generation sehr einfach als das automatische Erstellen von Inhalten mithilfe von Algorithmen. (Bidarra et al., 2014, S. 78). In Bezug auf Videospiele und künstliche Intelligenz kann man prozedurale Generation auch wie Gillian Smith definieren (Smith, 2015, S. 501):

> Procedural content generation (PCG) is the process of using an AI system to author aspects of a game that a human designer would typically be responsible for creating, from textures and natural effects to levels and quests, and even to the game rules themselves.

Um eine einheitliche Definition zu erreichen, wird der Begriff der prozeduralen Generation für diese Bachelorarbeit wie folgt definiert:

> Prozedurale Generation bezeichnet das Erstellen von Inhalten mithilfe von Algorithmen, ohne diese komplett manuell erstellen oder verändern zu müssen. Diese Inhalte werden teilweise oder vollständig in Abhängigkeit von anderen Parametern und Eigenschaften der Spielwelt erstellt.

Wie schon erwähnt, findet prozedurale Generation in vielen verschiedenen Bereichen Anwendung, hierzu gehören unter anderem Filme, Musik und digitale Anwendungen, wie etwa Videospiele. Bei Videospielen und anderen 3D-Anwendungen werden damit häufig Texturen, 3D-Modelle und in einigen Fällen ganze Spielwelten erschaffen. Ein gutes Beispiel dafür ist das Videospiel No Man's Sky[8], bei dem der Großteil der Spielwelt prozedural erstellt wird. Dazu gehören jegliche Flora und Fauna der Spielwelt, sowie ganze Planeten und Sonnensysteme.

Für das prozedurale Generieren von Spieleinhalten gibt es laut Gillian Smith (2015) mehrere Ansätze (S. 503), welche auch teilweise miteinander vermischt werden können (S. 508) und die einzelnen Grenzen dieser Ansätze können, je nach Art der Implementierung, schwer einzustufen sein. Drei der Ansätze von Gillian Smith, die für diese Arbeit relevant sind, werden im Folgenden kurz erläutert (Smith, 2015, S. 503 f.):

Simulation Based

Bei einem simulations-basierten Ansatz wird der Inhalt durch Simulation erstellt, ohne wirkliche Kontrolle über das Endergebnis zu haben und diese Simulation kann vor allem auch zur Laufzeit durchgeführt werden. (Smith, 2015, S. 503). Die prozedurale Generation der Kletterpflanzen in der Anwendung dieser Arbeit lässt sich hierbei leicht diesem Ansatz zuordnen, da die Kletterpflanzen selbst für das Suchen eines Pfades zur Laufzeit verantwortlich sind und keine direkte Kontrolle ausgeübt wird.

Constructionist

Hierbei wird, laut Gillian Smith, mithilfe eines Algorithmus, aus vorgefertigten Bausteinen ein prozeduraler Inhalt generiert und die Regeln für das Erstellen sind mehr oder weniger allein dem implementierten Algorithmus überlassen. (Smith, 2015, S. 503). Ein Vorteil hiervon ist, dass die einzelnen vorgefertigten Bausteine manuell erstellt werden können und damit keiner der Limitationen von PCG unterliegen. Zu diesem Ansatz kann man das Platzieren der Pflanzen auf

[8] Offizielle Website von No Man's Sky - https://www.nomanssky.com/

horizontalen Flächen in der Anwendung dieser Arbeit zählen. Die Pflanzenmodelle dienen hier als vorgefertigte Bausteine, die durch den Algorithmus in der Spielwelt zusammengefügt werden.

Grammars

Hier werden Regeln für das Erstellen der Inhalte festgelegt, die durch eine formale Grammatik umgesetzt werden. (Smith, 2015, S. 504). Die Eingabe der formalen Grammatik und ihrer Regeln kann in vielen Fällen von Entwicklern und Designern leicht verändert werden, um das Endergebnis anzupassen. Ein Beispiel für das Generieren von prozeduralen Inhalten, mithilfe einer formalen Grammatik, sind sogenannte L-Systeme. L-Systeme sind besonders gut, um sich-selbst-ähnelnde Gebilde, wie etwa Bäume, deren Äste und andere komplexe Vegetation darzustellen.

Das Nutzen von prozeduraler Generation bei der Entwicklung eines Spiels, unabhängig davon, ob das Spiel komplett darauf basiert oder nur Teile des Spieles prozedurale Generation nutzen, kann viele Vorteile mit sich bringen. Zum einen kann man damit Abwechslung in den Verlauf des Spiels bringen, vor allem wenn z.B. der Spieler den gleichen Spielabschnitt öfters durchläuft. Smith, Short und Adams nutzen hierfür den Begriff Replayability (Smith, 2015, S. 501, Short & Adams, 2017, S. 9) und Smith gibt als Beispiel dafür das Anpassen des Schwierigkeitsgrades, je nach Fähigkeitslevel des Spielers. (Smith, 2015, S. 501). Damit wird der Spieler dazu angeregt, z.B. das gleiche Level im Spiel mehrmals zu besuchen. Short und Adams erwähnen einen weiteren Vorteil, nämlich dass sich damit leicht real anfühlende, „lebendige“ Systeme erstellen lassen, die, im Vergleich zu manuell erstellten Inhalten, weniger repetitiv wirken können. (Short & Adams, 2017, S. 11). Natürlich gibt es auch viele positive Aspekte auf der Seite der Entwickler, so kann damit zum Beispiel der allgemeine Speicherbedarf des Spiels gesenkt werden, da viele Spieleinhalte so gesehen erst zur Laufzeit erstellt werden. Mit PCG können auch Inhalte generiert werden, die sich hervorragend an die Spielwelt und die Umgebung des Spielers anpassen. Für manche Spiele kann dieser Aspekt besonders wichtig sein, vor allem wenn die Spielwelt oder die direkte Umgebung im Voraus nur teilweise oder gar nicht bekannt ist.

Obwohl mit prozeduraler Generation viele Vorteile einhergehen, gibt es auch Aspekte, die das Spielerlebnis des Spielers und die Entwicklung der Anwendung negativ beeinflussen können. Es wird in der Regel zwar weniger Speicherplatz für die prozedural erstellten Inhalte auf der Festplatte benötigt, jedoch kann das Erstellen dieser zur Laufzeit bedeuten, dass mehr Rechenleistung, im Vergleich zu bereits manuell erstellten Inhalten, benötigt wird. Ein weiterer solcher Nachteil ist, dass prozedurale Generation das Testen und Debuggen der Inhalte stark erschweren kann, insbesondere wenn diese prozedural erstellten Inhalte sehr wenig Eingabe des Entwicklers benötigen und sich sozusagen „selbst überlassen“ sind. Viele Entscheidungen für das spezifische Design eines Inhaltes liegen dadurch nicht direkt in den Händen der Entwickler und Designer, sondern müssen, teils vage, durch das Verhalten des implementierten Algorithmus festgelegt werden. Laut Short und Adams kann es auch sein, dass sich prozedural generierte Inhalte negativ auswirken, wenn der Endnutzer das Endergebnis der prozeduralen Generation vorhersehen kann, auch wenn es ihm nur so vorkommt. (Short & Adams, 2017, S. 19).

Es gibt auch Aspekte die gleichzeitig als positiv und negativ aufgefasst werden können, je nach dem, an welcher Stelle der Entwicklung man sich befindet. In manchen Fällen kann das Ergebnis des prozeduralen Erstellens, wie Short & Adams erwähnen, sehr schwer einzuschätzen sein. (Short & Adams, 2017, S. 11). Nach Short und Adams kann das den Entwicklern zugutekommen, da diese das Ergebnis aus den Augen eines Spielers betrachten können, jedoch sind sie auch der Meinung, dass dieser Aspekt von Nachteil für die Quality Assurance (QA) des Spieles sein kann.

(Short & Adams, 2017, S. 11). Ähnliches wird von Short & Adams in Hinsicht auf das prozedurale Erstellen von Leveln eines Spieles angemerkt (Short & Adams, 2017, S. 6):

> One of the most prominent blockers to the use of PCG in mainstream games, in particular AAA games, is quality assurance (QA). A PCG game may not be guaranteed to work as intended 100% of the time. There is no way of having testers go through every single iteration of the procedural content. The generator may have serious bugs in 0.1% of its levels that are never picked up until it gets released to a wide playerbase.

Wann PCG verwendet werden sollte, ist aufgrund der vielen unterschiedlichen Aspekte und der Tatsache, dass viele davon sowohl positive als auch negative Auswirkungen auf das Endprodukt haben können, je nach Projekt unterschiedlich. Laut Short und Adams kann es für ein Spiel unabdingbar sein, prozedural generierte Inhalte zu verwenden, wenn das grundlegende Design, die Ziele des Spieles und die Art des Spielprinzips auf der Nutzung dieser Inhalte aufgebaut sind. (Short & Adams, 2017, S. 3 f.). Ein Beispiel, dass dafür von Short & Adams angegeben wird, ist das schon erwähnte Spiel No Man's Sky. Ohne PCG wäre es unmöglich Planeten und vor allem keine ganzen Sonnensysteme, in dem Maßstab zu erstellen, wie es in No Man's Sky getan wird. Natürlich macht es auch nur Sinn prozedurale Generation für ein Spiel zu verwenden, wenn die negativen Aspekte, die damit einhergehen, nicht überwiegen und/oder unter Umständen umgangen werden können, um den Spielverlauf der Anwendung und die Erfahrung des Spielers nicht zu beeinflussen. Ein Weg zum Beispiel das Problem der Quality Assurance zu umgehen, ist wie Short & Adams erwähnen, PCG nur zum Erstellen eines Entwurfs des Inhalts zu nutzen und die letztendlichen Spieleinhalte vor der Implementierung manuell auf die Qualität zu überprüfen und gegebenenfalls Änderungen per Hand vorzunehmen. (Short & Adams, 2017, S. 4).

3.2 VEGETATION IN 3D-ANWENDUNGEN

In Videospielen spielt Vegetation eine zentrale Rolle für das Erschaffen einer glaubhaften Spielwelt. Die Atmosphäre des Spiels kann durch das Auswählen spezifischer Pflanzenarten und deren Variationen gezielt beeinflusst werden. In Spiele-Serien wie Metro und Fallout wird davon sehr stark gebraucht gemacht. In beiden Spielen findet sich der Spieler in einer post-apokalyptischen Welt, die durch einen Atomkrieg zerstört wurde. Hierbei kann der Spieler von toten Bäumen und wildem, überwachsenen Gras, bis hin zu verstrahlten, leuchtenden Pilzen und anderen mutierten Pflanzenarten, jegliche Vegetation finden, die man als typisch in solch einer postnuklearen Spielwelt ansehen kann. Durch die auftauchende Vegetation kann dem Spieler auch indirekt und unterschwellig beigebracht werden, in welchem Teil der Welt er sich befindet, oder ob er sich überhaupt auf der Erde befindet, und was er von der Spielwelt zu erwarten hat. Durch gezieltes Auswählen der Vegetation und des Designs der verschiedenen Pflanzen können die Entwickler und Designer eines Videospieles diese Informationen oft schon direkt mit dem ersten Blick des Benutzers offensichtlich machen und dadurch ein intuitives, immersives Spielerlebnis für den Spieler schaffen.

Da Vegetation so eine große Rolle in Videospielen spielen kann, muss man sich als Entwickler, abgesehen von den Entscheidungen zum Design der Vegetation, die man treffen muss, um die Atmosphäre des Spiels richtig einzufangen, auch die Frage stellen, wie die Vegetation implementiert wird. Bei der Implementation gibt es viele Tricks, die man verwenden kann, um das gewünschte Ergebnis zu erzielen, diese können sehr schnell sehr kompliziert werden und viel Aufwand von Seiten der Entwickler beanspruchen. Wie in einer Präsentation von Gilbert

Sanders[9], einem Mitarbeiter von Guerrilla Games[10], bei der GDC 2018 zu sehen ist, gehört zu diesen Tricks zum Beispiel das Anpassen der Vertices des Modells zur Laufzeit, in Abhängigkeit von dem Winkel zum Sichtfeld des Spielers (19:15-19:50). Auch das Verwenden eines angepassten Vorgehens beim Shading der Vegetation, um die Performance in Hinsicht auf Alpha-Texturen zu verbessern wird hier erwähnt (20:47-28:14). Ein weiterer Trick, der von Gilbert Sanders aufgelistet wird, ist das geschickte Anpassen der Normalen des Modells, um die Beleuchtung glaubwürdiger erscheinen zu lassen (29:38-31:44). Ob und wann diese Tricks angewandt werden können, variiert stark von Projekt zu Projekt und hängt von anderen Eigenschaften und Zielen des jeweiligen Projekts ab, daher wird im Folgenden nur tiefer auf das grundsätzliche Modellieren von Vegetationsmodellen eingegangen.

Ein, vielleicht naiver, Ansatz für das Modellieren von Pflanzenmodellen in Spielen, ist es, die gesamte Pflanze und alle ihrer Details, komplett drei-dimensional zu modellieren. Bei „einfachen“ Modellen wie etwa toten Bäumen und Blüten, kann das Ganze eine hervorragende Lösung sein, da man hiermit die Illusion des Spielers in der Spielwelt, auch aus nächster Nähe, aufrechterhalten kann. Sobald man aber versucht, Bäume und Büsche mit ihrem Blätterdach und dichtes Gras zu modellieren, fällt einem schnell auf, dass das durch diese Methode für ein Videospiel nicht machbar ist. Zum einen ist der Aufwand, um alle Details einzufangen, enorm groß und zum anderen, geht durch solche Modelle die Bildrate des Spieles stark herunter, da die Vertice-Anzahl sehr schnell riesig wird. Bei Videospielen wird daher auf das Nutzen von Ebenen, die mit einer transparenten Textur versehen werden, zurückgegriffen. Diese Ebenen werden bei der Modellierung, von den Entwicklern, an verschiedenen Stellen des Modells, teils sich überschneidend und mit verschiedenen Rotationen platziert, um die Illusion einer Pflanze mit Volumen zu erschaffen.

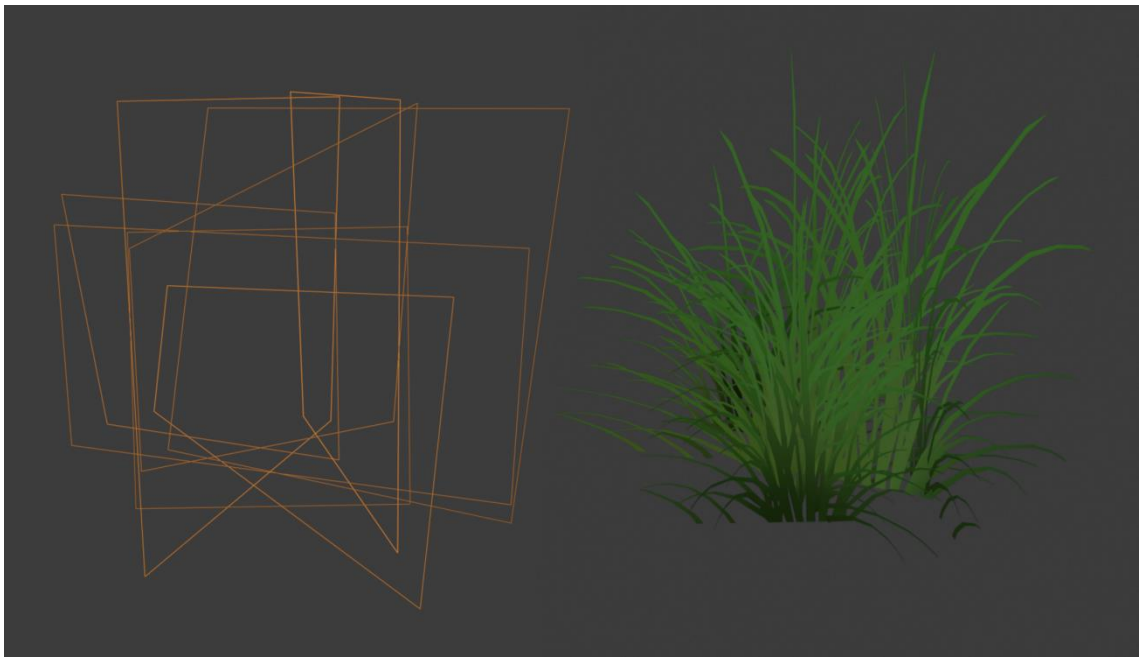

ABBILDUNG 5: EIN BEISPIEL FÜR DAS MODELLIEREN VON VEGETATION DURCH EBENEN. RECHTS IST EINES DER MODELLE FÜR GRAS IN DER ANWENDUNG ZU SEHEN UND LINKS DER AUFBAU DIESES MODELLS DURCH EBENEN.

[9] Präsentation von Gilbert Sanders auf dem Youtube-Kanal der GDC veröffentlicht am 06.02.2020 - https://www.youtube.com/watch?v=wavnKZNSYqU

[10] Guerrilla Games - https://www.guerrilla-games.com

Durch den Einsatz von Ebenen mit einer transparenten Textur, kann die Vertice-Anzahl und damit der allgemeine Aufwand für das Rendern, stark reduziert werden. Das drei-dimensionale Ausmodellieren einer Pflanze und das Nutzen von Ebenen mit transparenter Textur, kann, je nach Pflanzenmodell, natürlich auch kombiniert werden. So kann zum Beispiel der Stamm eines Baumes drei-dimensional modelliert werden, da dieser Teil des Baumes sich nur sehr schwer und unglaubwürdig in zwei Dimensionen darstellen lässt, und die Blätter werden mittels Ebenen mit transparenten Texturen modelliert. Ein Problem, das bei dem Modellieren von Vegetation mit Ebenen auftreten kann, ist inkorrekte Beleuchtung aufgrund des Überschneidens von Ebenen untereinander und das Verdecken von Ebenen durch andere Ebenen. Durch dieses Problem können unnatürlich wirkende Schatten entstehen, die die Illusion einer Pflanze mit Volumen zerstören und den Aufbau durch mehrere einzelne Ebenen sichtbar machen. Um das Problem zu lösen müssen die Normalen aller Ebenen, manuell oder automatisch, angepasst werden, um das Entstehen dieser Schatten zu vermeiden und die Beleuchtung zu verbessern. Dadurch wird zwar in den meisten Fällen trotzdem keine korrekte Beleuchtung erreicht, jedoch ist das Endergebnis in vielen Fällen glaubhafter für den Spieler.

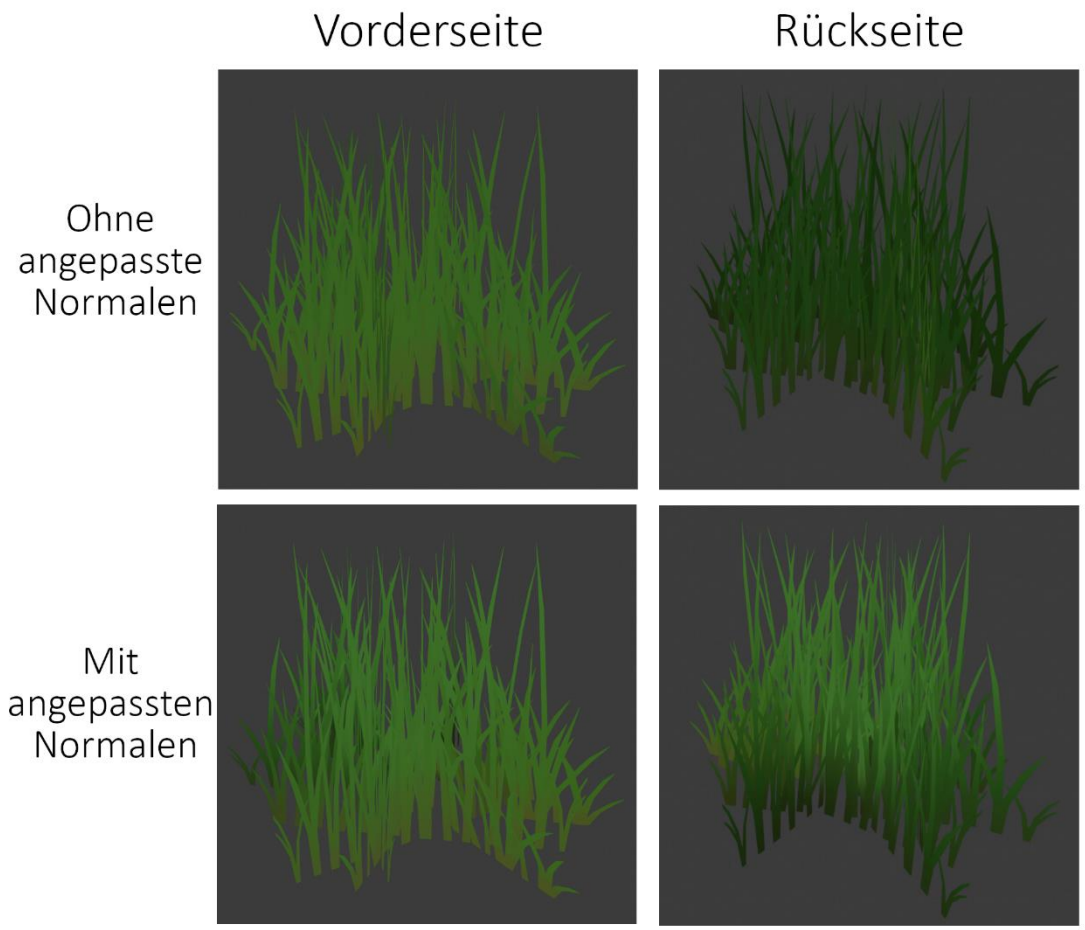

ABBILDUNG 6: EIN VERGLEICH DER BELEUCHTUNG ANHAND EINES IN DER ANWENDUNG VERWENDETEN MODELLS FÜR GRAS.

4 ANWENDUNGSDESIGN

Um die Ziele dieser Arbeit zu erfüllen und eine qualitativ hochwertige AR-Anwendung zu erstellen, die dem Endnutzer ein gutes Erlebnis bietet und die verschiedenen wichtigen Aspekte der prozeduralen Generation und Vegetationserstellung in 3D-Anwendungen und Videospielen erfüllt, werden im folgenden Abschnitt Anforderungen an die Anwendung gestellt, die den Designprozess beeinflussen und während der Entwicklung in Betracht gezogen werden müssen.

4.1 ANFORDERUNGEN FÜR DIE ALLGEMEINE ANWENDUNG

Die Performance stellt einen wichtigen Teil des Entwickelns für Augmented Reality dar und vor allem für das Entwickeln von Anwendungen für kabellose Head-Mounted Display, wie etwa die HoloLens 2, ist eine gute Optimierung der Anwendung besonders wichtig, um eine hohe Bildrate zu erhalten und dadurch eine gute Spielerfahrung zu erschaffen.

Dem Endnutzer sollte während des Spielens der größtmögliche Komfort gegeben werden. Zum einen dient das dazu, den Spieler positiv auf die Anwendung einzustimmen, und zum anderen soll damit verhindert werden, dass er das Spiel nach kurzer Zeit abbricht, da er das Spielerlebnis als unangenehm empfindet. Die Performance bzw. die Bildwiederholrate ist hier auch allgemein bei Augmented Reality-Anwendungen insbesondere zu beachten. Eine niedrige Bildanzahl pro Sekunde kann bei den Benutzern der Anwendung zu starkem Unwohlsein und in vielen Fällen auch Übelkeit führen. Das Ganze wird bei Augmented Reality und den verwandten Gebieten Virtual- und Mixed-Reality oft als Simulation Sickness bezeichnet, jedoch werden hierfür auch andere Begriffe, wie z.B. Motion Sickness, verwendet.

Da die Anwendung auf sehr spezifischer Hardware läuft, müssen einige Aspekte beachtet werden. Ein Aspekt ist dabei, dass der Nutzer bei der HoloLens 2 die Hologramme sehen kann, weil diese auf einer speziellen Art von Display, mithilfe von farbigem Licht, projiziert werden. Das Ganze funktioniert dabei nur additiv, dass heißt, dem Licht der Umgebung des Nutzers wird Licht auf dem Bildschirm hinzugefügt, um einzelne Hologramme darzustellen. Da das Projizieren von Hologrammen durch das additive Hinzufügen von Licht in das Blickfeld des Spielers geschieht, können sehr dunkle Farben nur schlecht oder gar nicht direkt dargestellt werden. Zum Beispiel würde ein Hologramm, für das als Farbe ein tiefes Schwarz gewählt wird, laut der offiziellen Dokumentation von Microsoft[11], vom Nutzer als transparent war genommen werden. (Microsoft, 2018).

Der Einsatz von externen Software Development Kits (SDKs) und verschiedenen Code-Bibliotheken kann das Entwickeln stark vereinfachen und das Implementieren einzelner Funktion beschleunigen. Besonders beim Entwickeln für die HoloLens und HoloLens 2 kann man hier, durch den Einsatz von externen Ressourcen, viele Abkürzungen nehmen, die das Entwickeln einer Anwendung erleichtern. Solche SDKs bieten Entwicklern oft verschiedene Algorithmen an, mit denen das Spatial Mapping-Mesh analysiert und weiter verarbeitet werden kann, um zum Beispiel verschiedene Objekte wie Wände und Möbel zu identifizieren und eventuelle Löcher im erhaltenen Mesh zu schließen und es damit „wasserdicht“ zu machen. Das Identifizieren von Objekten ist, je nach Art der Anwendung, besonders wichtig für das prozedurale Platzieren von Spieleinhalten und ein wasserdichtes Mesh sorgt dafür, dass Raycasts im Normalfall nicht außerhalb der Spielszene geschossen werden können. Der große Nachteil von externen Ressourcen ist aber, dass die Anwendung durch das Benutzen von SDKs und Code-Bibliotheken von diesen abhängig wird und nicht immer garantiert werden kann, dass

[11] Microsoft: What is a hologram? - https://docs.microsoft.com/en-us/windows/mixed-reality/discover/hologram

solche externen Ressourcen auch in der Zukunft noch Unterstützung und Aktualisierungen erhalten. Wenn die Anwendung in der Zukunft eventuell erweitert werden soll, kann das ein großes Problem darstellen.

Zusammengefasst ergeben sich hieraus die nachfolgenden Anforderungen an das Anwendungs-Design, die für den Designprozess und während der Entwicklung so gut wie möglich beachtet werden sollten:

- Die Anwendung muss über eine angemessene Performance verfügen, um eine gute allgemeine Spielerfahrung für den Spieler zu gewährleisten.
- Die Spielprinzipien und das Anwendungsdesign sollten so gewählt sein, dass der Spieler den größtmöglichen Komfort während des Spielerlebnisses hat.
- Die Anwendung und ihre Komponenten sollten gut an die verwendete Hardware angepasst sein.
- Während der Entwicklung sollte es so weit wie möglich vermieden werden, externe SDKs und Code-Bibliotheken zu verwenden, um zukünftige Kompatibilität zu gewährleisten und das problemlose Erweitern der Anwendung zu ermöglichen.

4.2 DESIGN DER ALLGEMEINEN ANWENDUNG

Aufgrund der definierten allgemeinen Anforderungen an die Anwendung als Augmented Reality-Anwendung und um eine unterhaltsame Anwendung für den Benutzer zu erschaffen, wurde bei der Entwicklung auf einige grundsätzliche Designaspekte geachtet.

Wie in der Arbeit von Sasha Azad et al. angeschnitten wird, ist es unwahrscheinlich das sich jeder Bereich eines Raumes für das Spielen einer Augmented Reality-Anwendung eignet. (Azad et al., 2016, S. 248). Ebenso ist es unwahrscheinlich, dass jeder Raum genau so gut wie ein anderer Raum für das Anwenden eines spezifischen Spielaspektes eines Augmented Reality-Spieles geeignet ist. Die Anwendung dieser Arbeit und die Komponenten der Anwendung sind daher gezielt für die Verwendung in einem durchschnittlichen Wohnungsraum konzipiert worden. Grundsätzlich kann die Anwendung zwar auch in anderen Arten von Räumen verwendet werden, aber dabei ist es wahrscheinlicher, dass es zu vermehrten Problemen in der Anwendung, vor allem bei der prozeduralen Erstellung der Kletterpflanzen und dem prozeduralen Platzieren aller Pflanzenarten, kommt.

Da die HoloLens 2 ein kabelloses HMD ist und daher kein Stromkabel besitzt, sondern einen Akku benutzt, der bei der Nutzung von rechenintensiven Anwendungen, im Vergleich schnell leer geht, bietet es sich an, den Spielzyklus relativ kurz zu halten. Damit kann sichergestellt werden, dass der Spieler das Spiel nicht mitten im Spielverlauf abbrechen muss, um das Gerät aufzuladen.

Die HoloLens 2 hat zudem den Nachteil, dass der Sichtbereich auf dem Bildschirm sehr klein, im Vergleich zu herkömmlichen Monitoren, ist. Das Field-Of-View der HoloLens 2 ist zwar größer als bei der ersten HoloLens, jedoch ergeben sich daraus immer noch einige Limitationen, vor allem in Kombination mit der Art und Weise, wie die Hologramme auf dem Bildschirm gerendert werden. Eine solche Limitation ist zum Beispiel, dass das Nutzen von soliden Farben, die das gesamte Blickfeld des Bildschirms bedecken, den Spieler leicht desorientieren kann und das richtige Interagieren mit der Anwendung erschwert. Aus diesem Grund wurde das Benutzen von Objekten, die einen Großteil des Bildschirms einnehmen und eine einzige, solide Farbe besitzen, in dieser Anwendung vermieden.

Ein weiteres Problem, das sich daraus ergibt, dass die HoloLens 2 kein kabelgebundenes HMD ist, ist die eingeschränkte Rechenleistung. Dazu kommt die Wichtigkeit einer hohen Bildwiederholrate, um bei Spielern die schon erwähnte Simulation Sickness zu vermeiden. Für die Anwendung bedeutet das, dass eine gute Balance zwischen der Performance und dem Detailgrad der Vegetation im Spiel und der Rechenintensivität des Codes gefunden werden muss. Die im Spiel benutzten 3D-Modelle der einzelnen Pflanzen wurden daher sehr simpel gehalten und besitzen vergleichsweise wenige Vertices.

Die Anwendung sollte zudem auch einen spielerischen Aspekt haben, um den Ablauf interessanter zu gestalten und dem Nutzer eine Interaktionsmöglichkeit zu verschaffen. Hiermit wird dem Spieler auch etwas mehr Kontrolle über das Wachstum der Pflanzen gegeben. Da das Benutzen von Augmented Reality-Anwendungen heutzutage noch nicht sehr weit verbreitet ist und die meisten Personen keine Erfahrung im Umgang mit AR-HMDs, wie etwa der HoloLens 2, haben, ist die Steuerung der spielerischen Elemente in der Anwendung intuitiv und simpel gehalten.

Von Short und Adams werden zudem mehrere Tipps für das Platzieren und Anpassen von prozedural erstellten Inhalten gegeben, um glaubhaften Zufall in die prozedurale Generation zu implementieren. Diese werden als Designprinzipien für die prozedurale Generation in dieser Anwendung verwendet. Zum einen sollten prozedural Inhalte im Spiel nicht in regelmäßigen Abständen auftauchen (Short & Adams, 2017, S. 19), dadurch ist es leicht für den Spieler ein Muster zu erkennen und die Illusion des Zufalls geht verloren. Genauso hilft es, wenn die prozeduralen Inhalte mit einer zufälligen Skalierung und Variationen im Aussehen versehen werden. (Short & Adams, 2017, S. 20). Durch das Anwenden dieser Tipps kann das Erkennen von wiederverwendeten, gleichen Objekten, für den Spieler sehr schwer werden, solange er nicht aktiv darauf achtet und versucht die einzelnen Objekte zu analysieren.

4.3 DESIGN DER EINZELNEN KOMPONENTEN

Für die einzelnen Komponenten der Anwendung müssen auch einige Design-Grundlagen festgelegt werden, damit diese den richtigen Effekt haben und die vorgesehenen Funktionen erfüllen.

Alle Pflanzen sollten während der Laufzeit in der Spielwelt entweder komplett prozedural erstellt oder zumindest prozedural platziert werden. Bei der Auswahl der Vegetation muss dabei beachtet werden, dass die Pflanzen vertikale und horizontale Flächen großflächig abdecken und eine dichte Überwucherung der Umgebung darstellen können, ohne dafür viele komplizierte 3D-Modelle zu benötigen. Zudem sollten die einzelnen Pflanzen thematisch übereinstimmen, um die gesamte Spielszene glaubwürdiger wirken zu lassen. Das bedeutet, dass zum Beispiel keine Pflanzen, die nur in kälteren Regionen wachsen, mit Pflanzen, die nur in tropischen Bereichen vorkommen, vermischt werden. Hierfür ist es wichtig, dass die Pflanzen der Anwendung an reale Vegetation angelehnt sind und daher leicht von den Spielern und auch bei der Entwicklung zugeordnet werden können.

Grundsätzlich ist es für das Platzieren eine gute Idee, die generierte Vegetation in Pflanzen, die auf vertikalen Flächen wachsen und Pflanzen, die auf horizontalen Flächen wachsen, einzuteilen. Die Art und Weise, wie die verschiedenen Pflanzenarten platziert werden, kann dadurch leichter an die beiden Kategorien und ihre Umgebung angepasst werden.

Für die Pflanzenarten, die an vertikalen Flächen platziert werden, bieten sich hier vor allem Kletterpflanzen an. Diese können in der Anwendung genutzt werden, um Wände und Objekte in der Umgebung des Spielers leicht großflächig zu überwachsen, ohne dafür viele einzelne, komplizierte Modelle generieren zu müssen. Wie der Name schon aussagt, sind Kletterpflanzen

in der Regel nicht selbst stützend und benötigen ein externes Gebilde, an dem sie für ihr Wachstum emporklettern können. Dadurch eignen sie sich hervorragend, um das genaue Abtasten des Raumes durch Spatial Mapping gut auszunutzen, da sie sich an die Formen des Raumes anpassen und dabei eine große Fläche abdecken können. Kletterpflanzen können dabei in der Theorie auch von den vertikalen Flächen auf horizontale Flächen übergehen und daher gegebenenfalls die horizontalen Oberflächen überwuchern, auf denen (noch) keine anderen Pflanzenarten wachsen.

Für das Überwuchern von horizontalen Flächen kann man theoretisch viele verschiedene Pflanzensorten nutzen, jedoch bietet sich hier vor allem das Benutzen verschiedener Modelle für Gras und Pflanzen, die von oben gesehen viel Fläche bedecken, an, weil diese den Untergrund gut abdecken können. Durch das Nutzen mehrerer unterschiedlicher 3D-Modelle für Gras und das richtige aussuchen verschiedener Gräser, Unkrautsorten und anderer Pflanzen, die auf dem Boden zu finden sind, kann man die Farbe der Vegetation leicht variieren und damit das schon erwähnte Problem des vollständigen Bedeckens des Blickbereichs durch eine solide Farbe vermeiden.

Für die Implementation einer prozedural generierten Kletterpflanze in einer Augmented Reality-Umgebung, müssen auch einige Designanforderungen erfüllt werden. Zum einen ist es wichtig, dass sich die einzelnen Kletterpflanzeninstanzen einen eigenen Pfad durch die AR-Umgebung suchen. Hierfür ist das Implementieren eines Algorithmus, der die Umgebung mit einem ausreichenden Detailgrad abtasten und analysieren kann und dadurch einen Pfad für das Wachstum der Kletterpflanzen findet, unabdingbar.

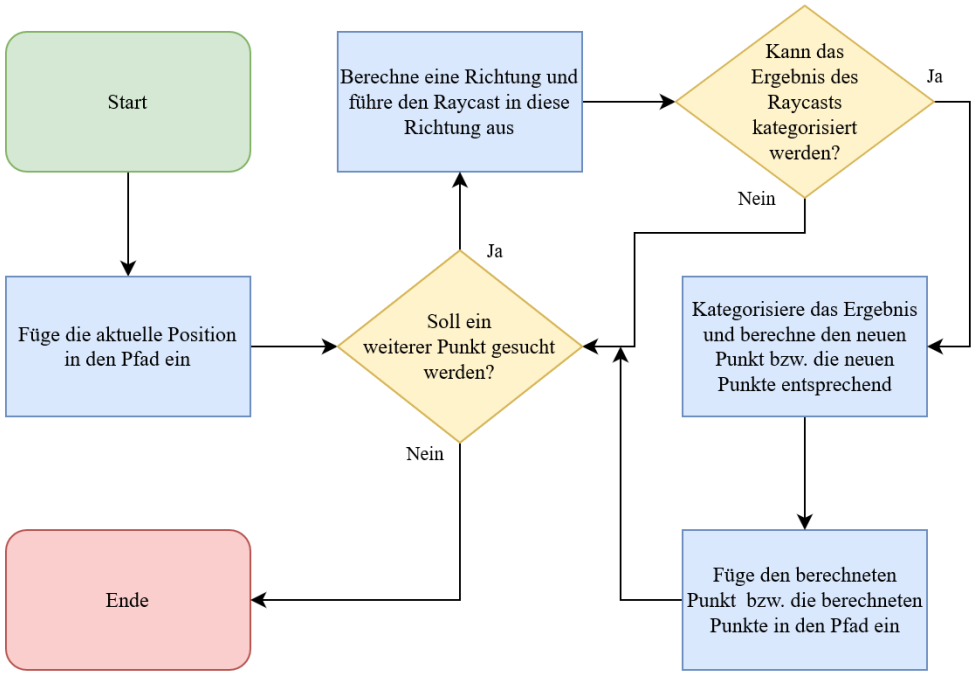

ABBILDUNG 7: EIN FLUSSDIAGRAMM, WELCHES DEN GRUNDSÄTZLICHEN ABLAUF DES BESCHRIEBENEN PFADFINDUNGS-ALGORITHMUS DARSTELLT.

Grundsätzlich muss solch ein Algorithmus erkennen können, was für eine Art von Hindernis sich direkt vor der Kletterpflanze befindet und den Pfad dementsprechend anpassen. Für das Abtasten einer AR-Umgebung, die durch das Spatial Mapping der HoloLens 2 erstellt wurde, können die Hindernisse, die der Algorithmus unterscheiden muss, zum Beispiel je nach Winkel

der Oberfläche im Vergleich zur aktuellen Richtung des Pfades oder durch das Ausführen mehrere Raycasts und das Auswerten dieser, in verschiedene Kategorien eingeteilt werden. Der Algorithmus kann dann, je nach Art des Hindernisses, den nächsten Punkt des Pfades entsprechend anpassen, um das Entlangklettern der Kletterpflanze an der Oberfläche der Umgebung zu simulieren. In Abbildung 7 ist ein Flussdiagramm für die allgemeine Implementierung solch eines Pfadfindungs-Algorithmus mithilfe von Raycasts dargestellt.

Sobald ein Pfad gefunden wurde, kann das Mesh der Kletterpflanze entlang dieses erstellt werden. Je nach Detailgrad des Pfades, kann man hierfür einzelne Vertices in einem Umkreis (dem gewünschten Radius des Kletterpflanzenstammes) an jedem Punkt des Pfades erstellen und durch Verbinden der individuellen Punkte zu gemeinsamen Polygonen ein Mesh erstellen. Falls der Pfad nur sehr wenige Punkte besitzt, kann es nötig sein, diesen zuvor durch verschiedene Methoden in eine detailreichere Version umzuwandeln (z.B. durch Nutzen der existieren Punkte als Kontrollpunkte für Bézier-Kurven oder Catmull-Rom Splines), um eine glaubhaft aussehende Kletterpflanze zu generieren. Das Ganze ist allerdings stark davon abhängig, auf welche Art und Weise der Pfad der Kletterpflanze erstellt wurde und welche Designidee dahintersteckt.

Je nach gewählter Implementierungsart für die Kletterpflanzen, ist es auch eine gute Idee, den Beginn der Kletterpflanze durch andere Vegetation zu verdecken oder auf andere Weise anzupassen. Wenn der Start des Stammes einer Kletterpflanze aus dem Nichts erscheint, sieht das für den Benutzer komisch aus und die Glaubwürdigkeit der Vegetation geht verloren.

In den Algorithmus für das Pfadfinden können auch weitere, unterschiedliche Verhaltensweisen von Kletterpflanzen eingebaut werden. Da Kletterpflanzen, wie die meisten Pflanzen, versuchen, ihre Lichtaufnahme zu maximieren, und das dazu führt, dass sie im Normalfall generell nach oben streben, kann man in den Algorithmus eine Tendenz für die entsprechende Richtung einbauen, falls sich die Pflanze an einer vertikalen Fläche befindet. Ein weiteres Verhalten von Kletterpflanzen, dass gerade bei der Pfadfindung mit Raycasts leicht zu implementieren ist, ist das Finden von Gebilden, an denen sie empor klettern können, ohne dabei auf den Zufall zu vertrauen. Manche Kletterpflanzenarten haben spezielle Mechanismen, die sie zum Finden von kletterbaren Strukturen nutzen. Ein gutes Beispiel hierfür ist das Wachsen in Richtung schattiger Bereiche, auch Skototropismus genannt, das bei verschiedenen Arten von Kletterpflanzen auftreten kann. (Gianoli, 2015, S. 4). Das richtige Definieren solcher schattigen Bereiche und finden dieser in der Spielwelt, insbesondere bei prozeduraler Generation des Pfades in einer Umgebung, die erst zur Laufzeit konkret bekannt ist, kann jedoch sehr aufwendig sein und den Algorithmus kompliziert machen. Da die botanische Exaktheit bei den Kletterpflanzen in dieser Arbeit nicht im Vordergrund steht, ist es eine gute Idee, derartige Verhaltensweisen in dem Pfadfindungsalgorithmus zu approximieren. Bei einer guten Approximation kann der Spieler den Unterschied nur schwer feststellen und so kann hier der Leistungsbedarf der Anwendung eingegrenzt werden.

Für das prozedurale Platzieren von Pflanzen auf den unterschiedlichen Flächen des Spatial Mapping-Meshes wird auch ein eigenes System bzw. ein eigener Algorithmus benötigt. Wichtig bei diesem ist es, die einzelnen Pflanzen nur dort zu platzieren, wo es auch Sinn macht. Zum Beispiel sollte Vegetation auf denen für sie geeigneten Oberflächen nur wachsen können, wenn dort genug Platz zur Verfügung steht. Für die Prüfung, ob an einer ausgesuchten Stelle genug Platz zur Verfügung steht, muss dabei bei Pflanzenmodellen, bei denen das Überlappen mit anderen Modellen (Clipping) möglichst vermieden werden soll, nicht nur auf die allgemeine Oberfläche getestet werden, sondern auch auf die schon existierenden, umliegenden Pflanzen. Falls an der gewünschten Stelle zwar Platz auf der Oberfläche ist, aber sich eine andere, bereits in der Spielwelt platzierte Pflanze, zu nahe befindet, muss entweder ein neuer Punkt für die Platzierung gefunden oder abgebrochen werden, um das Überschneiden der Modelle zu

vermeiden. Insbesondere bei den Pflanzen die auf horizontalen Flächen, wie etwa dem Boden, platziert werden sollen, muss zudem definiert werden, welche Flächen auch wirklich zulässig sind. Ein Kriterium, um die zulässigen Bereiche auf der entsprechenden Oberfläche zu definieren, ist zum Beispiel ein gewisser Abstand zu anderen Umgebungsstrukturen in der Spielszene, wie etwa den Wänden oder Möbeln im Raum. Das Einhalten dieses Abstandes dient dabei hauptsächlich dazu, um hier auch das Problem des Clippings der Pflanzenmodelle, in diesem Fall mit dem Spatial Mapping-Mesh, weitestgehend zu vermeiden. Ein weiteres Kriterium ist der Winkel zum Up-Vektor der Szene. Für viele Arten von Bodenpflanzen macht es nur Sinn, auf einem Untergrund bis zu einer maximalen Steigung der Oberfläche zu wachsen. Um einen zulässigen Bereich für das Platzieren der Bodenpflanzen zu definieren, muss das Spatial Mapping-Mesh der Umgebung daher anhand dieser Kriterien, entweder während der Laufzeit oder vor Spielbeginn, analysiert werden. Abbildung 8 stellt den Ablauf für das Platzieren von Pflanzen durch einen Algorithmus dieser Art dar.

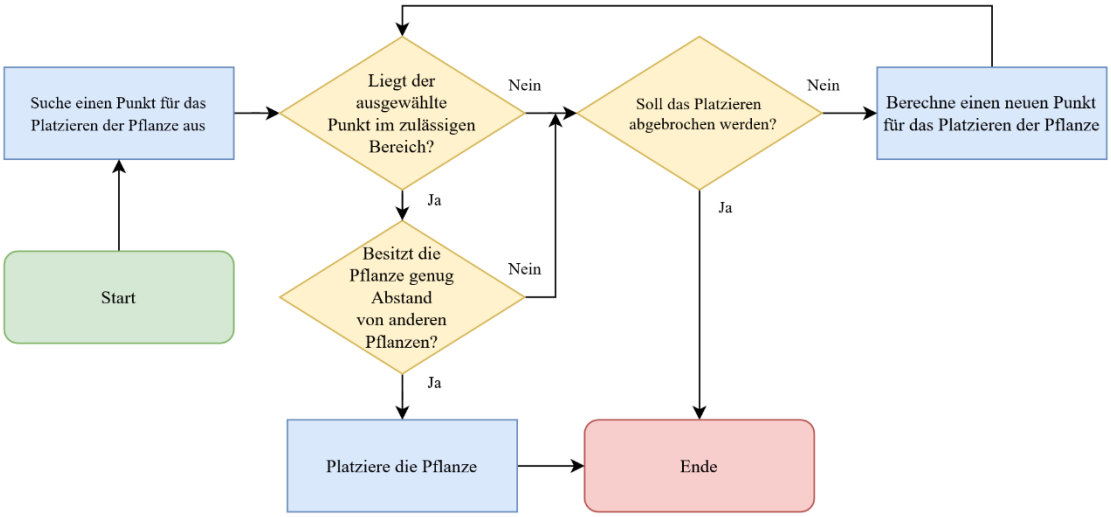

ABBILDUNG 8: EIN FLUSSDIAGRAMM, WELCHES DEN ABLAUF DES BESCHRIEBENEN ALGORITHMUS FÜR DAS PLATZIEREN VON PFLANZEN DARSTELLT.

Damit die Anwendung glaubwürdig wirkt und den Spieler die Illusion eines überwachsenen Raumes gibt, dürfen die Pflanzen nicht einfach aus dem Nichts erscheinen. Hierfür muss für jede einzelne Pflanzenart eine Wachstumsanimation implementiert werden. Diese Wachstumsanimation sollte idealerweise direkt abgespielt werden, wenn die Pflanze platziert wird. Um eine glaubhafte Animation für das Wachstum der Vegetation zu erstellen, reicht es oft schon aus, das Pflanzenmodell einfach entlang seiner Achsen zu skalieren. Bei komplizierteren 3D-Modellen kann sich auch das Nutzen von Blend-Shapes oder detailreichen Animation, die beim Erstellen des jeweiligen Modells angefertigt werden, als notwendig erweisen. Unabhängig davon, wie und welche Art der Animation genutzt wird, muss man darauf achten, dass diese angemessen für die Pflanzenart und deren Nutzen in der Anwendung sind. Eine aufwendige Animation, durch Blend-Shapes oder kompliziertes animieren in der Modellierungs-Software, für simple Vegetationsmodelle, wie etwa einzelne Blätter und Gras, kann einen sehr starken, negativen Effekt auf die Performance des Spiels haben. Solche Modelle kommen normalerweise oft in der Szene vor und dementsprechend werden viele einzelne Kopien bzw. Instanzen davon gebraucht, für die die Animation jeweils separat ausgeführt werden muss.

5 IMPLEMENTIERUNG DER ANWENDUNG

Im folgenden Kapitel wird die Implementierung der Anwendung näher erläutert, hierbei wurde so gut wie möglich versucht, sich an das festgelegte Anwendungsdesign zu halten und die aufgeführten Anforderungen zu erfüllen. Wie schon erwähnt, gleichen sich die Begriffe „Klasse“, „Skript“ und „Komponente“ in Unity sehr stark, daher werden diese in diesem Abschnitt als synonym angesehen. Um einen kurzen Überblick über die wichtigsten Komponenten der implementierten Anwendung zu geben, werden diese an dieser Stelle kurz aufgelistet. Der allgemeine Ablauf der Anwendung wird durch die Komponenten Game-Manager und Game-Settings gehandhabt. Der Object-Placer, das Ivy-Skript und die Pathfinding-Komponente sind an der prozeduralen Generation von Spieleinhalten beteiligt. Ein Großteil der Optimierung für die Anwendung ist in den Object-Pooling, Mesh-Square-Manager und Mesh-Combiner Klassen zu finden, auf die in Kapitel 6 näher eingegangen wird.

5.1 ABLAUF DER ANWENDUNG

Das Erste, dass der Benutzer sieht, sobald er die Anwendung startet, ist eine Informationstafel. Diese Informationstafel gibt dem Benutzer eine kurze Anleitung, die ihn durch die einzelnen Zustände, in denen sich die Anwendung befinden kann, lotsen soll. Diese Zustände können wie folgt kurz zusammengefasst werden:

1. Scannen des Raumes
2. Generieren und überprüfen des zulässigen Bereichs
3. Spielen/Platzieren der Pflanzen im Raum

Um von einem Zustand in den nächsten Zustand zu kommen, benutzt der Spieler die „Air-Tap“ Geste der HoloLens 2. So kann der Spieler, sobald er mit der Abtastung des Raumes zufrieden ist, einfach zum zweiten Zustand wechseln. Im zweiten Zustand sieht der Benutzer den Bereich als blau markiert, der von der Anwendung als zulässiger Bereich für das Platzieren der Pflanzen auf horizontalen Flächen eingestuft wurde. Mit einem weiteren ausführen der „Air-Tap“ Geste kann der Spieler von dem zweiten in den letzten Zustand wechseln.

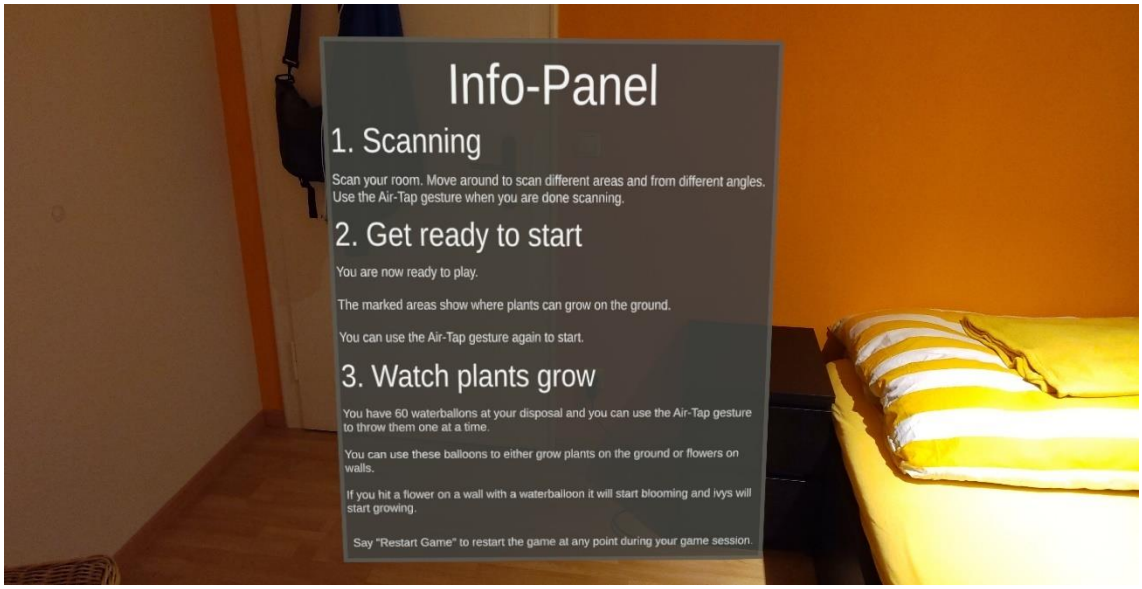

ABBILDUNG 9: DIE INFORMATIONSTAFEL, DIE DER SPIELER ZU BEGINN DER ANWENDUNG SIEHT.

Wenn der Spieler vom zweiten in den dritten Zustand wechselt, verschwindet die Informationstafel und Vegetation fängt langsam an, von allen Richtungen auf die Position des Spielers hinzu zu wachsen. Zur gleichen Zeit wachsen an zufälligen vertikalen Flächen Blumenblüten. Mithilfe der „Air-Tap“ Geste kann der Spieler nun bis zu 60 Wasserballons werfen, die durch den Raum fliegen und an den getroffenen Stellen das Wachstum weiterer Vegetation verursachen. Falls ein Wasserballon auf einer horizontalen Fläche landet, wachsen dort verschiedene Bodenpflanzen. Wenn ein Wasserballon eine vertikale Fläche (wie z.B. eine Wand) trifft, fängt eine Blumenknospe an dort zu wachsen, aber falls sich an dieser getroffenen Stelle bereits eine Blumenknospe befinden sollte, fängt diese an zu blühen und Efeu beginnt von der Blume aus in zufällige Richtungen zu wachsen. Sobald alle Wasserballons geworfen wurden, wird dem Spieler eine neue Tafel angezeigt, auf der die erreichte Punktezahl und der Highscore der aktuellen Spielsitzung zu sehen sind. An dieser Stelle kann der Spieler die Spracherkennung der HoloLens 2 nutzen und durch die Worte „Restart Game“ das Spiel neustarten.

ABBILDUNG 10: DIE PUNKTETAFEL AM ENDE DES SPIELES.

5.2 ALLGEMEINE KOMPONENTEN

Die Skripte Game-Manager und Game-Settings sind, wie die Namen schon vermuten lassen, jeweils für das Steuern des Anwendungsverlaufs und das Verwalten der Einstellungen von verschiedenen Komponenten, die in der Anwendung verwendet werden, verantwortlich. Diese beiden Skripte und alle anderen Skripte und weitere Komponenten, die für deren Aufgabe benötigt werden, sind einem einzelnen Game-Object in der Szene angehangen, welches als Prefab (ein vorgefertigtes Spielobjekt mit Komponenten, dass in der Szene platziert werden kann) abgespeichert wird und damit den Zugriff auf einzelne Einstellungen und den Szenenwechsel erleichtert.

Der Game-Manager verfügt über eine öffentliche statische Referenz auf eine Instanz von sich selbst, welche in der Awake-Methode gesetzt wird, falls sie noch nicht festgelegt wurde. Zudem werden öffentliche Referenzen auf andere Komponenten, wie zum Beispiel die Game-Settings

oder den Object-Placer, welche auch an dem Game-Object hängen, gespeichert. Das führt dazu, dass jedes Objekt innerhalb der Szene auf die, im Game-Manager gespeicherten, Referenzen zugreifen kann, was das Finden dieser Komponenten in anderen Skripten stark erleichtert. Des Weiteren verfügt der Game-Manager auch über Referenzen auf die Prefabs der Informationstafel und der Punktetafel, sowie eine Referenz auf den aktuellen Spatial-Awareness-Mesh-Observer, der durch das Mixed Reality Tool Kit zur Verfügung gestellt wird. Der Spatial-Awareness-Mesh-Observer ist für das Analysieren des Raumes und Erstellen des Spatial Mapping-Meshes verantwortlich. Wie schon erwähnt, ist die Aufgabe des Game-Managers, den Anwendungsverlauf zu steuern und Komponenten zu koordinieren. Hierfür wird ein Enum verwendet, dass die einzelnen Zustände, in denen sich die Anwendung momentan befinden kann, darstellt. Der Zustand kann hiermit zwischen *IsScanning*, *HasFinishedScanning, IsPlaying* und *HasFinishedPlaying* gewechselt werden.

Der Input-Manager des Game-Objects ist für das Erkennen jeglicher Eingaben verantwortlich. Sobald eine Eingabe erkannt wird, ruft dieses Skript die InputRegistered-Methode der aktuellen Game-Manager-Instanz auf und übergibt einen Richtungsvektor. Für das Testen der Anwendung an einem Desktop-Computer berechnet sich dieser Richtungsvektor als Vektor von der Szenen-Kamera zur Position des Maus-Cursors auf dem Bildschirm. Falls die AR-Brille benutzt wird, ist dieser Richtungsvektor der Forward-Vektor der Augmented Reality-Kamera. Alternativ hätte man hier auch den Vektor von der AR-Kamera zur Hand des Spielers nehmen können, jedoch kann damit das Zielen für den Spieler, aufgrund des relativ eingeschränkten Field-of-Views der HoloLens 2, erschwert werden. Die InputRegistered-Funktion ist die Kernfunktion des Game-Managers, da dort zwischen den einzelnen Anwendungszuständen gewechselt wird und die einzelnen Komponenten des Game-Objects koordiniert werden. Je nachdem, in welchem Zustand sich die Anwendung momentan befindet, führt die InputRegistered-Funktion beim Aufruf unterschiedlichen Code aus, bevor in den jeweils nachfolgenden Zustand gewechselt wird, insofern der letzte Zustand noch nicht erreicht wurde.

```
if (mCurrentGameState == GameStates.IsScanning) //Erster Zustand
{
    //Aussetzen des Observers, um das Scannen zu beenden
    //und Unsichtbar machen des Spatial Mapping Meshes
    if (mObserver != null)
    {
        mObserver.Suspend();
        mObserver.DisplayOption = SpatialAwarenessMeshDisplayOptions.None;
    }
    //Erstellen der Meshes der zulässigen Bereiche für das Platzieren auf horizontalen Flächen
    mValidAreaManager.GenerateValidAreaMeshes();
    //Wechseln des Zustandes
    mCurrentGameState = GameStates.HasFinishedScanning;
}
```

ABBILDUNG 11: ABLAUF DES ERSTEN ZUSTANDES DER ANWENDUNG IN DER INPUTREGISTERED-FUNKTION.

Falls sich die Anwendung im Zustand *IsScanning* befindet, wird das Abtasten der Spielumgebung mithilfe des Spatial-Awareness-Mesh-Observer der Szene eingestellt. Der Grund hierfür ist, dass das Abtasten der Umgebung des Spielers durch den Observer ressourcenaufwendig ist und während des restlichen Anwendungsverlaufs nicht unbedingt benötigt wird. Zudem werden mithilfe des Valid-Area-Managers mehrere Meshes für das Finden von zulässigen Bereichen für das Platzieren von Bodenvegetation erstellt (Genaueres hierzu in Kapitel 5.3). Die Meshes werden zusätzlich im nächsten Zustand farblich hervorgehoben. Der Sinn dahinter ist es, dem Spieler einen grundlegenden Überblick zu geben, wo weitere Pflanzen platziert werden können. Das wird gemacht, da nicht unbedingt alle Oberflächen, die sich in der Theorie für das Platzieren

von Vegetation eignen, garantiert erkannt wurden. Des Weiteren wird das Spatial Mapping-Mesh, das durch den Observer erstellt wurde, unsichtbar gemacht, da der Spieler so die zulässigen Bereiche für die Vegetationsplatzierung besser erkennen kann.

```
else if (mCurrentGameState == GameStates.HasFinishedScanning) //Zweiter Zustand
{
    //Verdeckung von Objekten durch das Spatial Mapping Mesh aktivieren
    if (mObserver != null)
    {
        mObserver.DisplayOption = SpatialAwarenessMeshDisplayOptions.Occlusion;
    }
    //Verstecken der Informationstafel
    mInfoPanel.SetActive(false);
    //Unsichtbar machen des zulässigen Bereichs
    mValidAreaManager.RenderValidAreaMeshes(false);
    //Automatisches Platzieren der Vegetation an vertikalen und horizontalen Flächen
    OBJECTPLACER.PlacePlantsOnWalls(5);
    OBJECTPLACER.PlaceObjectsInRadius(new Vector3(0f, 0.2f, 0f), 5f, 8);
    //Wechsel des Zustandes
    mCurrentGameState = GameStates.IsPlaying;
}
```

ABBILDUNG 12: ABLAUF DES ZWEITEN ZUSTANDES DER ANWENDUNG IN DER INPUTREGISTERED-FUNKTION.

Befindet sich die Anwendung im Zustand *HasFinishedScanning*, wird zunächst die Sichtbarkeit des Spatial Mapping-Meshes auf die Option *Occlusion* gestellt. Die Sichtbarkeitseinstellung *Occlusion* bewirkt, dass Objekte, die in der Spielwelt platziert wurden, hinter dem Spatial Mapping-Mesh (ggf. auch nur teilweise) verschwinden, falls dieses das jeweilige Objekt verdeckt. Die Informationstafel und die Meshes, die die zulässigen Bereiche für das Platzieren von Pflanzen darstellen, werden anschließend auch ausgeblendet, um die Szene auf den weiteren Anwendungsablauf vorzubereiten. Bevor in den nächsten Zustand gewechselt wird, werden mehrere Funktionen des Object-Placers aufgerufen, die beginnen, Pflanzen an vertikalen Flächen und an zufälligen Punkten auf dem Spatial Mapping-Mesh zu platzieren, falls diese in zulässigen Bereichen liegen (Näheres hierzu in Kapitel 5.5). Der Spieler kann dadurch während des Spielens nicht nur besser nachvollziehen, wo Pflanzen auf horizontalen Flächen platziert werden können, da die zulässigen Bereiche nicht mehr farbig hervorgehoben sind, sondern die Atmosphäre eines Raumes, der langsam von der Natur erobert wird, wird dadurch auch direkt zum Spielbeginn verstärkt.

```
else if (mCurrentGameState == GameStates.IsPlaying) //Dritter Zustand
{
    if (numberOfWaterballoons > 0)
    {
        //Schießen eines Wasserballons in die erhaltene Richtung
        ShootBallon(direction);
    }
    else
    {
        //Zeige dem Spieler die aktualisierte Punktetafel und wechsel in den letzten Zustand
        ShowGameStats();
        mCurrentGameState = GameStates.HasFinishedPlaying;
    }
}
```

ABBILDUNG 13: ABLAUF DES DRITTEN ZUSTANDES DER ANWENDUNG IN DER INPUTREGISTERED-FUNKTION.

Wird eine Eingabe erkannt, wenn der aktuelle Zustand der Anwendung *IsPlaying* ist, wird lediglich ein Wasserballon, in Abhängigkeit von dem erhaltenen Richtungsvektor, geworfen. Danach findet nur ein Zustandsübergang statt, sobald alle Wasserballons vom Spieler geworfen wurden. Ist das der Fall, wird dem Spieler die Punktetafel angezeigt und der Zustand auf *HasFinishedPlaying* gesetzt, in welchem kein weiterer Code ausgeführt wird.

Die GameSettings-Komponente ist dafür da, eine Klasse anzubieten, die alle Einstellungen für die unterschiedlichen Komponenten der Szene verwaltet und den Zugriff auf diese an beliebiger Stelle einfach möglich macht. Die Klasse besitzt eine Liste vom Typen *Plant*, in welcher die einzelnen Pflanzenarten definiert werden, und Einstellungen für den Object-Placer und die ObjectPooling-Komponente, sowie alle Einstellungen für das prozedurale Generieren der Efeu-Pflanzen und der Blumen, von denen aus der Efeu das Wachsen beginnt, zu finden sind.

Zusätzlich ist auch eine spielerische Komponente implementiert worden, die dem Spieler eine aktive Aufgabe in der Anwendung verschafft. Wenn der Spieler den Pflanzen in der Spielszene nur beim Wachsen zuschauen kann, wird das sehr schnell langweilig und der Nutzer hat keinen Grund, sich weiter in die Anwendung zu vertiefen. Die Implementierung eines spielerischen Aspekts weckt dabei bei dem Benutzer nicht nur ein höheres Interesse an der Anwendung, sondern sorgt auch dafür, dass diesem mehr Kontrolle über das Wachstum der Pflanzen gegeben wird. Da der Raum, in dem die Anwendung verwendet wird, vor dem Abtasten durch die HoloLens 2 nicht bekannt ist, und sich jeder Raum in den Möbeln, verfügbaren Oberflächen, freien Bereichen und anderen Strukturen unterscheidet, kann durch das automatische Platzieren der Vegetation in der Anwendung mittels des Algorithmus nicht unbedingt garantiert werden, dass die Vegetation optimal platziert und verteilt wird. Um dieses Problem zu lösen, kann der Spieler mit der „Air-Tap“ Geste der HoloLens 2 einen Wasserballon in seine Blickrichtung werfen, welcher an der getroffenen Stelle, je nach Oberflächenart, versucht, Vegetation zu platzieren. Des Weiteren löst der Spieler damit das Wachstum des Efeus aus, falls er eine bereits existierende Blume trifft. Um den Wasserballon zu werfen, hat dieser eine Rigidbody-Komponente angehangen, die das Nutzen von Unity‘s Physik-Engine ermöglicht und dafür sorgt, dass der Wasserballon durch die Schwerkraft beeinflusst wird. Sobald der Input-Manager die Geste des Spielers erkennt und sich die Anwendung im letzten Spielzustand befindet, wird ein Wasserballon durch ein angegebenes Prefab instanziiert und in die Blickrichtung des Spielers geworfen, indem auf seine Rigidbody-Komponente ein Kraftimpuls von der Richtung des Spielers aus ausgewirkt wird. Da das Sichtfeld vergleichsmäßig klein ist und es beim Tragen eines HMDs unangenehm sein kann, steil nach oben zu schauen, wird der Wasserballon hierbei zusätzlich mit einem geringen Kraftimpuls nach oben geschossen, um das Treffen der gewünschten Stelle zu erleichtern. Der Wasserballon hat ein Skript, dass durch eine angehangene Physics-Collider-Komponente überprüft, ob mit dem Wasserballon etwas getroffen wurde. An der Stelle wo der Ballon auftrifft, werden Wasserspritzer durch ein Partikelsystem dargestellt. Sollte eine Blume getroffen werden, wird eine Funktion des IvyFlower-Skripts der Blume aufgerufen, die sie blühen lässt und das Wachsen des Efeus beginnt. Wird das Spatial Mapping-Mesh getroffen, sendet der Wasserballon den getroffenen Punkt und dessen Oberflächennormale an den Object-Placer, um das Wachstum von Vegetation an dieser Stelle einzuleiten.

Der Spieler hat zu Beginn der Anwendung 60 Wasserballons zur Verfügung, die er durch die Spielszene werfen kann. Das Ziel für den Spieler ist es, so viele Pflanzen wie möglich zu platzieren und einen Highscore aufzustellen. Sobald alle Wasserballons geworfen wurden, wird dem Spieler eine Punktetafel angezeigt, auf der die aktuelle Anzahl der erreichten Punkte und der Highscore der aktuellen Spielesitzung angezeigt werden. Für jede Pflanze, die im Raum platziert wurde, bekommt der Spieler eine gewisse Anzahl an Punkten zugeschrieben. Dabei geben die meisten Pflanzen einen Punkt und die an vertikalen Flächen platzierten Blumen

geben einen Punkt für das Platzieren und zusätzlich einen weiteren Punkt für jede Efeupflanze, die von der Blume aus wächst. Um das Spiel an dieser Stelle (oder einen beliebigen Zeitpunkt im gesamten Spielverlauf, z.B. bei Fehlern bei der Abtastung des Raumes) neu zu starten, kann der Spieler die Worte „Restart Game“ sagen, welche durch die Spracherkennung der HoloLens 2 erkannt werden und den Neustart des Spieles bewirken. Der Highscore bleibt in der aktuellen Sitzung der Anwendung auch bei mehrmaligen Neustarten bestehen.

5.3 ERSTELLEN DES ZULÄSSIGEN BEREICHS

Um einen zulässigen Bereich für das Platzieren der Vegetation auf horizontalen Flächen festzulegen, muss das Spatial Mapping-Mesh analysiert werden, da hierfür der Umfang und die Höhe der verschiedenen Vegetationsarten beachtet werden muss. Am Anfang wurde hierfür versucht, einen eigenen Algorithmus zu schreiben, der zu Beginn des Spieles das erhaltene Mesh der Umgebung untersucht und versucht, genau die Polygone des Meshes ausfindig zu machen, die sich für das Platzieren von Vegetation eigenen. Grundsätzlich müsste man hierfür nur die einzelnen Normalen der Polygone finden, die ungefähr mit dem Up-Vektor der Spielszene übereinstimmen, was leicht durch den Winkel der beiden Vektoren zueinander herauszufinden ist, und anschließend muss man den umliegenden Bereich noch auf den Umfang der einzelnen Pflanzen und deren Höhe prüfen. Jedoch ist schnell klar geworden, dass es, ohne direkte Vorkenntnisse in diesem Bereich zu besitzen, sehr schwer ist einen effizienten Algorithmus dafür zu schreiben, mit dem das Mesh ohne Probleme zügig verarbeitet werden kann. Da die Anwendung auf der HoloLens 2 laufen soll, haben sich hier auch noch zwei weitere Probleme ergeben. Zum einen ist die Rechenleistung der HoloLens 2, im Vergleich zu einem Desktop-Computer, stark eingegrenzt und zum anderen hat das durch Spatial Mapping erstellte Mesh normalerweise eine relativ hohe Polygon-Anzahl, wodurch das Analysieren des Meshes noch aufwendiger ist. Nachdem gute Performance einen integralen Aspekt der Anwendung darstellt, wurden daher für das Ausfindigmachen der zulässigen Bereiche, die NavMesh-Komponenten der Unity Game Engine verwendet. Um das NavMesh zur Laufzeit erstellen zu können, wird zusätzlich ein weiteres Packet namens *NavMeshComponents*[12] von Unity verwendet, in dem die benötigten Komponenten enthalten sind.

Diese Komponenten werden alle direkt von Unity zur Verfügung gestellt und sind nicht nur performant implementiert, sondern auch zuverlässig und es ist einfach, Änderungen an den Einstellungen für die Nav-Mesh-Generation vorzunehmen. Nav-Meshes (Navigation Meshes) werden normalerweise für das Navigieren bzw. Pfadfinden eines KI-Charakters genutzt, allerdings stimmen viele Aspekte davon, mit denen für das Definieren eines zulässigen Bereichs benötigten Aspekten, überein. Die meisten Arten von Videospiel-Charakteren können sich nur auf horizontalen Flächen befinden, deren Steigung nicht über einen bestimmten Winkel hinaus geht, und das Gleiche kann von den meisten implementierten Pflanzenarten dieser Anwendung gesagt werden. Des Weiteren können eine Höhe und ein Radius, sowie der maximale Winkel der Oberfläche angegeben werden. Diese Parameter werden bei der Erstellung des Nav-Meshes beachtet und führen dazu, dass nur Bereiche beim Analysieren des Spatial Mapping-Meshes in den zulässigen Bereich aufgenommen werden, auf denen sich in der Theorie ein Charakter, der diese Parameter besitzt, mit genügend Platz und ohne weitere Probleme, befinden könnte. Natürlich sollte bei der Implementation einer Anwendung wie in dieser Arbeit normalerweise kein Nav-Mesh für solch einen Zweck verwendet werden, da es nicht für die Verwendung auf diese Art vorgesehen ist und dadurch relativ viele ungenutzte Funktionen mit sich bringt. Im

[12] Github-Seite der NavMeshComponents - Unity Technologies - https://github.com/Unity-Technologies/NavMeshComponents

Rahmen dieser Bachelorarbeit bietet das Ganze jedoch eine, im Vergleich zu den erkundeten Alternativen, sehr performante Lösung für das Finden eines zulässigen Bereichs, der für das Platzieren von Vegetation auf horizontalen Flächen benötigt wird.

Für das Generieren und Verwalten der Meshes für den zulässigen Bereich sind die Komponenten Valid-Area-Manager und Nav-Mesh-Converter verantwortlich. Die Bodenvegetation wird in verschiedene Größenkategorien eingeteilt, für die jeweils ein einzelnes Mesh, das als zulässiger Bereich für diese Größenkategorie gilt, durch den Nav-Mesh-Converter erstellt wird. Hierzu wird für jede Größenkategorie ein Nav-Mesh-Agent mit der jeweiligen Höhe, dem Radius und dem maximalen Winkel der Oberfläche festgelegt. Vom Game-Manager wird vor dem Übergang vom ersten Spielzustand in den Zweiten, die Funktion des Valid-Area-Managers aufgerufen, die das Erstellen des Meshes des zulässigen Bereichs startet. In dieser Funktion wird für jede Größenkategorie das Erstellen des Meshes eingeleitet, indem jeweils eine entsprechende Funktion des Nav-Mesh-Converters aufgerufen wird, an die der Index eines für das spezifische Mesh vorgesehenen Physics-Layers und die Nummer des dazugehörigen Nav-Mesh-Agents übergeben wird. Der Nav-Mesh-Converter erstellt damit dann ein Nav-Mesh für den dazugehörigen Nav-Mesh-Agent, welches trianguliert und in ein herkömmliches Mesh umgewandelt wird. Das erhaltene Mesh wird anschließend an ein eigenes Game-Object angehangen und dem spezifizierten Physics-Layer zugewiesen.

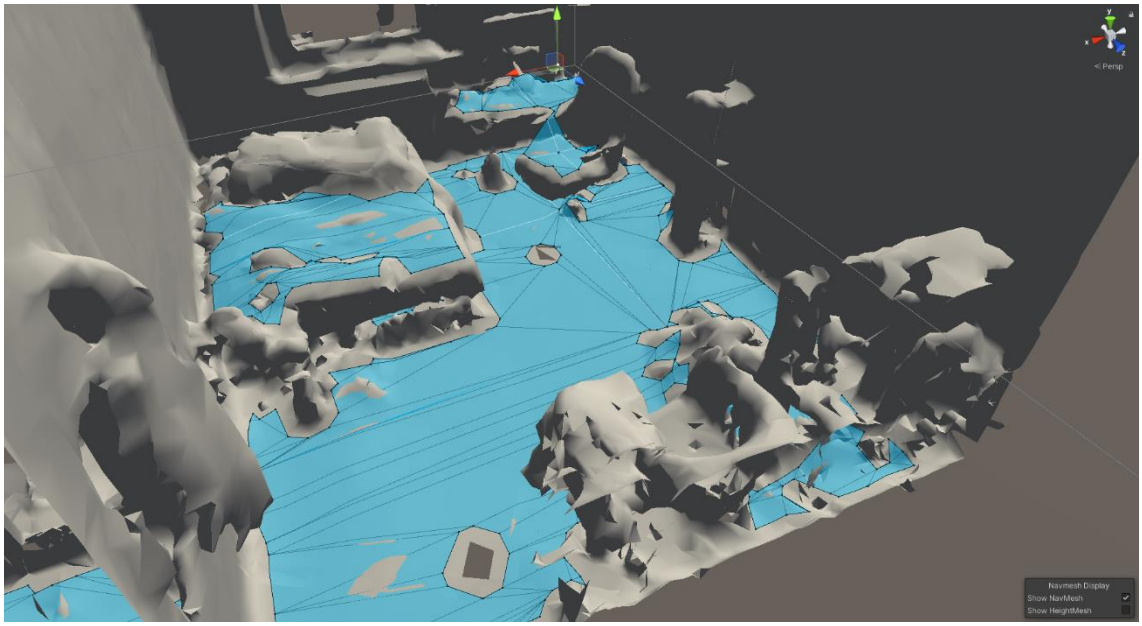

ABBILDUNG 14: DARSTELLUNG EINES NAV-MESHES ALS ZULÄSSIGER BEREICH FÜR DAS PLATZIEREN.

5.4 IMPLEMENTIERUNG DER VEGETATION

Alle Pflanzen, die direkt auf dem Boden oder anderen horizontalen Flächen prozedural platziert werden können, müssen in die Pflanzenliste der GameSettings-Komponente, in Form der Klasse *Plant*, eingetragen werden. Dafür werden mehrere verschiedene Parameter für die jeweilige Pflanzenart benötigt, auf die in anderen Komponenten zugegriffen werden kann. Zum Beispiel braucht jede Pflanzenart eine Bezeichnung bzw. einen Namen, mit dem sie identifiziert werden kann, einen benötigten Radius um die platzierte Pflanze herum, der frei sein muss, und eine Liste an 3D-Modellen in der Form von Prefabs, die die jeweilige Pflanzenart repräsentieren. Weitere Parameter, die für jede Pflanzenart benötigt werden, sind zum einen eine Variable die

angibt, ob die spezifische Pflanzenart auf horizontalen oder vertikalen Flächen wachsen kann, und zum anderen, eine weitere Variable für das Festlegen davon, wie oft eine Pflanzenart in der Szene vorkommen soll und wie viele davon dementsprechend zu Spielbeginn durch das Object-Pooling instanziiert werden. Jede Pflanzenart hat zudem eine Variable für die Wahrscheinlichkeit, mit der sie vom Object-Placer für das Platzieren ausgewählt wird. Dadurch wird erreicht, dass manche Pflanzenarten öfter in der Szene vorkommen können als andere.

Als Vegetation für diese Anwendung wurden mehrere Unkrautarten, Gräßer und Waldpflanzen ausgewählt, die hauptsächlich im mitteleuropäischen Raum zu finden sind. Alle Texturen für die Vegetation wurden von Hand erstellt und sind stilisiert. Im Folgenden werden die gewählten Pflanzen, ihre Verwendung und die 3D-Modelle, die für diese entworfen wurden, kurz näher beschrieben.

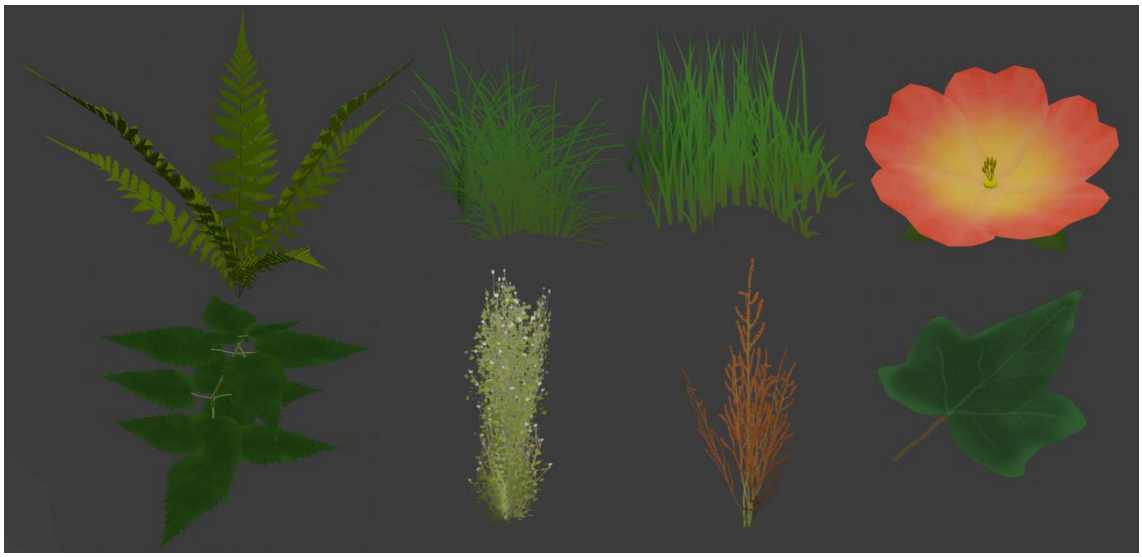

ABBILDUNG 15: ALLE MODELLE DER VEGETATION IM ÜBERBLICK.

Gras und Unkraut

Für das Bedecken des Großteils der horizontalen Flächen wurden allgemeines Gras und Unkraut ausgewählt, welche durch simple Modelle abgebildet werden können. Die Modelle dieser Vegetation werden in der Spielszene sehr nahe beieinander platziert und bilden so zu sagen das Fundament der überwachsenen Oberfläche. Aufgrund dessen, dass die einzelnen Modelle dicht zueinander platziert werden, ist der Detailgrad dieser Modelle nicht so wichtig wie bei den anderen Pflanzen und die 3D-Modelle sind daher sehr einfach gehalten. Das ist auch der Fall, da, je nach Dichte der Vegetation der bereits auf dem Boden platzierten Pflanzen, die Modelle mehrere hundert Mal auf dem Bildschirm zu sehen sein könnten. Aus diesem Grund sind die Modelle für das Gras und Unkraut alle unter 100 Vertices und in der fertigen Szene werden so, im Durchschnitt, nur etwa 10.000 bis 20.000 Vertices in Form dieser Pflanzenarten auf dem Bildschirm gerendert. Um die Vertice-Anzahl so niedrig wie möglich zu halten, bestehen diese Modelle ausschließlich aus einfachen Ebenen, auf die eine transparente Textur gelegt wird, wie in Abbildung 16 zu sehen ist.

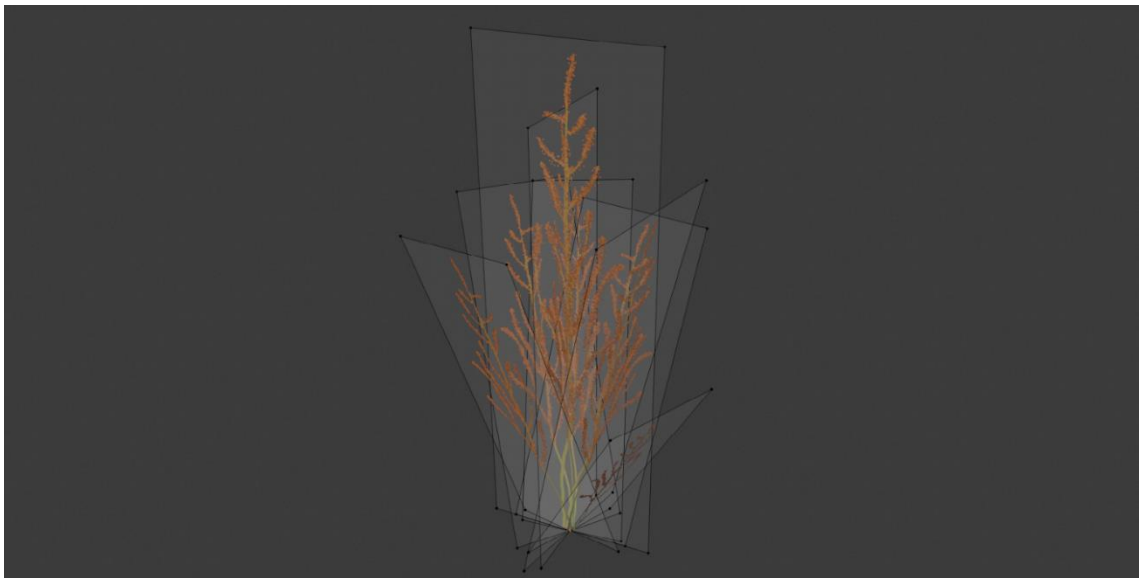

ABBILDUNG 16: DARSTELLUNG EINER PFLANZE DURCH SIMPLE EBENEN AUF DIE EINE TRANSPARENTE TEXTUR GELEGT IST.

Ein Nachteil davon, dass die Modelle nicht komplett ausmodelliert wurden, ist, dass bei der Beleuchtung, je nach Position im Verhältnis zur Lichtquelle, harte Schatten auf dem fertigen Modell zu sehen sind. Das lässt die Beleuchtung nicht nur schlechter wirken, sondern führt auch dazu, dass die Pflanzen als unnatürlich und künstlich vom Spieler wahrgenommen werden. Um das Problem zu lösen, wurden die Ebenen dupliziert und invertiert, damit das Modell beidseitig gerendert wird. Zusätzlich wurden die einzelnen Normalen der Ebenen in der Modellierungs-Software angepasst, bevor sie nach Unity importiert wurden. Das Duplizieren der Ebenen führt zwar zu der doppelten Anzahl an Vertices, jedoch wird die Beleuchtung dadurch auch merkbar verbessert. Für die Anpassung wurden, mithilfe der in der Modellierungs-Software zur Verfügung stehenden Werkzeuge, alle existierenden Normalen auf eine Richtung zu rotiert, welche den Richtungsvektor vom Ursprungspunkt zu der jeweiligen Vertex-Koordinate des Normalen-Ursprungs darstellt. Der Ursprungspunkt liegt bei diesen Modellen an der untersten Stelle in der Mitte des Modells bzw. an dem Punkt, an dem die Pflanze den Boden berührt. Die Rotation der Normalen wurde zudem auf unter 90 Grad beschränkt, da die Richtungen der Ebenen sonst unter Umständen umgedreht werden und so wieder nur eine Seite der Ebenen gerendert wird.

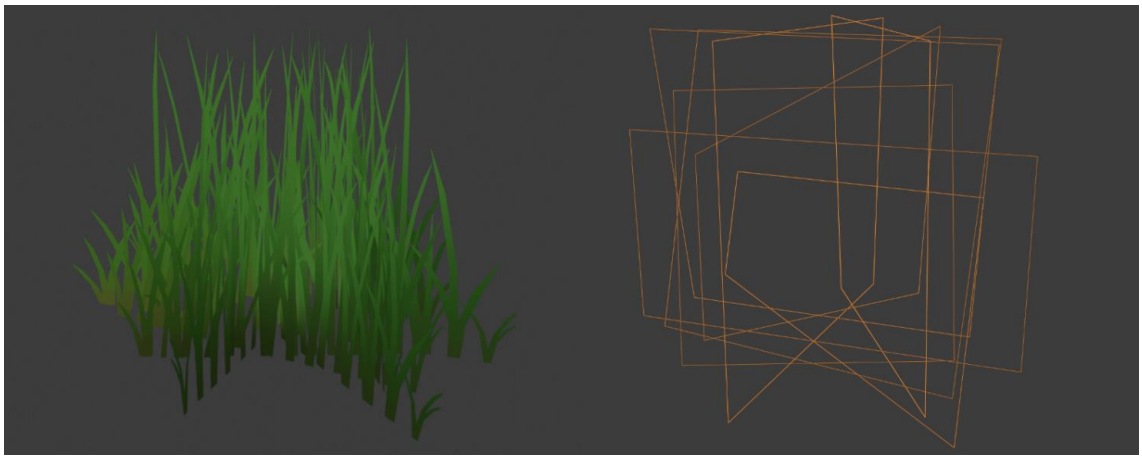

ABBILDUNG 17: EIN MODELL FÜR GRAS IN DER ANWENDUNG.

Brennnesseln und Farn

Da die Vegetation an manchen Punkten, nur mit Gras und etwas Unkraut, teilweise dünn verteilt und monoton wirkt, wurden auch noch größere Pflanzenarten implementiert. Hierfür wurden Brennnesseln und Farn ausgesucht. Diese Pflanzenarten wurden vor allem deswegen ausgewählt, weil sich dafür einfache 3D-Modelle erstellen lassen, die viel Fläche abdecken können, falls der Spieler in einem steilen Blickwinkel auf die Pflanzen schaut. Für das Modell der Brennnesseln werden hierbei, ähnlich wie bei den Modellen für das Gras und das Unkraut, auch Ebenen mit einer transparenten Textur versehen. Diese Methode der Vegetationsmodellierung ist aber nur bei den einzelnen Blättern der Brennnesseln angewandt worden, der Stamm der Pflanze ist hingegen dreidimensional modelliert, jedoch ist dieser trotzdem sehr simpel gehalten. Für das Modell des Farns wurde auch ähnlich zu den Modellen der anderen Pflanzen vorgegangen. Die einzelnen Blätter bestehen hier auch nur aus einer einzigen Ebene, der eine transparente Textur zugewiesen wird. Die Winkel, in denen die Blätter des Farns wachsen, sind relativ steil gehalten, um das Überschneiden mit anderen Pflanzen möglichst zu verhindern. Das heißt zwar, dass das Modell nicht mehr so viel Fläche abdecken kann, aber es wurde versucht, einen guten Mittelwert zwischen der abgedeckten Fläche und den steileren Winkeln, für das Vermeiden von Überschneidungen, zu finden. Bei den Modellen dieser beiden Pflanzenarten wurde zudem auf das Duplizieren und Anpassen der Normalen verzichtet, da sich die einzelnen Ebenen der Modelle nicht merkbar überschneiden und dadurch die Probleme der Beleuchtung nicht auftreten. Um die Rückseiten der Ebenen trotzdem in der Anwendung sehen zu können, ist in Unity stattdessen ein Shader für das importierte Modell verwendet worden, der auch die Rückseiten der Polygone rendert.

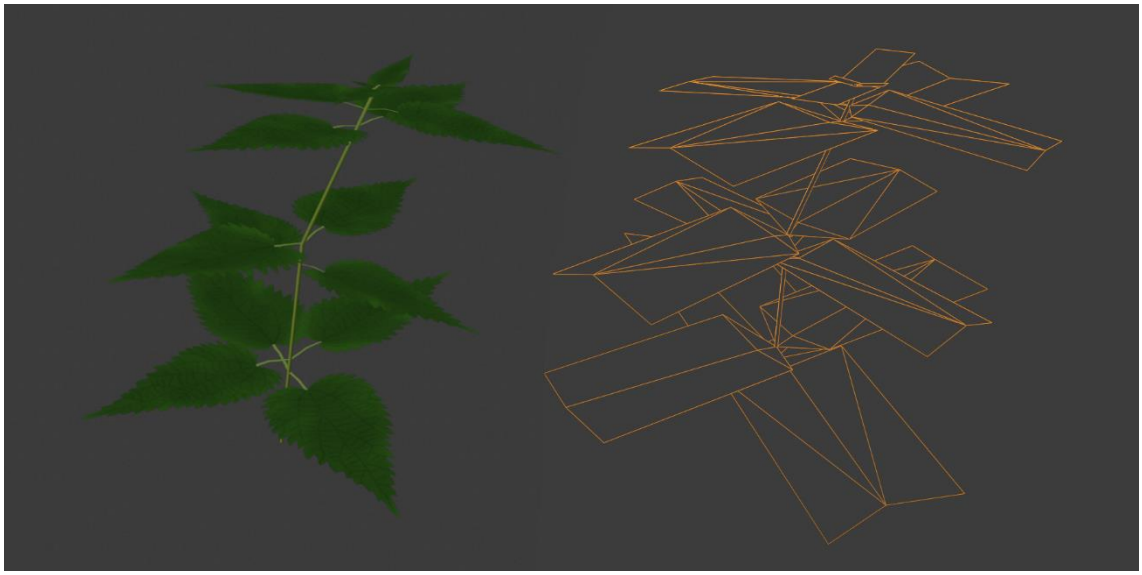

ABBILDUNG 18: DAS MODELL DER BRENNNESSELN IN DER ANWENDUNG.

Efeu und Blumen

Für das Überwuchern von vertikalen Flächen durch Kletterpflanzen wurde für diese Anwendung Efeu ausgewählt, da dieser überall in Mitteleuropa zu finden ist. Um diesen glaubwürdig an Oberflächen platzieren zu können, wurde zudem auch das Modell einer Blume erstellt. Diese Blume orientiert sich nicht direkt an einer realen Pflanze und stellt dadurch die Ausnahme in der implementierten Vegetation dar. Obwohl solch eine große Blume eher tropisch wirkt, wurde sich trotzdem für das Implementieren dieser entschieden, da dadurch der Start der

Kletterpflanze unauffällig versteckt werden kann. Das Modell der Blume ist mit Abstand das komplizierteste Modell dieser Anwendung, da bei der Erstellung auf das Verwenden von Ebenen mit transparenter Textur verzichtet wurde. Stattdessen ist das Modell vollständig dreidimensional ausmodelliert, was für die Implementation der Wachstumsanimation nötig ist, denn diese wurde durch mehrere Blend-Shapes erstellt. Der Efeu selbst besitzt kein festes 3D-Modell, das Mesh des Pflanzenstammes wird stattdessen vollständig prozedural generiert und die Efeublätter entlang des entstehenden Stammes platziert. Da, je nach Länge des Efeus, sehr viele Blätter entlang des Pflanzenkörpers benötigt werden, um den Efeu glaubwürdig aussehen zu lassen, ist das Modell der Blätter außerordentlich einfach gehalten. Wie bei dem Großteil der anderen Pflanzenmodelle der Anwendung wurde hier auch wieder von transparenten Texturen Gebrauch gemacht, aber mit dem Unterschied, dass jedes einzelne Blatt aus nur einer einzigen Ebene besteht und dadurch nur vier Vertices besitzt, wie in Abbildung 19 zu sehen ist.

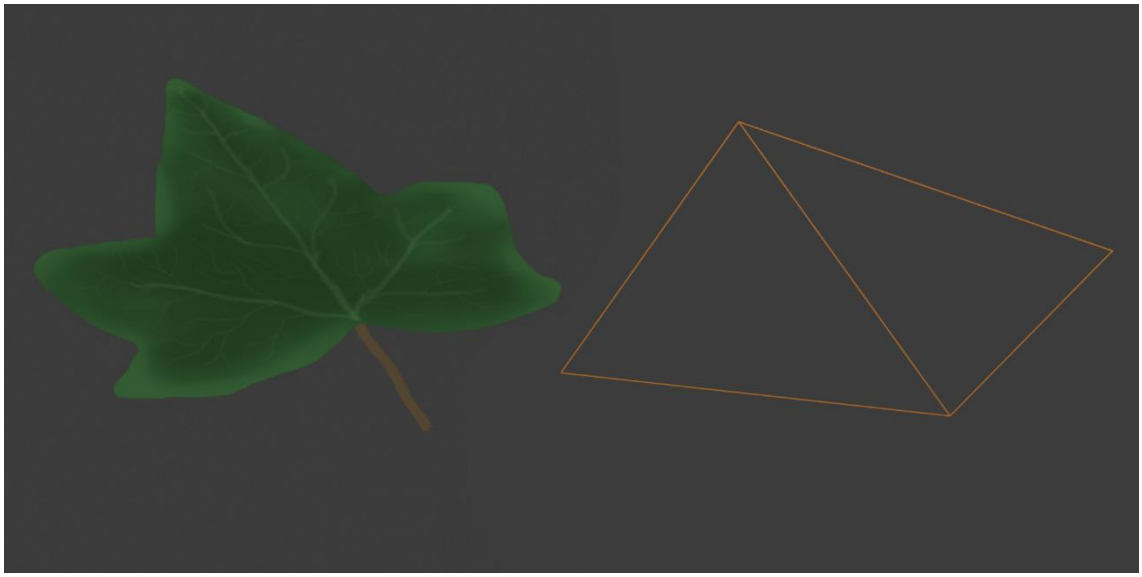

ABBILDUNG 19: DAS MODELL EINES BLATTES DES EFEUS.

Wachstumsanimationen

Jede der erwähnten Pflanzen hat eine Wachstumsanimation, die durch ein Skript, das an das Spieleobjekt der Pflanze angehangen ist, ausgeführt wird. Sobald eine Bodenpflanze, wie zum Beispiel das Gras oder die Brennnesseln, durch den Object-Placer in der Spielwelt platziert wird, setzt das Wachstumsskript die Skalierung entlang der Achsen auf einen vorbestimmten Wert, der von der Art der Wachstumsanimation der Pflanze abhängig ist. Da es reicht, wenn das Gras und Unkraut der Anwendung lediglich in die Höhe wachsen, wird die Skalierung für diese nur entlang ihrer Y-Achse (bzw. der vertikalen Achse) geändert und zu Beginn auf null gesetzt. Die Startskalierung der anderen beiden Achsen beträgt zu Beginn schon den maximalen Wert. Der maximale Wert für die vertikale Skalierung wird zudem auf einen zufälligen Wert zwischen 50% und 100% der ursprünglichen Skalierung gesetzt, um Variation zwischen den einzelnen Instanzen zu schaffen. Über einen vorgegebenen Zeitraum hinweg, wird dann in gleichen Intervallen die Skalierung erhöht und aktualisiert, um das Wachstum zu simulieren. Für die Brennnesseln und den Farn lässt sich diese Art der Animation für das Wachsen nicht gut nutzen, da die Modelle zusätzlich ein horizontales Wachstum benötigen, um glaubhaft zu wirken. Aus diesem Grund wird bei diesen beiden Pflanzenarten die Startskalierung nicht nur entlang der Y-Achse auf null, sondern auch entlang der X- und Z-Achsen auf 10% ihrer ursprünglichen Werte

gesetzt. Für die Wachstumsanimation wird das Modell zuerst nur vertikal hochskaliert, bis 50% des maximalen Wertes erreicht sind, und dann zusätzlich auch noch horizontal. Um hier Variation zwischen den Instanzen zu erreichen, werden die maximalen Werte der Skalierung auch für diese Pflanzenarten durch einen zufälligen Multiplikator entschieden, jedoch entlang aller Achsen anstatt nur entlang der vertikalen Achse. Hier befinden sich die Werte durch den Multiplikator zwischen 70% und 100% der Ursprungsskalierung. Bei den Blättern des Efeus muss im Gegensatz dazu fast gar nichts beachtet werden und ihr Modell besitzt deshalb die simpelste Animation für das Wachstum. Bei den Efeublättern wird die Skalierung aller Achsen am Anfang auf null gesetzt und das Modell wird dann in bestimmten Zeitintervallen hochskaliert. Sobald, wie bei den anderen Wachstumsanimationen, ein vorbestimmter Schwellwert erreicht ist, gilt das Wachstum des Blattes als abgeschlossen.

Für die Blume, von der aus der Efeu das Wachsen beginnt, ist eine aufwändigere Animation erstellt worden, um das Öffnen der Blüte richtig darzustellen. Sobald die Blume platziert wird, ist die Blüte noch geschlossen und fängt an, mit einer ähnlichen Skalierungsfunktion wie die des Efeus, zu wachsen. Wenn eine noch geschlossene Blume vom Spieler mit einem Wasserballon getroffen wird, startet das Öffnen der Blüte. Für das Animieren des Öffnens der Blüte wurden für das 3D-Modell, neben der Ausgangsform des Meshes, zwei Blend-Shapes definiert, zwischen denen in der Animation übergegangen wird.

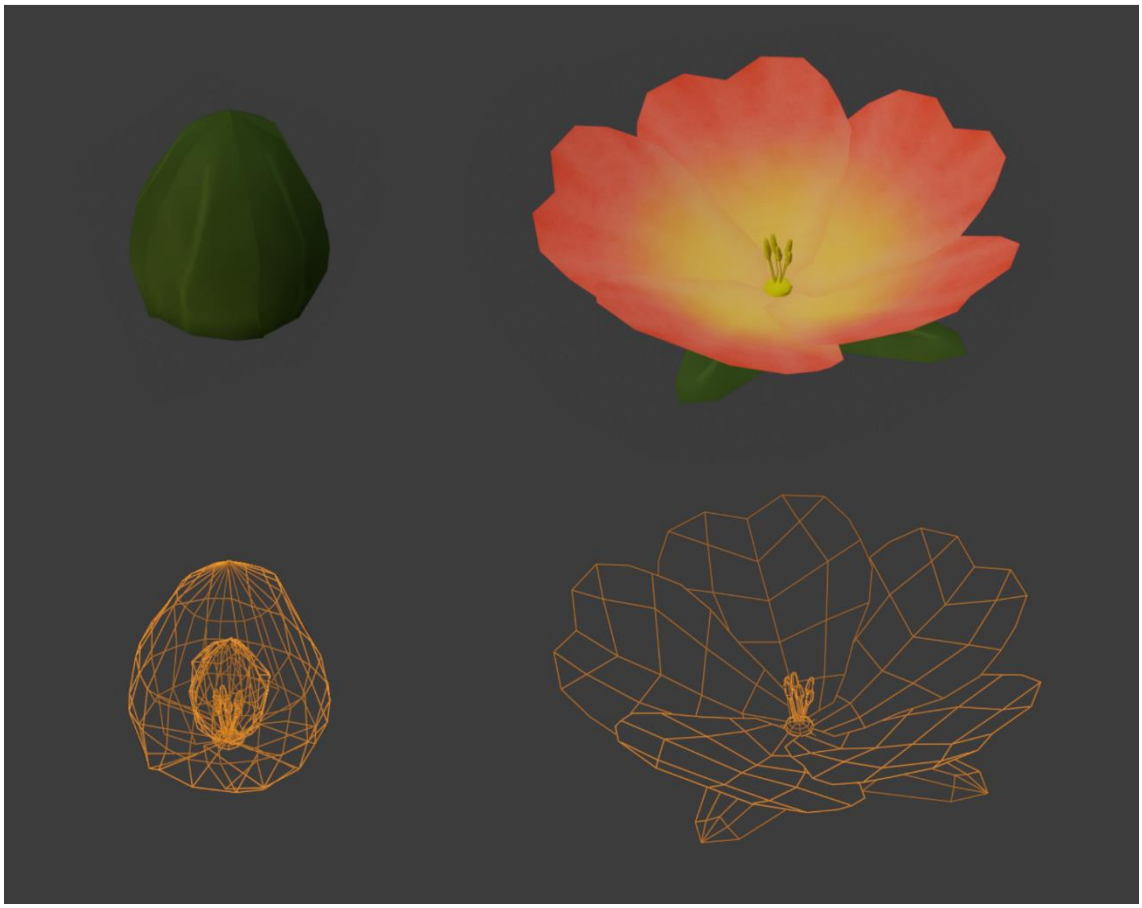

ABBILDUNG 20: ZWEI VERSCHIEDENE BLEND-SHAPES DES BLUMENMODELLS UND IHR WIRE-MESH.

5.5 PROZEDURALES PLATZIEREN DER VEGETATION

Das prozedurale Platzieren der Pflanzen findet durch den Object-Placer statt. Dieses Skript holt sich zu Beginn des Spiels eine Liste von platzierbaren Pflanzen von den Game-Settings und hat mehrere Funktionen, die Vegetation auf verschiedene Weisen platzieren können. Für das allgemeine Platzieren von Vegetation durch den Spieler wird die Funktion *SpawnObjectsAtHit* aufgerufen, welche einen Punkt auf dem Spatial Mapping-Mesh und dessen Normale benötigt. Je nachdem, welchen Winkel die Normale des übergebenen Punktes zum Up-Vektor der Szene hat, entscheidet die Funktion, ob an dieser spezifischen Stelle eine horizontale oder vertikale Oberfläche vorliegt und ruft die entsprechende Funktion für den vorliegenden Fall auf.

Falls an dem Punkt eine horizontale Fläche vorliegt, werden wieder der Punkt und dessen Normale, ein Radius in dem versucht wird, Vegetation zu platzieren, und die maximale Anzahl an zu platzierenden Pflanzen, sowie eine maximale Anzahl an Versuchen für das Platzieren dieser benötigt. Die Variablen für die maximale Anzahl an zu platzierenden Pflanzen und die maximale Anzahl an Versuchen, diese zu platzieren, werden gebraucht, da einzelne Pflanzen Platz brauchen und deshalb nicht garantiert werden kann, dass in dem gewünschten Radius auch wirklich Platz für alle Pflanzen gefunden wird. Die Funktion wird solange ausgeführt, bis entweder die maximale Anzahl an Versuchen erreicht oder die maximale Anzahl an Pflanzen platziert wurde. Bei jedem Versuch, eine Pflanze zu platzieren, wird eine zufällige Pflanzenart, in Abhängigkeit von den einzelnen Wahrscheinlichkeiten aller Pflanzenarten, ausgesucht und eine Stelle innerhalb des Radius gewählt. Anschließend werden mehrere Tests ausgeführt, um herauszufinden, ob die ausgesuchte Stelle geeignet ist. Falls einer dieser Test fehlschlägt, wird der aktuelle Versuch abgebrochen und ein neuer Versuch, an einer neuen zufälligen Stelle und mit einer erneut zufällig ausgesuchten Pflanzenart, gestartet.

Als erstes wird geprüft, ob die Stelle im zulässigen Bereich liegt. Diese Prüfung findet statt, indem an dem zufällig ausgesuchten Punkt, mit etwas Abstand zur Oberfläche, ein Raycast nach unten ausgeführt wird, der auf die verschiedenen Meshes des zulässigen Bereichs prüft. Je nach dem, in welcher Größenkategorie sich die ausgesuchte Pflanze befindet, wird für diesen Raycast eine andere Maske für das Physics-Layer gewählt, wodurch nur auf das korrespondierende Mesh des zulässigen Bereichs geprüft wird, für das diese Pflanzenart vorgesehen ist. Ist der Punkt zulässig, wird an der gleichen Stelle ein Punkt auf dem Spatial Mapping-Mesh gesucht. Kann ein Punkt auf dem Spatial Mapping-Mesh gefunden werden, wird auf Kollision mit anderen Pflanzen getestet. Für den Kollisions-Test mit anderen Pflanzen wird eine Sphäre benutzt, deren Größe durch den benötigten Platz der platzierten Pflanze festgelegt ist. Sollte sich der Physics-Collider einer anderen Pflanze entweder innerhalb der Sphäre befinden oder diese schneiden, kollidiert die Pflanze an ihrer momentan ausgesuchten Position mit einer bereits existierenden Pflanze. Im Falle davon, dass keine Kollision gefunden wurde, kann die Pflanze an der gefunden Stelle auf dem Spatial Mapping-Mesh platziert werden. Falls eine Kollision vorliegt, wird versucht eine neue Position ausfindig zu machen. Für das Finden einer neuen Platzierungsposition wird auch wieder eine maximale Anzahl an Versuchen festgelegt, da, vor allem bei einem bereits dicht überwucherten Raum, nicht garantiert werden kann, dass die Suche nach einer neuen Position deterministisch ist. Die neue Position setzt sich aus der aktuellen Platzierungsposition und einem Vektor zusammen, der aus der aktuellen Platzierungsposition und den Positionen der Pflanzen, mit denen kollidiert wurde, berechnet wird, und sozusagen als Fluchtvektor der Pflanze dient. An der neu ausgesuchten Stelle wird geprüft, ob diese noch im Radius für das Platzieren liegt und anschließend wieder eine Stelle auf dem Spatial Mapping-Mesh gesucht, bei der auch wieder auf Kollision mit anderen, bereits existierenden Pflanzen, getestet wird. Liegt an der neuen Stelle immer noch eine Kollision vor, wiederholt sich das Finden einer neuen Position noch einmal, aber dieses Mal, mit der neu gefundenen Stelle als Startposition. Wenn alle Versuche, eine neue Platzierungsposition zu finden, zu keinem Ergebnis kommen, wird der gesamte Versuch, die spezifische Pflanze zu

platzieren, abgebrochen und ein komplett neuer Versuch gestartet. Für das Platzieren einer Pflanze, wird eine Instanz der Pflanze bei der ObjectPooling-Komponente angefragt, entsprechend positioniert und anhand des Up-Vektors der Szene ausgerichtet. Damit ist das Platzieren einer Pflanze auf einer horizontalen Oberfläche abgeschlossen.

Wenn die Oberfläche als vertikal kategorisiert wurde, wird an der getroffenen Stelle überprüft, ob genug Platz zur Verfügung steht. Da es, bei der Anwendung, die für diese Arbeit entwickelt wurde, nur eine einzige Pflanzenart gibt, die direkt an vertikalen Flächen platziert werden kann, nämlich die Blumenblüten, ist die Funktion für das Prüfen ob genug Platz vorhanden ist und das anschließende Platzieren stark an der spezifischen Pflanzenart orientiert und nicht so allgemein gehalten, wie beim Platzieren auf horizontalen Flächen. An vertikalen Flächen wird im Vergleich eine viel geringere Anzahl an Pflanzen direkt platziert und dadurch wird für das Platzieren kein zulässiger Bereich benötigt, sondern manuelle Tests in der direkten Umgebung durchgeführt. Erst einmal wird hier, auch wieder mit einer Sphäre, getestet, ob sich bereits eine Pflanze der gleichen Art in dem benötigten Radius befindet. Das Prüfen auf gleiche Pflanzen im Umkreis wird nicht nur gemacht, um Überschneiden der Modelle zu verhindern, sondern auch, da der Spieler sonst unbeabsichtigt Pflanzen platzieren kann, wenn er versucht, eine bereits existierende Blume mit einem Wasserballon zu treffen. Falls keine existierenden Pflanzen gefunden werden, werden mehrere Raycasts in gleichen Abständen, mit etwas Abstand zur Oberfläche und senkrecht zur Normale des getroffenen Oberflächenpunktes, ausgeführt. Diese Raycasts enden am Rand des benötigten Radius, falls sie nichts treffen. Wenn einer der Raycasts etwas innerhalb des Radius trifft, kann keine Pflanze an der ursprünglich getroffenen Stelle platziert werden. In dem Fall, dass alle Raycasts den Rand des Radius erreichen, wird von diesen Punkten aus, parallel zur Normale, nach unten ein Raycast ausgeführt. Wird hierbei bei allen Raycasts im Physics-Layer des Spatial Mapping-Meshes nichts getroffen oder die getroffene Oberfläche kann als relativ Eben angesehen werden, was durch den Winkel zur Normale des getroffenen Punktes bestimmt werden kann, dann wird der ursprünglich getroffene Punkt als valide für das Platzieren der Blume eingestuft. Liegt ein valider Punkt vor, platziert die Funktion an der Stelle die geeignete Pflanze, der eine zufällige Rotation um die Normale des Punktes zugewiesen wird.

Der Object-Placer hat zudem zwei weitere Funktionen, die durch den Game-Manager, vor dem Wechseln zum letzten Spielzustand, aufgerufen werden. Diese Funktionen sind jeweils für das automatische Füllen des Raumes mit Vegetation an vertikalen und horizontalen Flächen verantwortlich.

Beim automatischen Platzieren von Pflanzen an vertikalen Flächen zum Spielbeginn wird von einem Punkt, etwas über dem Kopf des Spielers, eine vorgegebene Anzahl an Raycasts ausgesendet, deren jeweilige Richtungen zufällig um die Y-Achse der Szene herum gewählt sind und etwas in ihren vertikalen Winkeln variieren. Sollten diese Raycasts etwas treffen, wird anhand der Normale überprüft, ob es sich bei dem getroffenen Punkt um eine Stelle an einer vertikalen Fläche handelt. Wenn das der Fall ist, kann, wie bei dem vorherig erwähnten Platzieren an vertikalen Oberflächen, getestet werden, ob genug Platz zur Verfügung steht und wenn ja, wird an der Stelle eine Pflanze platziert.

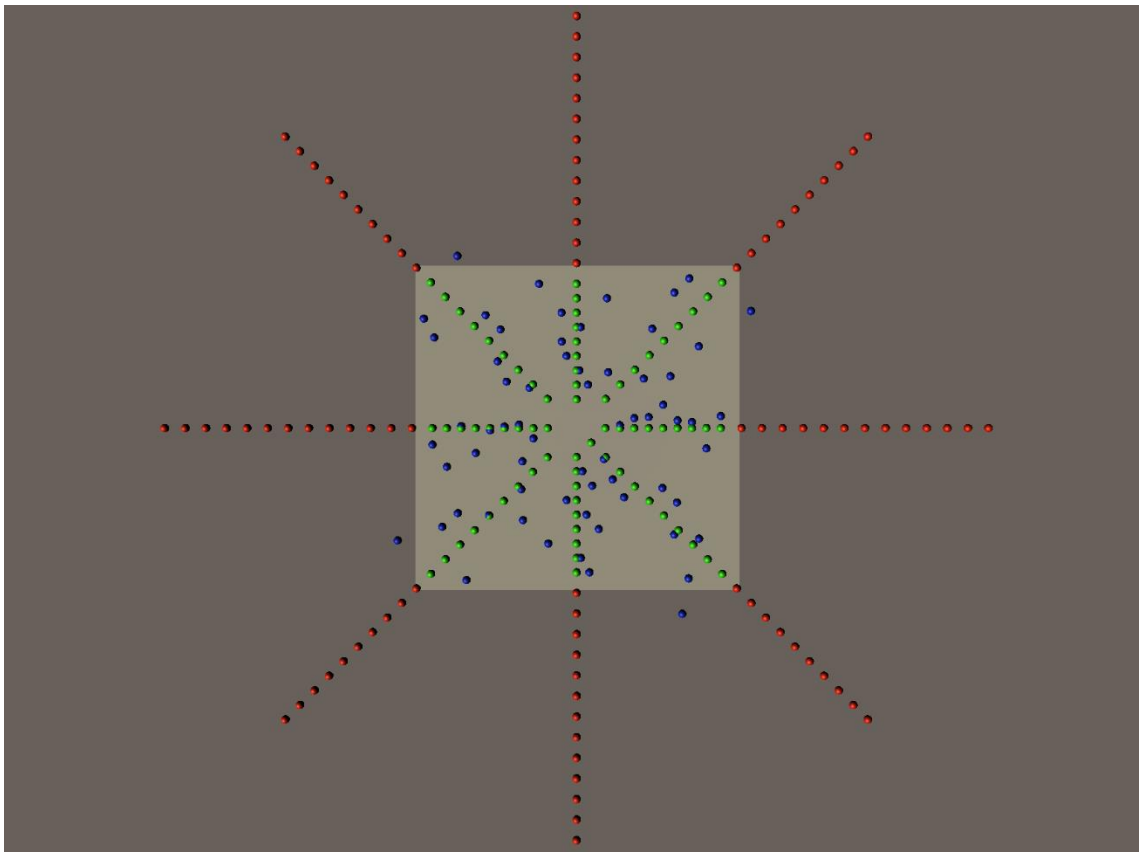

ABBILDUNG 21: VERANSCHAULICHUNG DES ALGORITHMUS FÜR DAS AUTOMATISCHE PLATZIEREN VON VEGETATION AUF HORIZONTALEN FLÄCHEN. DIE ROTEN PUNKTE STELLEN DEN BEREICH DAR, IN DEM DAS MESH NICHT GETROFFEN WURDE. DIE GRÜNEN PUNKTE SIND STELLEN, AN DENEN MESH ERKANNT WURDE. BLAUE PUNKTE MARKIEREN DIE ENDGÜLTIGEN STELLEN, AN DENEN DAS PLATZIEREN DER PFLANZEN AUSGEFÜHRT WIRD.

Für horizontale Flächen werden, auch etwas über dem Kopf des Spielers, in verschiedenen Richtungen, senkrecht zum Up-Vektor der Spielszene und in gleichen Abständen zueinander, mehrere Punkte in der Spielwelt festgelegt, die auf einem bestimmten Radius um den Spieler liegen. Dieser Radius wurde zum Beispiel in dieser Anwendung auf 10 Meter gesetzt, da die Verwendung der Anwendung für die Nutzung in durchschnittlichen Wohnungsräumen vorgesehen ist und mit einem Radius von 10 Metern die ausgesuchten Punkte, in den meisten Fällen, garantiert außerhalb des gescannten Raumes liegen. Die Strecke zwischen den diversen gefundenen Punkten und der Startposition wird jeweils in einzelne Segmente eingeteilt, die im nächsten Schritt von dem äußersten Segment aus nach innen durchgegangen werden. Bei jedem separaten Segment wird ein Punkt ausgesucht, der sich, in einem zufälligen Abstand, links oder rechts von der Verbindungslinie zum Spieler befindet. Der Abstand nach links oder rechts dient hier vor allem dazu, dass die Pflanzen nicht von mehr oder weniger uniform verteilten Punkten aus platziert werden und sich hier kein Muster bildet, dass der Spieler erkennen kann. Von dem Punkt aus, wird ein Raycast nach unten ausgeführt, der auf das Mesh des zulässigen Bereichs für die Vegetation, die sich in der kleinsten Größenkategorie befindet, prüft. An dieser Stelle könnte man zwar auch auf das Spatial Mapping-Mesh testen, jedoch müsste man in diesem Fall danach trotzdem überprüfen, ob sich die Stelle in einem zulässigen Bereich befindet. Wenn ein Punkt auf dem Mesh gefunden werden kann, wird dort das Platzieren von Pflanzen, wie bei der bereits beschriebenen Funktion für das Platzieren auf horizontalen Flächen durch den Spieler, gestartet.

5.6 GENERIEREN DER KLETTERPFLANZEN

Für das Finden eines geeigneten Pfades, an dem die Kletterpflanze entlang wachsen kann, wird das Mesh der Spielumgebung abgetastet. Der Spatial-Awareness-Mesh-Observer liefert hierfür das Spatial Mapping-Mesh des Raumes, in dem sich der Spieler befindet und welcher durch die HoloLens 2 zum Spielbeginn gescannt wurde. Das Mixed Reality Toolkit gibt dem generieten Spatial Mapping-Mesh in Unity standartmäßig ein eigenes Physics-Layer, welches für das Pfadfinden durch Raycasts sehr praktisch ist. Für das Problem des Pfadfindens der Kletterpflanze bietet es sich daher an, dass Ganze durch mehrere Raycasts zu lösen. Hierfür wurde die Komponente *Pathfinding* implementiert, die jeder Instanz des Efeus angehangen ist. Das Pfadfinden durch dieses Skript wird gestartet, sobald die FindPath-Funktion des Skripts aufgerufen wird, was durch das Treffen der Blume, die die spezifische Instanz des Efeus verwaltet, mit einem Wasserballon geschieht. Hierbei muss auch die Anzahl an Iterationen bzw. die maximale Anzahl an Pfadsegmenten übergeben werden. Für jede Iteration wird versucht, dem bereits gefundenen Pfad weitere Punkte hinzuzufügen, falls diese ohne Probleme gefunden werden können. Die Punkte des Pfades werden in eine Liste eingetragen und jeweils in der Form des Structs *Node*, welches die Position, Rotation, den Up-Vektor der Stelle und die Art des Hindernisses, welches an dem Punkt vorkommt, abspeichert. Als erster Punkt des Pfades wird die Startposition des Efeus genutzt. Jeder Punkt, der dem Pfad hinzugefügt wird, hat zudem den Radius des Efeu-Stammes als Abstand zur Oberfläche. Wenn eine vorgegebene Anzahl von Versuchen, einen nächsten Punkt des Pfades zu finden, nacheinander keinen Punkt ausfindig machen kann, wird das weitere Pfadfinden für die dazugehörige Efeu-Pflanze eingestellt, da angenommen wird, dass es auch bei weiteren Versuchen zu keinem Ergebnis kommt. Für das Finden der nächsten Punkte des Pfades, wird die Funktion *FindNextPoint* aufgerufen, welche von dem letzten gefundenen Punkt aus eine Richtung auswählt und mithilfe von Raycasts entscheidet, was für eine Art von Hindernis in dieser Richtung liegt. Diese Funktion macht auch Gebrauch von einem sogenannten Empty, welches im Folgenden als Helper bzw. Helper-Game-Object bezeichnet wird. Dieses Helper-Game-Object erleichtert das Herausfinden der aktuellen Vorwärtsrichtung, Position und Orientation für das Pfadfinden, da es sich in der Regel stehts an dem letzten Punkt des Pfades befindet. Die ausgewählte Richtung wird durch einen Winkel bestimmt, der sich aus einer Variabel für die maximale Winkeländerung des Efeus, im Vergleich zu der aktuellen Richtung des Helpers, und einer Variable für eine Richtungstendenz, zusammensetzt. Die maximale Winkeländerung der Richtung wird in den Game-Settings festgelegt und die Richtungstendenz wird durch die implementierten Verhaltensweisen von Kletterpflanzen bestimmt. Um das Überschneiden mit anderen Blumen möglichst zu vermeiden, wird sobald die Richtung berechnet wurde, mithilfe eines einzigen Raycasts getestet, ob in dieser Richtung eine Blume getroffen wird. Ist das der Fall, wird, um auszuweichen, eine neue Richtung mithilfe der maximalen Winkeländerung bestimmt. Sobald die endgültige Richtung bekannt ist, wird in diese ein weiterer Raycast, ausgehend von der aktuellen Position des Helpers, gesendet, welcher in einer vordefinierten Distanz auf das Mesh der Umgebung testet. Hier findet die Unterscheidung zwischen drei Arten von Hindernissen statt, welche sich anhand einer steigenden, absteigenden und relativ ebenen Oberfläche einteilen lassen. Die Art des Hindernisses wird durch ein Ausschlussverfahren mithilfe des Ergebnisses des Raycasts identifiziert und je nachdem, welche Art von Hindernis identifiziert wurde, wird eine andere Funktion für das Ermitteln der neuen Pfadpunkte aufgerufen.

Steigende Oberfläche

Wenn der Raycast etwas getroffen hat, liegt eine steigende Oberfläche (z.B. eine Wand) vor und es wird die entsprechende Funktion aufgerufen. Diese Funktion fügt dem Pfad insgesamt zwei Punkte hinzu, den Ersten kurz vor der gefunden Stelle und den Zweiten, mit etwas Abstand, an der gefundenen Stelle und orientiert entlang der neuen Oberfläche. Für den ersten Punkt wird

von dem getroffenen Punkt aus, in Richtung des letzten Punktes, etwas zurückgegangen, dort der aktuelle Up-Vektor des Helpers als Up-Vektor genutzt und der Punkt als Node in den Pfad eingetragen. Als zweiter Punkt wird der vom ersten Raycast getroffene Punkt, auch mit zusätzlichem Abstand zur Oberfläche, gewählt, in den Pfad als Node eingefügt und mithilfe der Oberflächennormale der Stelle orientiert. Um einen neuen Forward-Vektor für den Helper zu finden, werden zwei Ebenen in der Spielwelt definiert. Die erste Ebene ist die Ebene, die durch den Vektor, der senkrecht zu den aktuellen Up- und Forward-Vektoren des Helper-Game-Objects ist, an der Position des Helper-Objekts aufgespannt wird. Die zweite Ebene wird, sehr ähnlich, durch die Position, die der erste Raycast getroffen hat und die Oberflächennormale dieses Punktes definiert. Um die neue Richtung, an der sich der Helper orientieren soll, herauszufinden, wird die Schnittgerade der beiden Ebenen berechnet. Anschließend wird die Position der Transformations-Komponente des Helpers auf die Position des zweiten Punktes gesetzt und die Rotation mithilfe der Normale des zweiten Punktes und der Schnittgerade der beiden Ebenen aktualisiert.

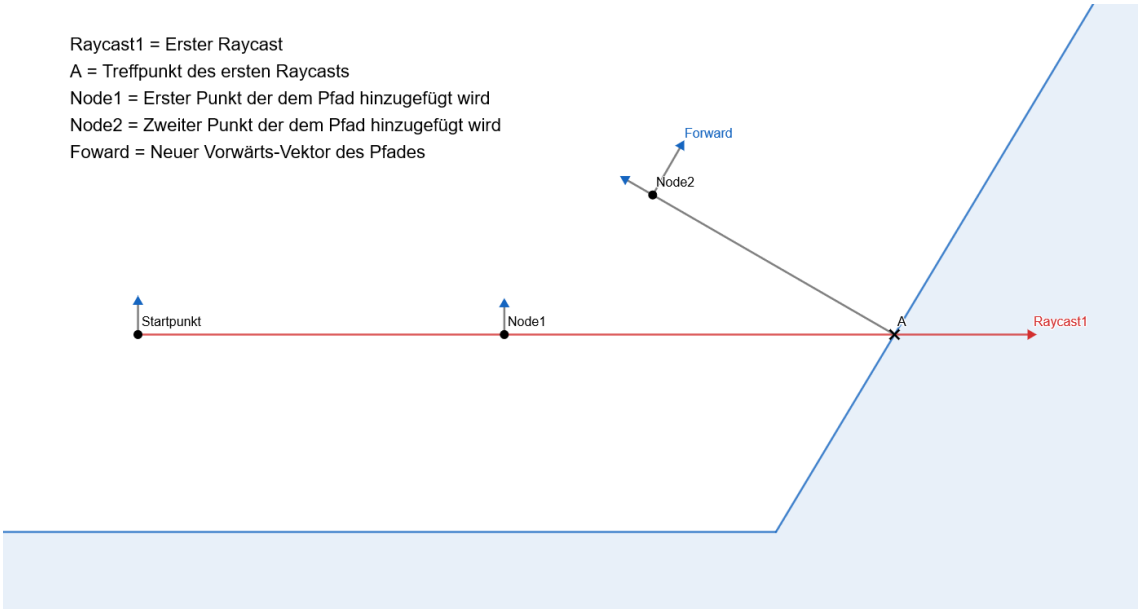

ABBILDUNG 22: 2D-DARSTELLUNG DER RAYCASTS BEI EINER STEIGENDEN OBERFLÄCHE.

Sollte nichts getroffen werden, muss es sich um eine abfallende oder, mehr oder weniger, ebene Oberfläche des Spatial Mapping-Meshes handeln. Um zwischen den beiden Fällen unterscheiden zu können, wird am Ende des ersten Raycasts ein zweiter Raycast in Richtung des Down-Vektors des Helper-Objekts ausgeführt, der in einem Radius, der etwas größer wie der Radius des Efeus ist, auf Kollision mit dem Spatial Mapping-Mesh prüft. Kommt es dabei zu einem Treffer, kann davon ausgegangen werden, dass die Oberfläche mehr oder weniger eben ist. Wenn in dem Radius kein Teil des Umgebungsmeshes getroffen wird, ist die Steigung der Oberfläche an dieser Stelle sehr stark abfallend, das heißt, es handelt sich zum Beispiel um die Kante eines Tisches oder ein Loch im Mesh.

Ebene Oberfläche

Bei einer ebenen Oberfläche wird, in der entsprechend aufgerufenen Funktion, im Vergleich hingegen nicht viel gemacht. Hier wird nur ein neuer Punkt an der Stelle, die der zweite Raycast getroffen hat, als Node in den Pfad eingefügt und, ähnlich wie bei der Funktion für eine

Oberfläche mit starker Steigung, ein neuer Forward-Vektor für den Helper berechnet, um dessen Position und Rotation entsprechend anzupassen und zu aktualisieren.

Raycast1 = Erster Raycast
Raycast2 = Zweiter Raycast
B = Treffpunkt des zweiten Raycasts
Node1 = Der Punkt der dem Pfad hinzugefügt wird
Forward = Neuer Vorwärts-Vektor des Pfades

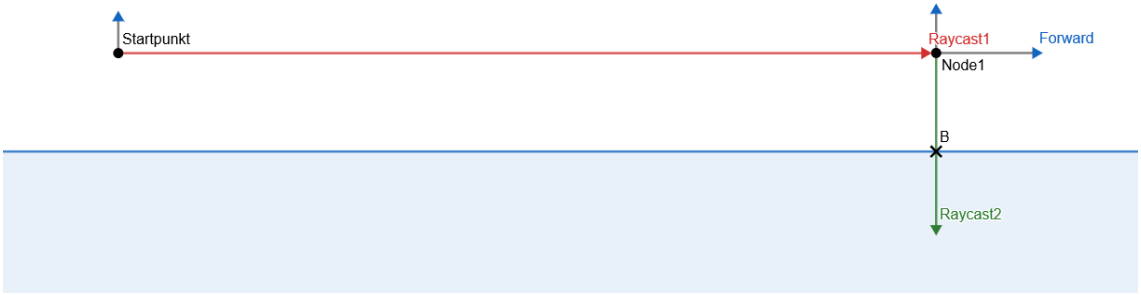

ABBILDUNG 23: 2D-DARSTELLUNG DER RAYCASTS BEI EINER EBENEN OBERFLÄCHE.

Absteigende Oberfläche

Wenn es sich um eine Oberfläche mit abfallender Steigung handelt, wird von dem Endpunkt des zweiten Raycasts, ein dritter Raycast in die umgekehrte Richtung des ersten Raycasts ausgeführt. Trifft dieser nichts, wird der aktuelle Versuch, einen weiteren Punkt für den Pfad zu finden, abgebrochen und eine neue Iteration startet, da ohne eine große Anzahl an zusätzlichen Raycasts nicht sicher entschieden werden kann, ob es sich bei dem aktuellen Untergrund in diesem Fall um ein Loch im Mesh oder zum Beispiel um ein sehr dünnes Objekt handelt. Hat der dritte Raycast etwas getroffen, reicht es nicht aus, dem Pfad den getroffenen Punkt als Node hinzuzufügen, da das dazu führen würde, dass die Kletterpflanze, je nach Steigung der Ebene, durch das Mesh hindurch wächst. Stattdessen muss hier der Punkt in der Szene berechnet werden, an dem sich die beiden Geraden, die sich aus den Positionen und den jeweiligen Richtungsvektoren bilden, schneiden. Die erste Gerade wird durch den Startpunkt und den Richtungsvektor des ersten Raycasts definiert und die zweite Gerade durch den getroffenen Punkt mit Abstand zur Oberfläche und den entsprechenden Richtungsvektor entlang der getroffenen Oberfläche. Die Normale dieses Punktes wird als Vektor, der sich aus dem Up-Vektor des Helpers und der Oberflächennormale des getroffenen Punktes des dritten Raycasts zusammensetzt, vor dem Einsetzen des Punktes in den Pfad berechnet. Anschließend wird, wie bei den beiden anderen Funktionen, auch hier ein neuer Vorwärtsvektor mittels der zwei Ebenen berechnet und damit, zusammen mit der Position des berechneten Punktes, die Transformation des Helpers aktualisiert.

Raycast1 = Erster Raycast
Raycast2 = Zweiter Raycast
Raycast3 = Dritter Raycast
C = Treffpunkt des dritten Raycasts
Direction = berechneter Vektor entlang der Oberfläche
Node1 = Punkt der dem Pfad hinzugefügt wird
Normal = Normale des neuen Punktes
Forward = Neuer Vorwärts-Vektor des Pfades

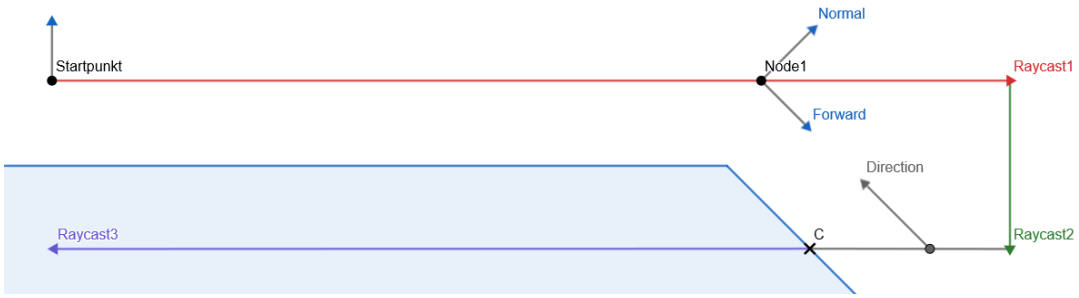

ABBILDUNG 24: 2D- DARSTELLUNG DER RAYCASTS BEI EINER ABSTEIGENDEN OBERFLÄCHE.

Wie schon erwähnt, benötigt die Pathfinding-Komponente für das Finden der initialen Richtung bei jedem Versuch, die nächsten Punkte im Pfad zu finden, eine Winkeltendenz, die von den verschiedenen Verhaltensweisen der Kletterpflanze geliefert wird. Diese Verhaltensweisen sind in der Klasse *IvyBehaviour* implementiert, bei der die Pathfinding-Komponente den Tendenzwert anfragen kann. Für das Anfragen der Tendenz ruft die Komponente eine Funktion der IvyBehaviour-Klasse auf, an die die aktuelle Position und Rotation (definiert durch den Forward- und Up-Vektor) des Helper-Game-Objects, sowie eine Referenz auf das Pathfinding-Skript als Eingabeparameter übergeben werden. Diese Funktion kann mithilfe der erhaltenen Parameter entscheiden, welches Verhalten ausgeführt werden soll. Dafür wird der Winkel zwischen dem erhaltenen Up-Vektor und dem Down-Vektor der Spielszene berechnet. Mit dem berechneten Winkel kann die Funktion dann durch vorbestimmte Schwellwerte entscheiden, auf welcher Art von Oberfläche gerade versucht wird, einen Pfad zu finden und je nach dem, das passende Verhalten wählen. In dieser Anwendung wurden insgesamt drei Verhaltensweisen implementiert, die jeweils auf einer horizontalen Fläche, einer vertikalen Fläche oder an Flächen, deren Normale nach unten zeigt, ausgeführt werden.

Verhalten an horizontalen Flächen

Falls der berechnete Winkel größer als oder gleich 150 Grad ist, wird angenommen, dass sich der Helper momentan auf dem Boden bzw. einer relativ ebenen Fläche befindet. Um weiter nach oben zu gelangen, versucht die Kletterpflanze auf solch einer Fläche ein Gebilde in der Umgebung zu finden, an dem sie emporklettern kann. Die Implementierung der realen Mechanismen, die dafür genutzt werden, würde das Verhalten sehr kompliziert machen, daher wird das Ganze hier durch einen Sichtkegel an der Spitze simuliert, der das Abtasten der Umgebung ermöglicht. Innerhalb dieses Wahrnehmungskegels, welcher durch einen maximalen Winkel und eine Distanz definiert ist, wird, von links nach rechts, in regelmäßigen Winkelabständen ein Raycast entlang der Oberfläche ausgeführt. Durch die Ergebnisse der Raycasts, die etwas getroffen haben, wird die Richtung ausgewählt, in der sich das Gebilde für das Klettern befindet, das der momentanen Position am nächsten ist. Die Winkeltendenz für dieses Verhalten ist der Winkel zwischen der aktuellen Vorwärtsrichtung des Helpers und der gefundenen Richtung für das nächstgelegene Gebilde. Sollte jedoch keiner der Raycasts etwas getroffen haben, ist die Winkeltendenz natürlich null.

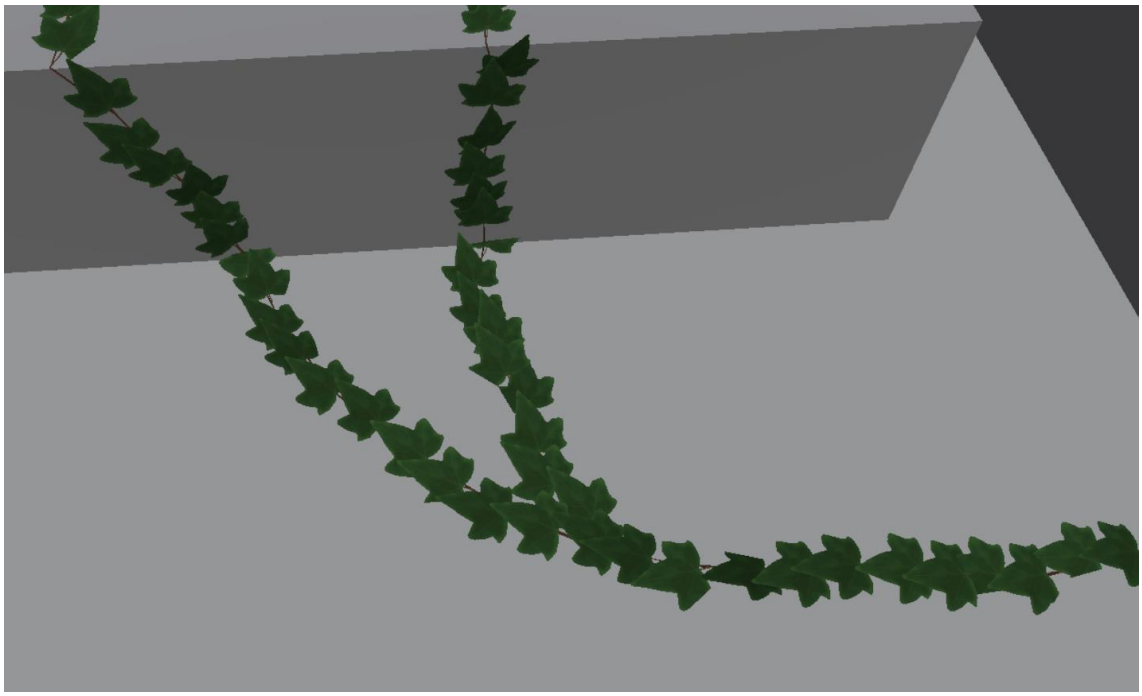

ABBILDUNG 25: EINE BEISPIELDARSTELLUNG DES VERHALTENS DES EFEUS AUF HORIZONTALEN FLÄCHEN. DER EFEU SUCHT SICH DURCH ABTASTEN SEINER UMGEBUNG EIN GEBILDE AN DEM ER EMPORKLETTERN KANN.

Verhalten an vertikalen Flächen

Befindet sich der Winkel zwischen 150 Grad und 30 Grad, wird das Verhalten für vertikale Flächen ausgeführt. Das Verhalten, das für diesen Fall implementiert wurde, ist, dass Kletterpflanzen normalerweise in die Höhe wachsen möchten und nicht etwa nach unten. Daher wird geprüft, ob die Kletterpflanze hier durch die Richtung des Pfades momentan nach unten wachsen würde, also der Winkel zwischen der Vorwärtsrichtung und dem Down-Vektor der Spielszene unter 90 Grad liegt. Sollte das der Fall sein, berechnet das Verhalten, welche Richtung, also links oder rechts, sich besser dafür eignet, wieder nach oben zu wachsen. Um das zu überprüfen wird jeweils für die positive und negative maximale Winkeländerung ein Richtungsvektor berechnet und dessen Winkel im Verhältnis zum Down-Vektor der Szene bestimmt. Je nachdem, welcher der beiden Winkel kleiner ist, wird entweder die positive oder negative maximale Winkeländerung als Winkeltendenz des Kletterpflanzenverhaltens zurückgegeben. Sollte der Winkel bei beiden gleich sein, wird zwischen den Beiden zufällig ausgewählt und falls die Richtung gar nicht nach unten zeigt, wird als Tendenz null zurückgegeben.

Verhalten Kopfüber

Fällt der Winkel weder in den Wertebereich einer vertikalen Fläche noch in den einer horizontalen Fläche, also die Normale der Fläche mehr oder weniger nach unten zeigt, wird die Oberfläche als kopfüber angesehen. In diesem Fall versucht der Pfadfindungsalgorithmus den Pfad so gut wie möglich zurück zu der vorherigen Fläche zu steuern. Für die Kletterpflanzen in der Anwendung macht es wenig Sinn, über längere Zeit an Flächen wie etwa den Unterseiten von Tischen und anderen Objekten zu wachsen, da diese zum Beispiel entweder nicht sehr oft von dem Spieler gesehen werden oder sehr häufig ein unvollständiges Mesh besitzen, dass unter Umständen Löcher oder andere Probleme haben kann. Um zurück zu der vorherigen Fläche zu gelangen, werden, ähnlich wie auf vertikalen Flächen, die Winkel der Richtungen, die sich aus der maximalen Winkeländerung der Kletterpflanze ergeben, mit einem weiteren

Richtungsvektor verglichen. Dieser weitere Richtungsvektor zeigt in Richtung der letzten bekannten Node des Pfades, die auf der vorherigen Fläche liegt. Zwischen den beiden erhaltenen Winkeln wird der Winkel gewählt, der kleiner ist, und anschließend die entsprechend maximale Winkeländerung der Kletterpflanze, entweder negativ oder positiv, als Winkeltendenz gesetzt. Bevor diese an die Pathfinding-Komponente zurückgegeben werden kann, wird die Tendenz zusätzlich noch mit einem zufälligen Wert zwischen 0.7 und 1.0 multipliziert. Wie in Abbildung 26 zu sehen ist, kann damit das Bilden von einheitlichen Kurven und Kreisen in der allgemeinen Form der Kletterpflanze vermieden werden.

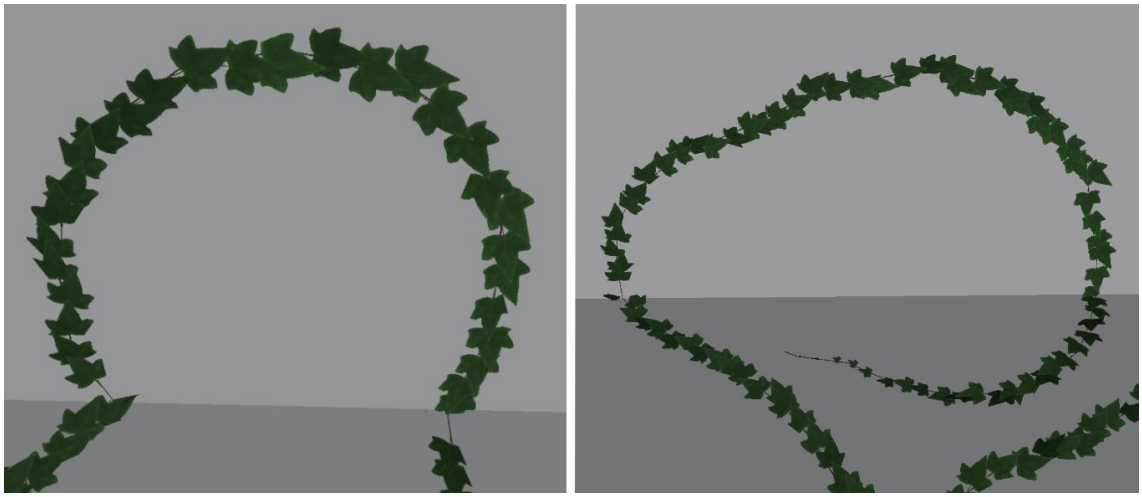

ABBILDUNG 26: VERGLEICH DER FORM DER KLETTERPFLANZE BEI DEM VERHALTEN KOPFÜBER. LINKS IST DAS VERHALTEN OHNE UND RECHTS MIT DER MULTIPLIKATION MIT EINEM ZUFÄLLIGEN WERT DARGESTELLT.

Erstellen des Meshes

Um den Efeu an den gefundenen Pfad anzupassen, wird dessen Mesh komplett prozedural durch das Ivy-Skript erstellt. Das Mesh des Efeus besteht dabei aus zwei separaten Stücken, dem Körper bzw. dem Stamm der Pflanze und der Spitze, welche durch ein eigenes Game-Object dargestellt wird, das dem Efeu-Objekt als Kind-Objekt zugeordnet ist. In der Update-Funktion, die in jedem Tick der Anwendung ausgeführt wird, prüft das Skript, ob der Efeu gerade wächst und falls er das nicht tut, wird weiter geprüft ob der Wachstumspfad des Efeus bereits vollständig abgearbeitet ist. Sollten sich noch unerreichte Nodes im Pfad befinden und der Efeu wächst momentan nicht, wird das Wachstum zur nächsten Node des Pfades eingeleitet.

Für das Erstellen des Meshes der Spitze und des Stammes des Efeus werden zuerst mehrere Vertices festgelegt, die als allgemeine Vorlage für das Mesh hergenommen werden. Diese Vertices werden als Schablone für die beiden Meshes genutzt und sind abhängig von der Auflösung und dem Radius des Efeu-Meshes, welche in den Game-Settings festgelegt sind. Die Vertices werden auf einem Kreis mit dem festgelegten Radius und in einem gleichmäßigen Abstand, der sich durch die Auflösung berechnen lässt, verteilt. Anschließend wird das Erstellen des Meshes der Spitze begonnen, indem die Vertices der Schablone kopiert und zusammen mit einem weiteren Vertex, der etwas weiter vorne liegt, platziert werden. Durch das Verbinden der Vertices der Spitze in der richtigen Reihenfolge werden die einzelnen Polygone des Meshes definiert und danach die Normalen berechnet. Das Berechnen der Normalen für das erstellte Mesh wird durch eine Funktion gehandhabt, die standartmäßig in Unity enthalten ist. Bei der Definition der Polygone muss in Unity darauf geachtet werden, dass die Vertices der einzelnen

Polygone im Uhrzeigersinn angeordnet sind, um die Richtung des Dreiecks richtig festzulegen. In das Mesh des Stammes werden zu Beginn nur zweimal die Vorlagevertices eingefügt, entsprechend rotiert und zu Dreiecken verbunden, um das Erstellen des Meshes auf den Wachstumsprozess vorzubereiten.

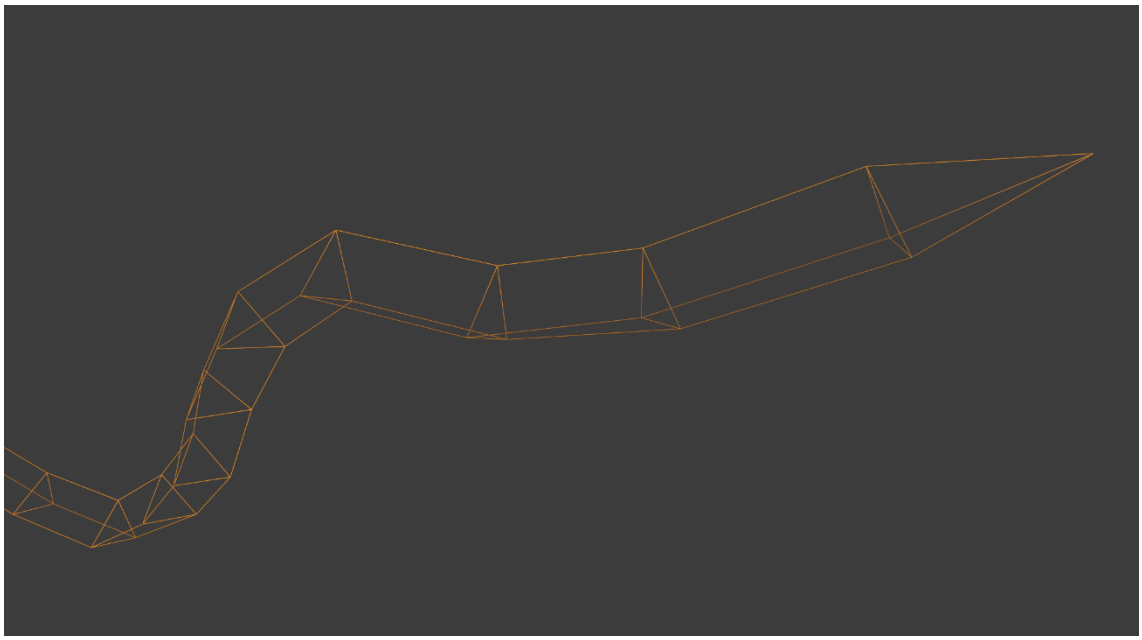

ABBILDUNG 27: DAS WIRE-MESH DES STAMMES EINES EFEUS MIT AUFLÖSUNG 3.

Sobald der Efeu das Wachsen beginnen soll, fängt das Pathfinding-Skript an, einen Pfad zu suchen und die Spitze beginnt, sich mit einer in den Game-Settings festgelegten Geschwindigkeit zur nächsten Node des Pfades zu bewegen. Währenddessen zeigt diese stehts in Richtung des Zielpunktes, auf den sie momentan zuwächst. Bei jedem Tick werden die letzten, in das Mesh des Stammes eingetragenen, Vertices mit der Position der hinteren Vertices der Spitze aktualisiert, um das Wachstum des Efeus darzustellen. Die Anzahl der Vertices, die hier aktualisiert werden, ist abhängig von der Auflösung des Meshes. So werden etwa bei einer eingestellten Auflösung von sechs nur die letzten sechs Vertices des Meshes aktualisiert. Wenn sich die Spitze in einem festgelegten Radius um die nächste Node befindet, wird diese Node als erreicht angesehen und die Position der Spitze auf die Position der Node gesetzt. Die Rotation der Spitze berechnet sich an dieser Stelle durch die Richtungsvektoren von der Letzten zur aktuellen Node und von der aktuellen Node zur Nächsten, dadurch werden unschöne „Brüche“ im Mesh verhindert. Wird eine Node erreicht, werden die letzten Vertices noch einmal aktualisiert, um sie auf ihre finale Position zu bringen und danach wird dem Stamm-Mesh ein neuer Satz an Vertices hinzugefügt, welcher mit den alten Vertices zu Polygonen verbunden wird. Dieser Ablauf wiederholt sich anschließend von Punkt zu Punkt, bis das Ende des Pfades erreicht wurde.

Während der Efeu wächst, werden zudem auch noch Blätter entlang des Meshes platziert. Dafür ist ein minimaler und maximaler Abstand in den Game-Settings definiert, zwischen denen ein zufälliger Wert ausgewählt wird. Bei jedem Tick wird die Distanz seit dem letzten Blatt überprüft und falls der ausgewählte Zufallswert erreicht wurde, wird ein neues Blatt platziert. Das Blatt besitzt hierfür ein Prefab, von dem eine Instanz bei der Object-Pooling-Komponente angefragt wird. Dieser Instanz des Blattes wird eine zufällige Rotation zugewiesen und dadurch erreicht, dass die Blätter unterschiedlich wirken und Variation zwischen den einzelnen Efeupflanzen entsteht. Da es bei dieser Art des Platzierens zu überstehenden Blättern an harten

Kanten des Spatial Mapping-Meshes kommen kann, wird zusätzlich überprüft, ob die Node, auf die der Efeu gerade zuwächst, eine Kante ist. Sollte das der Fall sein, wird sichergestellt, dass nur ein Blatt platziert werden kann, wenn die aktuelle Position noch nicht zu nahe an der Kante ist.

Des Weiteren kann der Efeu auch während des Wachstumsprozesses bei jeder erreichten Node des Pfades abzweigen. Hierfür ist eine Wahrscheinlichkeit festgelegt, auf die bei dem Erreichen jeder Node durch das Auswählen einer Zufallszahl zwischen 0.0 und 1.0 geprüft wird. Diese Prüfung findet jedoch nur statt, wenn bereits eine bestimmte Anzahl an Punkten des Pfades abgearbeitet wurden. Das Verhindert, dass direkt am Anfang oder bei sehr kurzen Efeupflanzen bereits eine Abzweigung entsteht. Sollte an einer Node eine Abzweigung entstehen, wird an dieser Stelle eine neue Instanz des Efeus platziert, die die noch übrige Anzahl an Iterationen des Eltern-Efeus als eigene maximale Iterationsanzahl übernimmt. Durch das Übernehmen der verbleibenden Iterationen wird verhindert, dass unrealistisch lange Abzweigungen kurz vor dem Ende einer Efeupflanze entstehen. Die Wachstumsrichtung dieser Instanz wird zufällig links oder rechts und in einem Winkel zwischen 10 Grad und 45 Grad festgelegt. Bei der neu platzierten Efeupflanze wird zudem die Möglichkeit des Abzweigens ausgestellt, da die Anzahl der Efeu-Instanzen in der Szene sonst sehr schnell unkontrolliert und unvorhersehbar wachsen kann.

6 OPTIMIERUNG DER ANWENDUNG

Wie schon mehrmals erwähnt, stellt die Performance einen sehr wichtigen Aspekt der Anwendung dar, da eine hohe Bildwiederholungsrate bei AR- und VR-Anwendung ausgesprochen stark zu einer guten Erfahrung des Spielers beiträgt. Vor der ersten Implementation und allgemein während der Entwicklung wurde schnell klar, dass verschiedene Funktionen und die gesamte Anwendung besser optimiert werden müssen, um dieses Ziel zu erreichen.

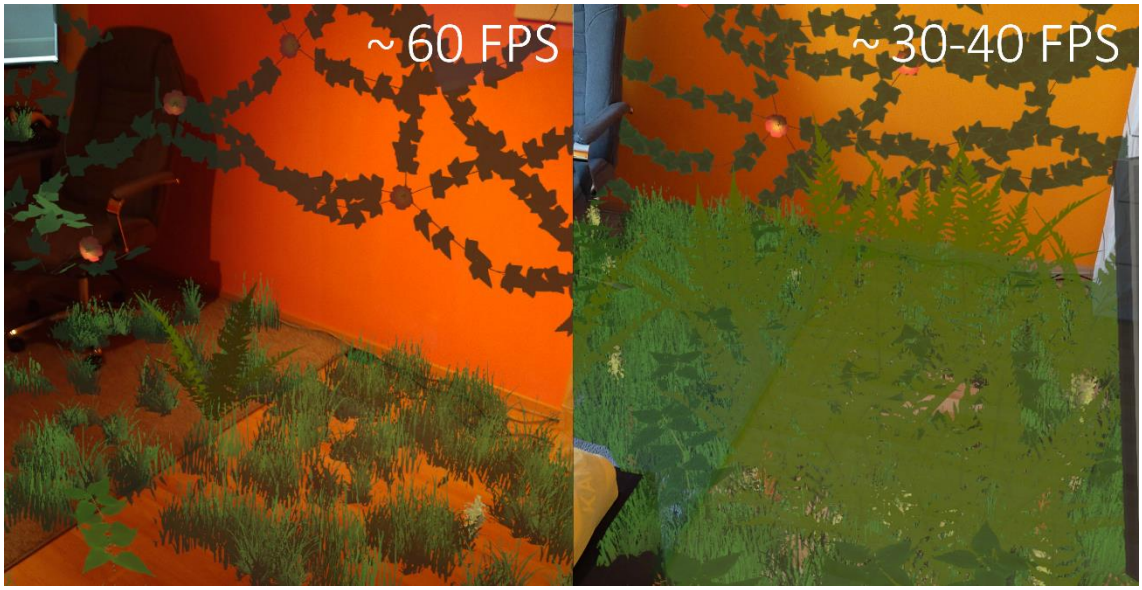

ABBILDUNG 28: VERGLEICH DER PERFORMANCE EINER SZENE MIT DÜNN VERTEILTER VEGETATION (LINKS) UND EINER SZENE MIT DICHTERER VEGETATION (RECHTS).

6.1 OPTIMIERUNG BEIM RENDERN

In der Spielszene der Anwendung sind, in verschiedenen Bereichen, mehrere Verbesserungen für das effizientere Rendern der Szene vorgenommen worden. Bei den Materialien der einzelnen Pflanzenmodelle der Spielszene wurde darauf geachtet, keine großen Texturen zu nutzten und eine weitere Verbesserung ist das Verringern der Anzahl der sogenannten Draw-Calls in Unity. Draw-Calls sind grob gesagt Aufforderungen der Unity Engine, ein Objekt bzw. Meshes in der Szene zu rendern. Laut der Dokumentation[13] der Unity Engine, können diese im Vergleich sehr aufwendig sein, wenn zwischen verschiedenen Draw-Calls Zustandsänderungen ausgeführt werden müssen. (Unity, 2020). Aus diesem Grund macht es Sinn, Draw-Calls der gleichen Art zu gruppieren. Unity bietet für das direkte Gruppieren von Draw-Calls zwei Mechanismen an, nämlich das Dynamic-Batching und das Static-Batching. Das Dynamic-Batching wird vollkommen automatisch ausgeführt und wird in der Anwendung verwendet, aber kann nur die Game-Objects der Szene in einem Draw-Call zusammenfassen, die bestimmte Eigenschaften besitzen. Für diese Anwendung sind hauptsächlich zwei dieser Eigenschaften relevant und laut Unity[13] sind diese wie folgt festgelegt: Das Game-Object darf nicht zu komplex sein bzw. zu viele Vertices besitzt (bis zu 900 Vertex Attribute und nicht mehr als 300 Vertices) und zum anderen müssen die einzelnen Objekte das gleiche Material zugewiesen bekommen haben. (Unity, 2020). Der Großteil der Objekte des Spiels erfüllt die erforderliche Grenze der Komplexität, da diese wenige Vertices besitzen, und es wurde nur eine Handvoll an Materialien für die Objekte der Pflanzen verwendet, indem die Texturen der einzelnen Pflanzen, in Hinsicht auf die benötigten Eigenschaften (einseitiges oder zweiseitiges Rendern, transparent oder opak), zu großen Texturen für die Materialien kombiniert wurden. Static-Batching besitzt zwar weniger Einschränkungen als das Dynamic-Batching, aber findet jedoch keine Verwendung bei der entwickelten Anwendung, da alle Objekte der Spielszene zur Laufzeit erstellt und platziert werden. Das macht das Nutzen des Static-Batchings unmöglich, da die Objekte hierfür während der Entwicklung im Editor als statisch markiert werden müssen. Die Anzahl der Draw-Calls kann zudem auch durch das sogenannte GPU-Instancing verringert werden, wenn der verwendete Shader des Materials diese Funktion unterstützt. Bei der Verwendung von GPU-Instancing werden Objekte der Szene, die ein Mesh besitzen, dass sich nur in der Position, Rotation, Skalierung und Farbe von anderen Objekten der gleichen Art unterscheidet, gruppiert und das Mesh nur einmal an die Grafikkarte gesendet. In der Anwendung wird GPU-Instancing für alle verwendeten Materialien aktiviert, da die meisten Objekte des Spiels dafür geeignet sind und die Performance davon besonders stark profitiert.

Um die Performance weiter zu verbessern, werden die verschiedenen Meshes der Pflanzeninstanzen, je nach Material, zusätzlich zu mehreren großen Meshes kombiniert. Das macht zwar auf den ersten Blick die Optimierungen der Performance, die durch Nutzen von Dynamic-Batching und GPU-Instancing erreicht wurden, zunichte, jedoch hat sich hingegen beim Testen davon gezeigt, dass damit bei der Anwendung eine etwas höhere Bildrate erzielt wird.

Die Blätter der Efeupflanzen werden hierbei pro Efeu Stück für Stück zu einem einzelnen Mesh zusammengefasst, sobald sie vollständig gewachsen sind. Um zu wissen, welche Blätter des Efeus bereit für das Kombinieren sind, besitzt das Ivy-Skript jedes Efeus eine Queue, in der sich die, für das Kombinieren bereiten, Blätter befinden. Jedes Blatt hat in dem Skript, dass das Wachstum des Blattes regelt, eine Funktion, die es in diese Queue einreiht, sobald das maximale Wachstum erreicht wurde. Der Efeu überprüft nach einem vorgegebenen Zeitintervall, ob sich bereits Instanzen von Blättern in der Warteschlange für das Kombinieren

[13] Unity Documentation: Draw call batching - https://docs.unity3d.com/2019.3/Documentation/Manual/DrawCallBatching.html

befinden. Sollte das der Fall sein, werden diese Blätter aus der Queue entfernt und an eine Funktion des Mesh-Combiners weitergegeben. Diese Funktion holt sich das Mesh jedes übergebenen Objekts und kombiniert diese. Im Anschluss wird das neue Mesh einem Kind-Objekt des Efeus zugewiesen. Für den Fall, dass dieses Objekt schon ein Mesh aus bereits kombinierten Meshes von Blättern besitzt, wird das bereits vorhandene Mesh auch in das neu erstellte Mesh eingefügt.

Bei dem Zusammenfassen der Vegetation auf dem Boden und anderen horizontalen Flächen wird etwas anders vorgegangen. Die Pflanzen, die garantiert sehr oft in der Szene vorkommen, haben alle das gleiche Material zugewiesen und könnten theoretisch zu einem einzigen, riesigen Mesh zusammengefasst werden, jedoch würde sich das aufgrund der Komplexität des erstellten Meshes negativ auf die Performance auswirken. Stattdessen wird die Spielszene in einzelne, quadratische Bereiche aufgeteilt und nur die Pflanzen innerhalb jedes Bereichs miteinander kombiniert. Die Breite und Länge der Bereiche sind auf einen Meter festgelegt, da damit ein gutes Ergebnis beim Testen erreicht wurde. In diesen Bereichen werden alle Bodenpflanzen kombiniert, außer die Brennnesseln und der Farn, da diese beiden Pflanzen normalerweise nicht außergewöhnlich häufig in der Szene vorkommen. Aus diesem Grund ist es für die Performance besser, bei diesen Pflanzenarten weiter GPU-Instancing und Dynamic-Batching zu verwenden.

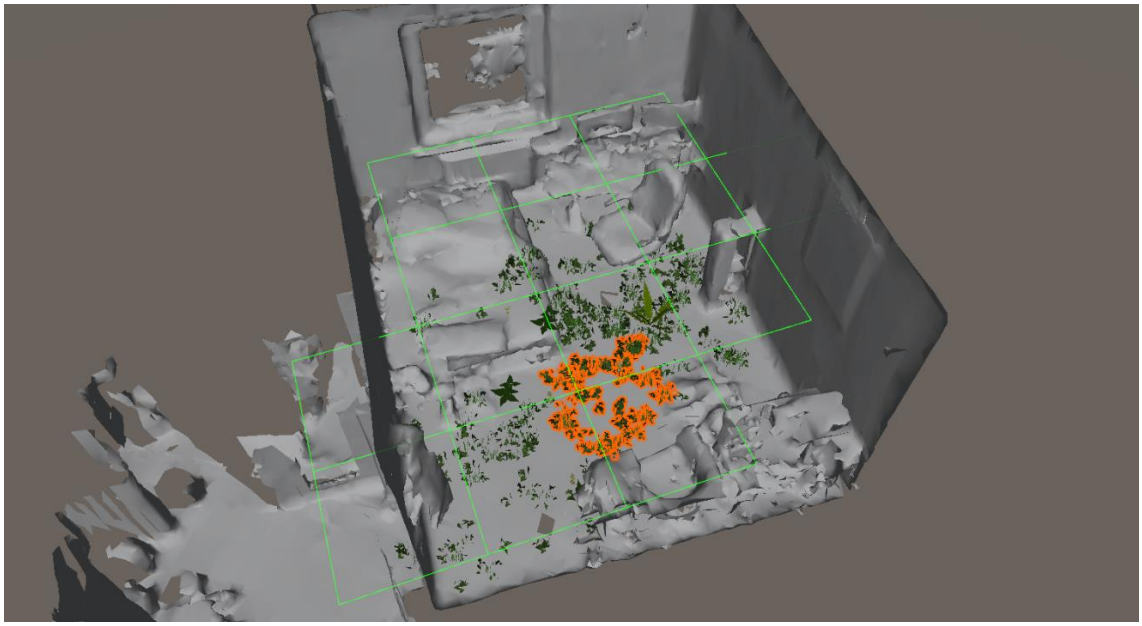

ABBILDUNG 29: EINTEILUNG DER SPIELSZENE IN EINZELNE BEREICHE, HIER IN GRÜN MARKIERT.

Das Aufteilen der Spielwelt in die einzelnen Bereiche wird durch den Mesh-Square-Manager geregelt, welcher eine Liste an Bereichen in der Form der Klasse *MeshSquare* führt, in denen sich platzierte Pflanzen befinden. Sobald eine der Pflanzen platziert wurde, wird beim Mesh-Square-Manager der dazugehörige Bereich angefragt und falls an der Stelle der Pflanze noch kein Bereich definiert wurde, wird ein neuer Bereich angelegt und in die Liste eingefügt. Sobald die Pflanze komplett gewachsen ist, wird ihre Unity-Komponente Mesh-Filter, die das Mesh verwaltet, ähnlich wie bei den Blättern des Efeus, in eine Liste des Bereichs eingetragen, in der sich alle Mesh-Filter der ausgewachsenen Pflanzen des Bereichs befinden. Der Mesh-Square-Manager löst in einem vorgegebenen Intervall, für alle Bereiche in seiner Liste, das Überprüfen ob bereits Meshes von Pflanzen in dem jeweiligen Bereich kombiniert werden können aus. Sollte das der Fall sein, gibt der Bereich die einzelnen Mesh-Filter an den Mesh-Combiner weiter, welcher die Meshes, wie bei den Efeublättern, kombiniert und an ein, dem Bereich

zugeordnetes, Game-Object anhängt. Der Nachteil hiervon ist, dass das Abstandhalten zu und zwischen den bereits kombinierten Pflanzen nicht mehr auf die ursprüngliche Weise machbar ist. Das ist ein weiterer Grund, wieso das Ganze nicht bei den Brennnesseln und dem Farn gemacht wird. Bei den Modellen für Gras und Unkraut fällt das Überschneiden nicht stark auf, jedoch bei den Pflanzenarten, die etwas größere Modelle haben und im Idealfall eigenen Platz benötigen, schon.

6.2 OPTIMIERUNG DES CODES

Abgesehen von den Optimierungen für das Rendern der Spielszene, wurden auch Optimierungen an dem Code der Anwendung durchgeführt. Ein Großteil der Optimierungen hierbei ist das Auslagern von Funktionen, die Code normalerweise in jedem Tick des Spiels ausführen, in Coroutinen. Bei Coroutinen kann der Funktionsablauf an bestimmten Stellen unterbrochen und im nächsten Tick der Anwendung bzw. nach einer gewissen Zeit, weiter ausgeführt werden. Viele der Funktionen in den Skripten dieser Anwendung müssen nicht jeden Tick ausgeführt werden und Coroutinen ermöglichen zusätzlich das kontrollierte Ausführen von Funktionen über einen bestimmten Zeitraum hinweg. Das benötigt weniger Leistung auf einmal und ist von Vorteil, falls das Ergebnis der spezifischen Funktion nicht sofort gebraucht wird. Ein gutes Beispiel hierfür ist das Pfadfinden des Efeus, vor allem wenn viele Pfade gleichzeitig in der Spielszene gefunden werden müssen. Um den nächsten Punkt des Pfades zu finden, wird bis zu acht Mal ein Raycasts durch die Szene geschossen, was bei einer gut verteilten Ausführung der Funktion kein Problem darstellt. Ohne das Auslagern wie bei Coroutinen würden diese Raycasts für jede Instanz des Efeus und für jeden Tick der Anwendung ausgeführt werden, was sich bei dem gleichzeitigen Pfadfinden durch mehrere Kletterpflanzen schnell addiert und die Performance einschränkt.

```
IEnumerator GrowGrass()
{
    mMaxScale.y *= Random.Range(0.5f, 1f);

    Vector3 scale = mMaxScale;
    scale.y = 0f;

    bool fullyGrown = false;
    gameObject.transform.localScale = scale;
    float increase = (timeInterval / timeToScale);
    while (!fullyGrown) //Hier startet die Coroutine wieder
    {
        if (scale.y < mMaxScale.y)
        {
            scale.y += increase * mMaxScale.y;
        }

        if (scale.x >= mMaxScale.x && scale.y >= mMaxScale.y && scale.z >= mMaxScale.z)
        {
            scale = mMaxScale;
            fullyGrown = true;  //Beende die While-Schleife und
                                //somit das Pausieren und Weiterausführen der Coroutine
        }

        gameObject.transform.localScale = scale;
        //Warte für eine bestimme Anzahl an Sekunden (z.B. 0.05)
        yield return new WaitForSeconds(timeInterval);  //Coroutine "pausiert" die Ausführung an dieser Stelle
    }

    HasFullyGrown(1);
}
```

ABBILDUNG 30: DIE COROUTINE FÜR DIE WACHSTUMSANIMATION VON GRAS MIT KOMMENTAREN DIE DEN ABLAUF ERKLÄREN.

Ein weiteres Beispiel hierfür sind die Wachstumsanimation der einzelnen Pflanzenarten. Diese würden die Skalierung der Pflanze normalerweise auch in jedem Tick des Spiels aktualisieren, was nicht unbedingt nötig ist, um eine glaubhafte Animation für das Wachstum der Pflanzen zu erschaffen. In der Szene können sich mehrere dutzend Pflanzen befinden, die zur gleichen Zeit die Wachstumsanimation ausführen, deshalb wird hier durch den Einsatz von Coroutinen versucht, die Aktualisierungen der Skalierung in größeren Zeitabständen durchzuführen, ohne dabei die Glaubhaftigkeit zu verlieren.

Der Code und die Performance der Szene wurden des Weiteren durch die Implementierung eines Object-Pooling-Systems verbessert. Unter dem Begriff „Object-Pooling“ ist das gebündelte Instanziieren der Spielobjekte zum Spielbeginn zu verstehen, um diese in sogenannten *Pools* in einem inaktiven Zustand zu speichern. Durch das Object-Pooling-System werden die Instanzen nicht individuell zur Laufzeit erstellt und können einfach bei der ObjectPooling-Komponente angefragt werden. In der Anwendung wird dieses gebündelte Erstellen insbesondere bei den Blättern des Efeus und den häufig vorkommenden Modellen von Gras und Unkraut benötigt, um schwerwiegende Leistungseinbrüche der Anwendung zur Laufzeit zu vermeiden. Ohne das Object-Pooling-System würde die Bildrate der Anwendung während dem Platzieren dieser Pflanzen stark variieren und insgesamt niedriger sein. Für das Bündeln der Objekte ist die Komponente *ObjectPooling* verantwortlich, auf die über den Game-Manager zugegriffen werden kann. Um die einzelnen Object-Pools zu speichern und zu verwalten, wird eine Wörterbuch-Datenstruktur genutzt, die als Schlüssel den Namen der Pflanzenart verwendet und die dazugehörigen Instanzen der Pflanzen in der Form einer Queue von Game-Objects abspeichert. Da Wörterbücher den gewünschten Eintrag sehr schnell durch den gegebenen Schlüssel finden können, kann ein gesuchtes instanziiertes Objekt schnell und effizient angefragt werden. Für das Speichern der benötigten Daten der Pflanzenarten wird die Klasse *Pool* verwendet. Diese Klasse speichert den Namen der Pflanzenart, eine Liste ihrer möglichen Modelle bzw. Prefabs und die Anzahl an Instanzen dieser Pflanzenart, die angelegt werden sollen, ab. Um diese Parameter leicht finden zu können, legt das ObjectPooling-Skript für die einzelnen Pools ein Wörterbuch an, dass auch den Namen der Pflanzenart als Schlüssel besitzt und auf die einzelnen Pools verweist. Sobald die Spielszene geladen wird, fängt das Object-Pooling an, die einzelnen Pools zu erstellen. Die benötigten Daten hierfür werden aus der Pflanzenliste der Game-Settings geholt und in den Parametern der separaten Pools abgespeichert. Wenn das Erstellen der Pools abgeschlossen und damit die Anzahl der benötigten Instanzen bekannt ist, wird damit begonnen, die Queue der eigentlichen Objekt-Pools zu füllen. Um einen Objekt-Pool zu füllen, wird ein zufälliges Prefab aus der Liste des Pools ausgesucht, eine Instanz davon erstellt, bei der der Status des erstellten Game-Objects auf inaktiv gestellt wird, und diese in die Liste der Instanzen des Objekt-Pools eingefügt. Das Ganze wird solange wiederholt, bis die gewünschte Anzahl erreicht ist. Andere Komponenten der Anwendung können dann über den Namen der Pflanzenart ein Game-Object anfragen und das ObjectPooling-Skript sucht den entsprechenden Pool, nimmt das erste Game-Object in der Queue, setzt dessen Status auf aktiv und gibt die Instanz der Pflanzenart an die anfragende Komponente zurück. Wenn ein anderes Skript eine Instanz anfragt, wird zusätzlich geprüft, ob sich noch genug Objekte in dem entsprechenden Object-Pool befinden. Falls nicht, wird dieser wieder mit einer bestimmten Anzahl an Objekten gefüllt. Der Grenzwert, ab dem Nachgefüllt wird, berechnet sich hierbei aus der ursprünglichen Anzahl an Instanzen, mit denen der Pool zu Beginn der Anwendung gefüllt wurde, und einem festgelegten Multiplikator, der in den Game-Settings zu finden ist. Für das Festlegen der Anzahl an Objekten, mit denen der Pool aufgefüllt wird, wird auch ein Multiplikator für den Nachfüllwert genutzt, der ebenso mit dem ursprünglichen Wert verrechnet wird. Beim Auffüllen des Object-Pools könnte man verschiedene Ansätze nutzen, wie zum Beispiel das Auffüllen über einen vorgesetzten Zeitraum hinweg. Dieser Ansatz würde jedoch zu einer niedrigeren Framerate während des gesamten

Auffüllvorganges führen, daher werden die Pools in dieser Anwendung stattdessen direkt mit allen benötigten Instanzen aufgefüllt.

7 AUSBLICK UND DISKUSSION

Allgemein ist die erstellte Anwendung in der Lage, die prozedural generierte Vegetation überzeugend und interessant darzustellen, zudem wurden die Ziele für die prozedurale Generation in dieser Arbeit alle erreicht. Die einzelnen Pflanzen sehen in ihrer Gesamtheit glaubwürdig aus und führen dazu, dass der Raum eine überwachsene Atmosphäre bekommt. Die Pfadfindung des Efeus hat sich in mehreren Tests während der Entwicklung als sehr robust gezeigt und das Wachstum des Efeus sollte sich damit in den meisten Fällen gut an die Umgebung anpassen können. Die Pflanzen, die an horizontalen Flächen platziert werden, berücksichtigen ihre Umgebung auch und passen sich dementsprechend an diese an.

ABBILDUNG 31: EINE WEITERE SZENE DER ANWENDUNG AUS DEN AUGEN DES SPIELERS.

Wie gut das Definieren eines zulässigen Bereichs und das Platzieren der Pflanzen in diesem in einer Vielzahl von verschiedenen Räumen funktioniert, ist schwer zu sagen. Grundsätzlich sollten die meisten Räume dafür geeignet sein, jedoch kann diese Aussage nur durch das Testen der Anwendung in vielen Räumen mit stark variierenden räumlichen Aspekten (Grundriss, Möbel, freie Oberflächen, etc.) tatsächlich bestätigt werden. Leider konnte die Anwendung nicht durch eine signifikante Anzahl an anderen Personen direkt getestet werden, um dazu eine unabhängige Meinung einzuholen. Nur eine Handvoll an Personen hatten die Möglichkeit, die Anwendung während der Entwicklung direkt auf der HoloLens 2 und durch die Präsentation von Videos auszuprobieren und zu begutachten. Alle beteiligten Tester fanden es interessant, einen Raum zu beobachten, der langsam von den verschiedenen Pflanzenarten überwachsen und durch die Natur erobert wird. Die Meisten davon waren vor allem von der Anpassung des Efeus an den Raum und dessen Wachstum begeistert und haben das als ausgesprochen glaubhaft empfunden. Viele der Beteiligten fanden jedoch, das ihnen die Bildrate der Anwendung negativ aufgefallen ist, sobald die Vegetation innerhalb des Raumes sehr dicht wurde.

7.1 AUFGETRETENE PROBLEME UND MÄNGEL DER ANWENDUNG

Die entwickelte Anwendung ist nicht fehlerfrei und es gibt an manchen Stellen Raum für Verbesserung. Bei den verschiedenen Pflanzenarten wurde während der Entwicklung hauptsächlich darauf geachtet, dass diese gut aussehen und sich glaubwürdig verhalten. Die implementierten Pflanzenarten sind daher, im Vergleich zu ihren realen Gegenstücken und in Hinsicht auf die Botanik, nicht sehr akkurat abgebildet. Ohne Vorwissen und Erfahrung in den Bereichen der Botanik und dem Entwickeln von 3D-Anwendungen auf mobilen Geräten, wie der HoloLens 2, hat es sich bei der Entwicklung der Anwendung als äußerst schwer erwiesen, eine gute Balance zwischen dem Detailgrad und Exaktheit der einzelnen Pflanzen, sowie der endgültigen Performance der Anwendung zu finden. Das Wachstum der Pflanzen kann zusätzlich nicht direkt durch die Physik oder den Spieler beeinflusst und kontrolliert werden. Des Weiteren werden, um die Performance der fertigen Anwendung weiter zu verbessern, die Pflanzen auf horizontalen Flächen in einzelnen Bereichen zusammengefasst, was dazu führt, dass die bereits kombinierten Pflanzen beim weiteren Platzieren und Abstandhalten von existierenden Pflanzen nicht mehr in Betracht gezogen werden.

Wie schon erwähnt, ist das Finden des zulässigen Bereichs nicht optimal implementiert. Der Weg, den für das Platzieren zulässigen Bereich durch ein Nav-Mesh zu definieren, funktioniert zwar bei dieser Anwendung erstaunlich gut, aber dadurch wird das Platzieren der Pflanzen auch auf viele Weisen stark eingeschränkt. Zum Beispiel müssen die einzelnen Pflanzenarten in Größenkategorien eingestuft werden, was ein großes Problem darstellen kann, wenn viele verschiedene Arten von Pflanzen implementiert werden oder einzelne Pflanzenarten stark in ihrer Größe variieren können.

Obwohl gegen Ende der Entwicklung versucht wurde, den Code und die Implementierung der Vegetation so gut wie möglich zu verbessern und zu optimieren, ist die endgültige Performance der Anwendung nicht so gut, wie ursprünglich vorgesehen. Je nach Raumgröße und Dichte, in der die Pflanzen platziert werden, fällt die Bildrate der Anwendung auf ungefähr 30 Bilder pro Sekunde. Die Anwendung ist aber mit dieser Bildrate immer noch leicht spielbar und gewisse Performanceprobleme wurden von Anfang an erwartet, da die Anwendung ohne vorherige Erfahrung im Bereich der Entwicklung für die HoloLens 2 und der prozeduralen Generation erstellt wurde.

Da bei der Entwicklung der Anwendung keine externen SDKs für das Entwickeln mit der HoloLens 2 verwendet wurden, haben sich dadurch auch einige Probleme ergeben. Das größte Problem hierbei ist die Qualität des erhaltenen Meshes der Umgebung des Spielers. Dieses Spatial Mapping-Mesh kann kleine und große Löcher enthalten und Oberflächen können sehr uneben sein. Die Gründe für diese beiden Mängel des Meshes können ein schlechtes Lichtverhältnis innerhalb des gescannten Raumes, unzureichendes Scannen durch den Spieler, spiegelnde Oberflächen von Objekten innerhalb des Raumes oder jede Kombination dieser drei Gründe sein. Oberflächen, die durch Fehler beim Abtasten des Raumes, uneben sind, stellen ein großes Problem für das Definieren eines zulässigen Bereichs dar und können dadurch auch zur Verwirrung des Spielers führen, wenn dieser versucht Pflanzen an einer, in der Theorie geeigneten, Stelle zu platzieren und die Anwendung das nicht zulässt, weil die Stelle nicht im zulässigen Bereich liegt. Beim Wachstum des Efeus können Löcher im Mesh zudem dazu führen, dass kein Pfad gefunden wird, weil der Algorithmus nicht wissen kann, ob es sich nur um ein Loch im Mesh oder einen Bereich des Raumes handelt, der noch nicht bzw. nicht richtig abgetastet wurde. Unebenheiten im Mesh äußern sich vor allem dadurch, dass der Efeu auf einmal nicht mehr weiterwächst und/oder starke knicke im Mesh zu sehen sind. Das führt auch dazu, dass der Efeu von manchen Blumen aus gar nicht oder nur teilweise wachsen kann. Auch

bei dem Verhalten des Efeus, wenn er sich an Flächen wie der Decke oder den Unterseiten von Objekten befindet, kommt es aufgrund der Qualität des Meshes zu Problemen. Es kann sein, dass der Pfadfindungs-Algorithmus eine Oberfläche als absteigend kategorisiert, wenn diese eigentlich in der Theorie eben sein sollte, und damit die vorherige Fläche für das Verhalten des Efeus überschrieben wird. Das führt dazu, dass der Efeu manchmal nicht zurück zur vorherigen Ebene finden kann und deswegen weiter an der Oberfläche wächst. Externe SDKs würden verschiedene Möglichkeiten bieten, diese Mängel, durch analysieren und weiteres verarbeiten des Meshes zu beseitigen und könnten daher bei zukünftigen Arbeiten Anwendung finden.

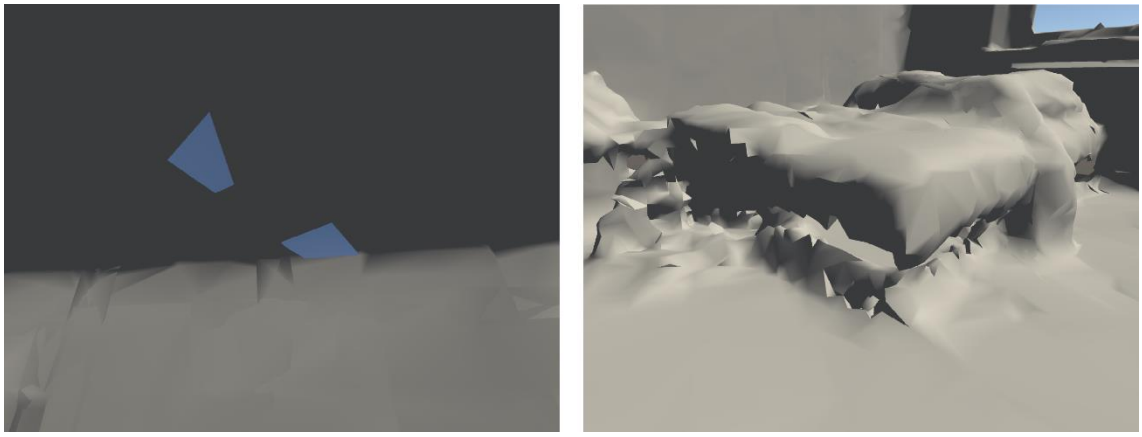

ABBILDUNG 32: BEISPIELE FÜR PROBLEME MIT DEM MESH DIE BEI DEM ABTASTEN DES RAUMES AUFGETRETEN SIND. LINKS SIND KLEINE LÖCHER IM MESH ZU SEHEN UND RECHTS RELATIV GROßE LÖCHER UND UNEBENE OBERFLÄCHEN AUFGRUND VON SPIEGELUNGEN.

7.2 ZUKÜNFTIGE ARBEITEN

In zukünftigen Arbeiten könnten viele dieser Probleme gelöst und die verschiedenen Prinzipien hinter der entwickelten Anwendung erweitert werden. Um die Performance zu verbessern und dem Nutzer das Interagieren mit den Pflanzen zu ermöglichen, kann ein ähnlicher Ansatz, wie bei der schon erwähnten Arbeit „Interactive Modeling and Authoring of Climbing Plants“ von Hädrich et al. (2017) Verwendung finden. Durch den Einsatz eines speziell erstellten Shaders für die Vegetation könnte die Leistung zudem weiter verbessert und vor allem Vegetation, wie etwa das Gras, einfacher dargestellt werden. Auch das Schreiben eines eigenen Algorithmus, der den zulässigen Bereich für die Pflanzen, durch analysieren des Spatial Mapping-Meshes der Umgebung des Spielers, definieren kann und primär für diesen Zweck vorgesehen ist, stellt ein weiteres Thema für eine Folgearbeit dar. Dadurch könnten die Flächen, auf denen die Anwendung die Pflanzen platziert nicht nur genauer definiert und kontrolliert, sondern auch individuell auf die einzelnen Pflanzenarten und ihre Größenverhältnisse und Wachstumsbedürfnisse angepasst werden.

Ein anderer Weg, das Ganze in einer zukünftigen Arbeit zu erweitern, ist die Verwendung der prozeduralen Generation in der Anwendung, um Spieleinhalte nicht zur Laufzeit, sondern während der Entwicklung zu erstellen, um sie dann in der fertigen Anwendung als normale Objekte zu nutzen. Wie schon in Kapitel 1 angeschnitten, wird prozedurale Generation häufig auf diese Weiße verwendet, damit die Arbeit der Entwickler erleichtert wird und Ressourcen gespart werden. Um die Pfadfindung des Efeus zu testen, wurde am Anfang der Entwicklung eine Testszene erstellt, die aus herkömmlicher Geometrie in der Form von vielen primitiven Unity-Objekten besteht, anstatt nur das Spatial Mapping-Mesh (welches so gesehen ein einziges, riesiges Mesh ist) zu benutzen. Falls die Option implementiert wird, zum Beispiel die generierten Kletterpflanzen in einem weit verbreiteten Format als 3D-Modelle zu exportieren,

ist es also schon in der Theorie möglich, die prozedurale Generation der Anwendung für diesen Zweck zu nutzen, aber natürlich würde hier noch ein gewisser Grad der Benutzerfreundlichkeit und Anpassbarkeit fehlen.

Die prozedurale Generierung von Vegetation kann zusätzlich auch interessant für Arbeiten in dem Bereich von Serious Games sein. Hier können zum Beispiel verschiedene Insektenarten mit einer künstlichen Intelligenz ausgestattet und in den Spielablauf implementiert werden. Diese modifizierte Version der Anwendung könnte dann unter Umständen in der Behandlung von verschiedenen Insekten-Phobien durch eine Expositionstherapie Verwendung finden. Eine weitere Anwendung im Bereich der Serious Games kann der Stressabbau bei Personen, die viel Zeit im Krankenhaus oder Zuhause verbringen müssen, sein. Es gibt viele Gründe, wieso eine Person ein Zimmer im Krankenhaus oder einen Raum in der eignen Wohnung nicht verlassen kann, dazu zählen nicht nur Krankheiten und Behinderungen. Viele gesunde Personen haben in den letzten Monaten, aufgrund der aktuellen Coronavirus-Pandemie, vermehrt Zeit in den eigenen vier Wänden verbracht, was unter Umständen zu Depression und anderen gesundheitlichen Folgen führen könnte. Eine abgeänderte Version der Anwendung dieser Arbeit könnte dafür genutzt werden, diesen Personen einen Weg zu geben, wenn auch nur virtuell, mit der Natur zu interagieren und eventuell bei der Vorbeugung der erwähnten gesundheitlichen Probleme zu helfen.

8 ZUSAMMENFASSUNG

Diese Arbeit beschreibt die Implementierung einer Augmented Reality-Anwendung, in der der Raum, in dem sich der Benutzer befindet, langsam durch prozedural generierte Pflanzen überwachsen wird. Die Anwendung wird mithilfe der Game Engine Unity und einem SDK namens Mixed Reality Tool Kit für Microsoft's HoloLens 2 AR-Brille entwickelt. Die prozedurale Generation orientiert sich dabei an den folgenden Zielen:

- Die verschiedenen Pflanzenarten können nur an Oberflächen des Raumes platziert werden, die für sie geeignet sind.
- Sollte die jeweilige Pflanzenart prozedural erstellt werden, muss diese ihre Umgebung bei der Erstellung berücksichtigen.
- Der Raum muss durch die generierte Vegetation auf eine glaubhafte Weise überwachsen wirken.

Mit Augmented Reality ist grundsätzlich das visuelle Überlagern von Informationen über die reale Welt gemeint. Die HoloLens 2 ist ein sehr hochwertiges AR-HMD, dass eine vergleichsweise hohe Leistung und viele Funktionen besitzt, die grundlegende Aspekte von AR erweitern. Dazu gehört nicht nur das Interagieren mit Hologrammen durch unterschiedliche Gesten, sondern auch das Erstellen eines Meshes durch Abtasten des Raumes, in dem sich der Nutzer befindet. Unity ist eine populäre Game Engine, die sich hervorragend für das Entwickeln von Anwendung für die HoloLens 2 eignet. Ein Grund dafür ist, dass das Mixed Reality Tool Kit, ein SDK, dass für das Entwickeln von AR, VR und Mixed Reality Anwendung vorgesehen ist, primär zur Entwicklung auf dieser Plattform verwendet werden kann.

Prozedurale Generation kann in vielen unterschiedlichen Gebieten Anwendung finden und es gibt deshalb auch verschiedene Definitionen. Für diese Arbeit wird die folgende Definition verwendet:

> Prozedurale Generation bezeichnet das Erstellen von Inhalten mithilfe von Algorithmen, ohne diese komplett manuell erstellen oder verändern zu müssen. Diese Inhalte werden

teilweise oder vollständig in Abhängigkeit von anderen Parametern und Eigenschaften der Spielwelt erstellt.

Das prozedurale Generieren von Inhalten kann zum Beispiel durch reine Simulation, das Zusammensetzen aus verschiedenen Bausteinen und durch formelle Grammatiken bzw. spezifisch festgelegte Regeln mithilfe eines Algorithmus erstellt werden. (Smith, 2015, S. 503 f.). Prozedurale Generation bietet dabei einige Vorteile gegenüber der herkömmlichen Inhaltserstellung. Dazu gehören größere Abwechslung im Spielverlauf (Smith, 2015, S. 501, Short & Adams, 2017, S. 9), vergleichsweise lebendig wirkende Inhalte (Short & Adams, 2017, S. 11) und Flexibilität in Hinsicht auf die Spielwelt.

Bei der Implementierung von Vegetation in 3D-Anwendungen gibt es viele Möglichkeiten, die Vegetation glaubwürdiger und performanter zu machen. Ein in der Industrie weit verbreiteter Ansatz, um die Flora von Videospielen leistungseffizient darzustellen, ist das Nutzen von transparenten Texturen, die auf einfache Ebenen gelegt werden. Um die Beleuchtung der Pflanzen bei dieser Modelliertechnik zu verbessern, können die Normalen angepasst werden. Dadurch wird zwar grundsätzlich auch keine korrekte Beleuchtung erreicht, jedoch wirkt das Modell dadurch für den Benutzer oft glaubwürdiger.

An das Anwendungsdesign der erstellten Anwendung werden verschiedene, allgemeine Anforderungen gestellt, die so gut es geht bei der Entwicklung erfüllt werden sollen. Zu diesen Anforderungen zählen unter anderem eine gute Performance, die Benutzerfreundlichkeit und die Anpassung an die verwendete Hardware. Bei dem Design der Anwendung wurden daher einige Punkte beachtet. Die Anwendung selbst sollte eine kurze Spieldauer haben, da die HoloLens 2 nicht kabelgebunden ist und der Akku des Geräts relativ schnell leer geht. Das vergleichsweise kleine Field-of-View, dass die HoloLens 2 besitzt, führt dazu, dass vermieden werden sollte, große, einfarbige Objekte in der Spielszene zu verwenden. Da die HoloLens 2 kabellos ist, ist die Rechenleistung auch eingeschränkt, was dazu führt das eine gute Balance zwischen Detailgrad und Performance gefunden werden muss. Für das Design der einzelnen Komponenten wurden auch einige Entscheidungen getroffen, wie etwa das alle Pflanzen der Anwendung entweder komplett prozedural erstellt oder zumindest prozedural platziert werden und die verschiedenen vertikalen und horizontalen Flächen gut überwuchern. Für vertikale Flächen wurden Kletterpflanzen ausgewählt, da diese sich gut an die Umgebung des Spielers anpassen können. Für die Kletterpflanzen ist daher die Implementation eines Algorithmus notwendig, der einen geeigneten Pfad suchen kann, dabei auf Hindernisse reagiert und spezielles Verhalten der Kletterpflanzen darstellt. Für das allgemeine Platzieren der Pflanzen muss darauf geachtet werden, dass diese auch nur auf Oberflächen platziert werden, auf denen das Platzieren Sinn macht. Für horizontale Flächen muss hier auf eine geeignete Weise ein zulässiger Bereich für das Platzieren definiert werden.

In der Implementierung werden die Pflanzenarten in Größenkategorien eingeteilt und ein Nav-Mesh für diese erstellt, mit welchem ein zulässiger Bereich definiert wird. Auf horizontalen Oberflächen können Gras, Unkraut, Brennnesseln und Farn wachsen, an den Vertikalen hingegen nur eine Blume, von der aus Efeu wachsen kann. Allen Pflanzenmodellen wurde zudem eine Wachstumsanimation gegeben, damit sie nicht aus dem Nichts auftauchen. Die Vegetation wird prozedural an horizontalen und vertikalen Flächen platziert, um sie an den Raum anzupassen. Das passiert zum einen durch den Spieler, der Wasserballons werfen kann, die das Wachstum von Pflanzen an der getroffenen Stelle auslösen, und zum anderen zusätzlich automatisch zu Beginn der Anwendung. Der Spieler hat insgesamt 60 Wasserballone zur Verfügung, um die höchstmögliche Anzahl an Pflanzen zu platzieren. Jede Pflanze gibt dem Spieler Punkte und er kann damit versuchen, einen Highscore aufzustellen. Der Efeu wird komplett prozedural erstellt, indem das Mesh entlang eines geeigneten Pfades generiert wird. Die Pfadfindung findet hierbei durch mehrere Raycasts statt, mit denen Hindernisse erkannt

werden können. Zusätzlich wurden in der Pfadfindung verschiedene Verhaltensweisen für die Kletterpflanzen implementiert.

An den einzelnen Komponenten der Anwendung wurde viel Optimierung durchgeführt, um eine akzeptable Performance zu erreichen. Die Materialien der Objekte wurden so angepasst, dass sie möglichst wenige Draw-Calls der Engine benötigen. Des Weiteren werden die einzelnen Pflanzen im Raum und die Blätter des Efeus jeweils zu mehreren größeren Meshes kombiniert. Ressourcenintensiver Code wurde, wenn möglich, in Coroutinen ausgelagert, um diesen über einen Zeitraum hinweg auszuführen, anstatt in einem einzigen Tick der Anwendung. Um die Last während des Spielverlaufs zu verringern, wurde zudem ein Object-Pooling-System implementiert, dass zum Spielbeginn einen Vorrat an benötigten Instanzen der Pflanzen anlegt und diesen Vorrat bei Bedarf wieder auffüllt.

Die fertige Anwendung hat einige Probleme und Mängel, wie zum Beispiel die Performance, die nach vielen Verbesserungen an der Anwendung trotzdem nicht so gut wie erhofft ist, oder die nicht optimale Implementierung des Findens eines zulässigen Bereichs, jedoch gibt es großes Potential für Folgearbeiten. Zukünftige Arbeiten könnten vor allem in der Inhaltsgeneration von Spielen während der Entwicklung und dem Bereich der Serious Games zu finden sein.

9 ABBILDUNGSVERZEICHNIS

10 LITERATURVERZEICHNIS

- Azad S., Saldanha C., Gan C. & Riedl M. (2016). Procedural Level Generation for Augmented Reality Games. The Twelfth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, pp. 247-249.

- Azuma R., Baillot Y., Behringer R. & Feiner S. (2001). Recent Advances in Augmented Reality. IEEE Computer Graphics and Applications, 21(6), pp. 34-47.

- Bidarra R., van der Linden R. & Lopes R. (2014). Procedural generation of dungeons, IEEE Transactions on Computational Intelligence and AI in Games, 6(1), pp. 78-89.

- Carmigniani J., Furht B., Anisetti M. & Ceravolo P. (2010). Augmented reality technologies, systems and applications. Multimedia Tools and Applications, 51(1), pp. 341-377.

- Gianoli E. (2015). The behavioural ecology of climbing plants. AoB Plants, 7, o.S.

- Hädrich T., Benes B., Deussen O. & Pirk S. (2017). Interactive Modeling and Authoring of Climbing Plants. Computer Graphics Forum, 36(2), pp. 49-61.

- Knutzen J. (2009). Generating Climbing Plants Using L-Systems.

- Microsoft (2018). (Aufgerufen am 14.12.2020). Microsoft, What is a hologram?. https://docs.microsoft.com/en-us/windows/mixed-reality/discover/hologram

- Microsoft (2019). (Aufgerufen am 14.12.2020). Microsoft, Getting started with MRTK for Unity. https://docs.microsoft.com/en-us/windows/mixed-reality/develop/unity/mrtk-getting-started

- Sanders G., GDC, (06.02.2020) Between Tech and Art: The Vegetation of Horizon Zero Dawn. Youtube, https://www.youtube.com/watch?v=wavnKZNSYqU (Aufgerufen am 14.12.2020)

- Short T. & Adams T. (2017). Procedural Generation in Game Design. Boca Raton, FL: CRC Press.

- Smith G. (2015). Procedural Content Generation: An Overview. In Rabin S. (Hrsg.), Game AI Pro 2, pp. 501-518. Boca Raton, FL: CRC Press.

- Unity (2020). (Aufgerufen am 14.12.2020). unity3d, Draw call batching. https://docs.unity3d.com/2019.3/Documentation/Manual/DrawCallBatching.html

- van Muijden J., GDC, (27.12.2019) GPU-Based Run-Time Procedural Placement in Horizon: Zero Dawn. Youtube, https://www.youtube.com/watch?v=ToCozpl1sYY (Aufgerufen am 14.12.2020)

11 ERKLÄRUNGEN

11.1 SELBSTSTÄNDIGKEITSERKLÄRUNG

Hiermit erkläre ich, dass ich die Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken als solche kenntlich gemacht habe.

Die Arbeit habe ich bisher keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher nicht veröffentlicht.

Ort, Datum Unterschrift

11.2 ERMÄCHTIGUNG

☐ Hiermit ermächtige ich/wir die Hochschule Kempten zur Veröffentlichung einer Kurzzusammenfassung sowie Bilder/Screenshots und ggf. angefertigte Videos meiner studentischen Arbeit z. B. auf gedruckten Medien oder auf einer Internetseite der Hochschule Kempten zwecks Bewerbung des Bachelorstudiengangs „Game Engineering“ und des Masterstudiengangs „Game Engineering und Visual Computing“.

Dies betrifft insbesondere den Webauftritt der Hochschule Kempten inklusive der Webseite des Zentrums für Computerspiele und Simulation. Die Hochschule Kempten erhält das einfache, unentgeltliche Nutzungsrecht im Sinne der §§ 31 Abs. 2, 32 Abs. 3 Satz 3 Urheberrechtsgesetz (UrhG).

Ort, Datum Unterschrift