HOCHSCHULE
MITTWEIDA
University of Applied Sciences

Fakultät Angewandte Computer- und Biowissenschaften

Professur Medieninformatik

Bachelorarbeit

Wave Function Collapse als Gamedesign Tool in der Unity Engine - Untersuchung zu Einsatzmöglichkeiten im Leveldesign

Johannes Bätz

Mittweida, den 29. November 2023

Erstprüfer: Prof. Dr. Christian Roschke
Zweitprüfer: Manuel Heinzig M. Sc.

Bätz, Johannes
Wave Function Collapse als Gamedesign Tool in der Unity Engine - Untersuchung zu Einsatzmöglichkeiten im Leveldesign
Bachelorarbeit, Fakultät Angewandte Computer- und Biowissenschaften
Hochschule Mittweida — University of Applied Sciences, November 2023

Referat

In dieser Arbeit wird der Einsatz des Wave Function Collapse Algorithmus untersucht. Dazu werden Anforderungen an das Leveldesign für das Videospiel Counter Strike: Global Offensive als Vorlage genutzt. Der Algorithmus wird in der Unity Engine implementiert und evaluiert. Es werden drei Versuchsreihen durchgeführt. Jede Versuchsreihe nutzt andere Einstellungen für die Levelgenerierung und analysiert welche Anforderungen erfüllt werden können. Die Ergebnisse werden verglichen und es werden Rückschlüsse auf die Anwendbarkeit des Algorithmus für die Erstellung von Multiplayer Level mit Ähnlichkeit zu Counter Strike: Global Offensive gezogen.

Name: Bätz, Johannes
Studiengang: Medieninformatik und Interaktives Entertainment
Seminargruppe: MI20w2-B
English Title: Wave Function Collapse as a Gamedesign Tool in Unity Engine - Investigating Possible Uses in Leveldesign

Inhaltsverzeichnis

1 Einführung und Motivation

Videospiele (Games) sind im Vergleich zu Radio, Fernsehen oder Büchern sehr junge Medien. Dabei scheint ihre Beliebtheit in den letzten 40 Jahren stetig anzusteigen. Die Spielerschaft wird immer größer und die Branche, welche einst eine Niche bediente, hat ältere Medien längst eingeholt. So ist der Umsatz der gesamten Branche größer als die Umsätze der Film und Musik Industrie zusammen (vgl. 1.1). Gamestudios sind nicht mehr ausschließlich kleine Firmen, welche mit zwei Personen ein Spiel entwickeln, sondern teilweise Konzerne mit über 10.000 Mitarbeitern auf der ganzen Welt. Techniken zum Entwickeln der Spiele haben sich verbreitet und werden jedes Jahr ausgeprägter. Dies führte dazu das Teils neue Berufe wie der des "Gamedesigners“ entstanden sind. Außerdem wurden technisch große Fortschritte erzielt, Textadventures sind eine Seltenheit und graphische 3D Echtzeit-Spiele dominieren den Markt.

Um die stetig wachsende Kundschaft zu bedienen, bemühen sich viele Unternehmen immer größere und komplexere Spiele zu erschaffen. Viele der großen Spieleproduktionen sind mehrere Jahre in Entwicklung. Neue Produkte sollen Kunden über einen langen Zeitraum beschäftigen und die Markenbindung verbessern. Die Spiele ordnen sich oftmals in das Genre der "Open World“ Games ein. Der Spieler bekommt dabei die Möglichkeit sich über eine große Fläche (oft mehrere Quadratkilometer) gefüllt mit Ingame Content (Aufgaben, Herausforderungen) zu bewegen. Eine Welt zu erstellen und mit ausreichend Inhalt auszustatten, damit ein Kunde bestenfalls mehr als 100 Stunden im Spiel verbringen kann erfordert viel Zeit, Geld und erfahrene Arbeitskräfte.

Abbildung 1.1: Vergleich des Weltweiten Umsatzes der Videospiel-Industrie und anderer Unterhaltungsindustrien.

1.1 Aufgabenbeschreibung

Diese Arbeit beschäftigt sich mit dem Wave Function Collapse Algorithmus (Wave Function Collapse). Im Zentrum steht die Frage, ob er als ein Tool der prozeduralen Levelgenerierung (prozedurale Levelgenerierung) im 3D Kontext eingesetzt werden kann. Das entstandene Tool wurde mit drei verschiedenen Einstellungen mit jeweils 31 Testläufen evaluiert. Für die Bewertung der Resultate dienen Anforderungen, die ein Level des Competetiven Multiplayer Shooters Counter Strike: Global Offensive (Counter Strike: Global Offensive) erfüllen muss. Die Anforderungen wurden aus Analysen zu CSGO abgeleitet. Der Algorithmus wird daraufhin in seiner Funktionalität für Levelgenerierung mit bestimmten Vorgaben bewertet.

1.2 Zielstellung und Aufbau der Arbeit

Ziel der Arbeit ist es den Einsatz des WFC Algorithmus als prozedurales Leveldesign Werkzeug zu untersuchen und ein Einsatz unter definierten Anforderungen möglich ist.

Im theoretischen Teil der Arbeit wird erklärt, welche Anforderungen ein Level des Spiels CSGO und ein Level des WFC Algorithmus erfüllen muss. Danach wird ein Überblick über die Methoden des prozeduralen Leveldesigns geschaffen. Anschließend wird der WFC Algorithmus erklärt. Hierbei wird auf den Ursprung und die Bedeutung des WFC Algorithmus eingegangen und wie dieser zu einem Werkzeug für Leveldesign wurde. Zum Ende des Kapitels wird ein Überblick über einige seiner Modifikationen dargestellt.

Das folgende Kapitel bezieht sich auf die Konzeption des Tools. Das hier entstandene Produkt unterscheidet sich in einigen Punkten stark von seiner ursprünglichen Implementierung als Bildsynthese Algorithmus. Daher wird auf Input und Parameter für den Algorithmus eingegangen. Das Tool ist entworfen um den Design Prozess zu unterstützen. Unterstützende Maßnahmen werden dabei beleuchtet. Des Weiteren wird die Umsetzung der automatischen Analyse der generierten Level diskutiert.

Im vierten Kapitel geht es um die Implementierung des Tools in der Unity Engine. Dabei wird konkret erklärt, wie bestimmte Konzepte umgesetzt wurden. Es wird ein Überblick über alle Bestandteile des Produkts gegeben und ihre Funktionsweise beschrieben. Die automatische Analyse der generierten Level wird erläutert.

Das fünfte Kapitel beschäftigt sich mit der Evaluation des Produkts. Die gesammelten Daten werden visualisiert und interpretiert. Es fällt auf, dass es nicht möglich ist diese klaren Anforderungen zu erfüllen. Das Ergebnis wird untersucht und nach möglichen Fehlerursachen und Möglichkeiten einer erfolgreicheren Durchführen analysiert.

Die Arbeit endet mit einer Zusammenfassung der Aufgabenstellung und ihrer Erfüllung. Es wird auf der Ergebnis eingegangen und ein möglicher Ausblick für weitere Untersuchungen geliefert.

1.3 Prozedurale Content Generierung als Unterstützendes Werkzeug

Prozedurale Contentgenerierung beschreibt die regelbasierte Erstellung von Videospielinhalten. Designer erstellen Regeln, die ein Tool benutzt um automatisch weitere Inhalte zu erschaffen. Das Ziel ist schnell große Mengen neuer Inhalte für Games zu erschaffen. Die Entwicklung von Techniken zur Prozeduralen Content Generierung beschäftigt die Games Industrie seit Jahren. Der umfangreiche Bedarf an Ressourcen für die Entwicklung von großen "Open World" Games führte zur Suche nach Lösungen den Prozess des Leveldesigns zu beschleunigen und die Arbeit zu erleichtern. Prozedurale Contentgenerierung beschäftigt sich mit vielfältigen Aufgaben in der Spieleentwicklung. Die Beiden wichtigsten Schwerpunkte liegen dabei auf dem Quest Design und dem Level- / Environment Design. Der Fokus des prozeduralen Quest Designs liegt dabei auf der Generierung von Aufgaben oder Herausforderungen für die Spieler. Prozedurales Environment- und Leveldesign setzt sich zum Ziel Umgebungen und Spielwelten zu erschaffen. Generierter Content wird je nach Spiel hinterher von Gamedesignern weiter überarbeitet und verbessert oder den Spielern direkt zur Verfügung gestellt.

1.4 Grundlegende Anforderungen an ein Level eines Videospiels

Videospiele gibt es in vielen verschiedenen Ausführungen und unterscheiden sich teils stark. Das Leveldesign ist abhängig vom Zugrunde liegenden Gamedesign. Dieses legt zum Beispiel Perspektiven, Dimension (2D/3D) und Tätigkeiten des Spielers fest. Das Leveldesign muss sich dabei an genau Anforderungen richten und den Spielern eine bestmögliche Erfahrung bieten. Die Anforderungen um eine bestimmte Erfahrung zu bieten sind schwer zu belegen. Daher wurde das Spiel CSGO gewählt. Es ist schon seit über 10 Jahren auf dem Markt und wird täglich von mehr als 100.000 Spielern gespielt. Außerdem gibt es umfangreiche Analysen, die viele Gemeinsamkeiten der Level belegen. Daraus lassen sich einfach technisch nachweisbare Eigenschaften ableiten, welche auch bei einem Level des WFC Algorithmus ermittelt oder widerlegt werden können.

1.5 Beschreibung des Spiels CSGO

Das Videospiel Counter Strike: Global Offensive (CSGO) ist ein Multiplayer Shooter des Entwicklers Valve. In dem Spiel gibt es verschiedene Spielmodi (Varianten das Spiel zu spielen). Der bekannteste und relevanteste Modus ist "Wettkampf". Darin spielen zwei Teams von je 5 Personen gegeneinander. Das eine Team besteht aus den "Terroristen", das andere aus der "Polizei". Die "Terroristen" müssen, je nach der Karte auf der gespielt wird, eine "Bombe platzieren" oder eine "entführte Geisel" beschützen. Die Polizei muss dabei die "Bombe entschärfen" oder die "Geisel befreien". Die verfolgten Ziele der Teams werden dabei "Objectives" genannt. Die Karten sind so ausgelegt, dass beide Teams eine möglichst gleiche Chance bekommen gegeneinander zu kämpfen.

2 Grundlagen

Das Kapitel dient der Vermittlung nötiger Grundkenntnisse. Es verfolgt grundlegend zwei Ziele: 1. die Darlegung und Begründung nötiger Anforderungen, welches ein Level für das Spiel CSGO erfüllen muss. Die Anforderungen gelten zudem gleichermaßen für die generierten Level des entwickelten Prototypen. 2. Es wird ein Überblick über Methoden der prozeduralen Levelgenerierung geboten. Der Überblick ist notwendig, um die Einordnung und Funktionsweise des WFC Algorithmus zu erklären, sowie seine speziellen Merkmale für den untersuchten Anwendungsfall zu verstehen. Zum Schluss wird noch eine Übersicht vermittelt, welche verschiedene Varianten des Modifizierten WFC Algorithmus beschreibt.

2.1 Game- und Leveldesign

Gamedesign ist ein Berufszweig, der durch die zunehmende Popularität der Videospiele entstanden ist. Mittlerweile gibt es viele Bücher und Medien, die sich mit diesem Thema beschäftigen und trotzdem ist eine exakte Definition schwierig (Schell zitieren). Das Kapitel versucht eine annähernde Definition für Gamedesign bereitzustellen, auf der die folgende Arbeit beruht und für das generelle Verständnis ausreichend ist. Darauf aufbauend wird Leveldesign erklärt und mit anderen Bereichen der Videospielentwicklung in einen Kontext gesetzt.

2.1.1 Definition Gamedesign

Gamedesign ist ein Bestandteil der Videospielentwicklung. Ein Gamedesigner plant und konzipiert Videospiele. Konkret geht es dabei um die Erstellung einer Spielwelt, Mechaniken und Regeln. Mechaniken bilden die grundlegenden Handlungsmöglichkeiten eines Spielers in der Spielwelt ab. Regeln sind Bestandteile, die definieren, wie sich die Spielwelt verhält und wie der Spieler mit ihr interagieren kann. Die Spielwelt

ist ein abstrakter Bereich in dem das Spiel mit seinen Regeln und Mechaniken besteht und eine möglichst glaubhafte Erfahrung für einen Spieler bieten soll. Die Arbeit erfolgt meist in theoretischer Umsetzung und soll anderen Entwicklern als Grundlage für die Umsetzung eines Spiels dienen.

2.1.2 Definition Leveldesign

Leveldesign ist ein Bestandteil des Gamedesigns. Ein Leveldesigner beschäftigt sich mit der konkreten Umsetzung der abstrakten Ausarbeitung des Gamedesigners. Die Ausprägung des Gamedesigns erfolgt in einem oder mehreren Leveln (auch genannt Stage oder Map). Ein Level ist ein abstrakter Raum, der die Spielwelt (oder einen Teil davon) und (mindestens) eine Mechanik beinhaltet. Alles was in einem Level geschieht, folgt dabei den Spielregeln. Das Ziel eines Levels ist es einen Spieler so immersiv wie möglich die Erfahrung des Spiels erleben zu lassen.

2.1.3 Bestandteile eines Levels

Jedes Level besitzt bestimmte Merkmale, welche hier erklärt werden.

- Eingang und Ausgang
  Jedes Level muss betreten und verlassen werden können. Es ist nicht unbedingt notwendig, dass diese an ein nächstes Level anknüpfen.
- Start und Ende
  Jedes Level besitzt mindestens eine Ausgangssituation, bevor ein Spieler damit interagiert hat. Ein Level braucht mindestens eine Bedingung um als beendet zu gelten (es muss ein Punkt erreicht werden, bei dem alle Aufgaben erledigt sind).
- Aufgabe
  Jedes Level bietet mindestens eine Aufgabe die der Spieler erfüllen soll.
- Physische Grenze
  Jedes Level ist physisch begrenzt (die Grenze kann auch technischer Herkunft sein) und damit endlich.

2.1.4 Leveldesign im Kontext der anderen Entwicklungsbereiche

Leveldesign ist in seiner Rolle neben Gamedesign sehr eng mit den Bereichen Grafikdesign (Artdesign) und Programmierung verbunden. Es besitzt die Aufgabe Gamedesign Konzepte verständlich an andere Bereiche zu vermitteln. Außerdem soll jedes Level, neben seinen Bestandteilen im Gamedesign, sowohl optisch als auch technisch optimiert sein und ein zufriedenstellendes Ergebnis liefern.

Damit ein Level im Grafikdesign überzeugen kann, muss das Leveldesign den gewählten Stil, Architektur und produzierte Assets bestmöglich integrieren. Technische Überschneidung der Bereiche können im Erkennen und Vermeiden von Performance Problemen liegen, sowie dem Ausarbeiten von Interaktionsmöglichkeiten des Spielers mit dem Level.

2.2 Anforderungen an das Leveldesign für den Wave Function Collapse Algorithmus

Das Kapitel legt die Anforderungen, welche an ein Level des Spiels CSGO gerichtet sind, dar. Daraufhin wird ein Level von CSGO analysiert. Außerdem wird erklärt, warum CSGO als Referenz für die Anforderungen dient. Daraus ergibt sich die Grundlage für die weitere Arbeit. Konzeption und Implementierung werden die Anforderungen nutzen und die Evaluation wird das entstandene Level auf die Anforderungen analysieren und bewerten.

2.2.1 Anforderungen an ein CSGO Level

Anforderungen an ein Level sind je nach Spiel sehr unterschiedlich. Die hier betrachteten Anforderungen gelten dem Spiel CSGO und beziehen sich explizit auf Level (in CSGO Maps genannt), welche im Spielmodus "Wettkampf" und bei offiziellen Turnieren gespielt werden. Dies sind Maps, welche intensiv geplaytestet wurden und in fast allen Fällen über Jahre schon vor ihrer Verfügbarkeit im Spielmodus in anderen Modi gespielt wurden. Die Maps wurden dabei teilweise von Mitgliedern der

Community, teilweise von Gamedesignern des Entwicklers Valve erstellt. Der Grundaufbau der Maps ist sich in einigen Punkten sehr ähnlich, wodurch sich allgemeine Design Anforderungen und Grundlagen ableiten lassen. [Men14, Gal13]

Die grundlegenden Anforderungen an eine solche Map lassen sich in vier Kategorien einteilen:

1. Ein Level benötigt zwei Spawn Punkte (einen für jedes Team), welche im besten Fall im Level gegenüber liegen.
2. Es gibt auf der Map zwei Objectives.
3. Jedes Objective ist über ein bis zwei Hauptwege mit den Spawn Punkten verbunden.
4. Es gibt identifizierbare Choke Points zwischen den Objectives und den Spawn Punkten.

2.2.2 Analyse eines CSGO Levels

Als Beispiel für ein CSGO Level wird ”Dust 2“ verwendet. Die Map gibt es seit März 2001 und ist in jedem Counter Strike Spiel vor CSGO vertreten. [csg] Auf der Karte 2.1 sind in grüner Farbe die Spawn Punkte (von oben nach unten) für die Teams der ”Polizei“ und der ”Terroristen“ markiert. Die roten Bereiche stellen die beiden Objectives dar. In diesen Zonen können die ”Terroristen“ eine ”Bombe platzieren“, welche die ”Polizisten” entschärfen können. Die blauen und gelben Flächen stellen die Wege und Verbindungen dar, auf denen beide Teams zu den Objectives gelangen können. Dabei ist zu erkennen, dass beide Teams jeweils auf der linken und rechten Seite mindestens einen Hauptweg zu den Objectives besitzen (durch kleinere Verbindungen sind es mehr). Die Abbildung 2.2 zeigt den Choke Point auf dem Weg der Terroristen von ihrem Spawn Punkt zum ”Bombenpunkt B“ (Objective). Der Gang ist dabei ein langer Tunnel ohne Deckung. Er mündet in ein offeneres Gebiet und bildet potentiell eine Stelle intensiver Gefechte zwischen den Teams.

Abbildung 2.1: Die CSGO Map ”Dust 2“ aus der Vogelperspektive. In grün sind Spawn Punkte (oben: ”Poliszisten“, unten: ”Terroristen“) zu sehen. Rote Objectives liegen links und rechts der Spawn Punkte. Blaue und gelbe Flächen sind begehbar und verbinden alle wichtigen Orte miteinander.

Abbildung 2.2: Die Abbildung eines Choke Points auf dem Weg der ”Terroristen“ zum Objective ”Bombenpunkt B“. Ein schmaler langer Gang führt in ein offenes Gebiet. Der Durchgang bietet keine Deckung und macht eine Durchquerung für ein Team gefährlich.

2.2.3 Wahl für CSGO

Das Videospiel CSGO wurde als Beispiel für die Anforderungen an den WFC Algorithmus gewählt. Dafür gibt es vorranging zwei Gründe. Der Algorithmus wurde bisher vorrangig in kleinen Videospielproduktionen, wie ”Bad North“, ”Dead Static Drive“, ”Caves of Quad“, ”Townscaper“, ”Proc Skater 2016“, ”Swapland“ oder ”Wave Function Collapse“ (benannt nach eben jenem Algorithmus), benutzt. [Gum16]. Die Spiele sind aufgrund ihrer Bekanntheit, Umfang, Ausrichtung auf Einzelspieler-Erlebnisse und Budget nicht repräsentativ für einen großen Teil des Videospielmarktes. CSGO ist ein Multiplayer Spiel. Es ist eines der wichtigesten Spiele in der E-Sport Szene (.1.4) und zählt zu den meistgespielten Spielen auf Steam (.1.2, .1.3, .1.1). CSGO ist ein Spiel welches eine breitere Zielgruppe und Community besitzt, als Spiele, die bisher den WFC Algorithmus verwendet haben. Die Evaluation nimmt sich daher ein Spiel als Referenz, welches sich stark von den bisherigen unterscheidet.

2.3 Taxonomie der prozeduralen Levelgenerierung

Prozedurale Levelgenerierung (PLG) hat sich in den letzten Jahren stetig weiterentwickelt und verändert. Das Feld kann daher in verschiedene Bereiche eingeordnet werden. Hier wird eine Systematik bereitgestellt, die wichtigsten Kriterien unterscheidbar darzustellen.

2.3.1 Online und Offline

Prozedurale Levelgenerierung kann während zwei unterschiedlichen Zeitpunkten geschehen. Online bezieht sich darauf, wenn das Produkt bereits erschienen ist. Das Level wird hierbei zur Laufzeit des Spiels für den Spieler erstellt. Offline ist das Gegenteil, dabei wird das Level während der Produktion erstellt. Entwickler können dabei noch aktiv das Level evaluieren und anpassen. [JT16, TYSB11]

2.3.2 Optional und Notwendig

PLG kann unterschieden werden anhand der Art des generierten Contents. Er kann optional oder notwendig für das beenden des Spiels sein. [JT16, TYSB11] Notwendiger Content ist zum Beispiel im Spiel "The Binding of Isaac" integriert. Die Level sind bei jedem Spieldurchlauf anders und der Spieler muss lernen sich den neuen Gegebenheiten anzupassen. Optionale PLGs sind im Spiel "Dead Cells" implementiert, in Form des zusätzlichen Spielmodus "Daily Run". Dabei bewegt sich der Spieler bis er besiegt wird durch ein prozedural generiertes Level.

2.3.3 Möglichkeiten des Einflusses auf PLG

Methoden und Ablauf der PLG sind sehr unterschiedlich. Die Möglichkeiten und Parameter, welche Nutzbar sind um den Prozess der PLG zu beeinflussen, sind je nach Wahl des Algorithmus unterschiedlich. Das Spiel Minecraft nutzt einen "Seed" mit dem es möglich ist das gleiche Level erneut zu generieren. [JT16, TYSB11] Der WFC hingegen kann nicht genutzt werden um zielgerichtet das gleiche Level erneut zu erstellen.

2.3.4 Generische und Adaptive PLG

PLGs können entweder generisch oder adaptiv generiert werden. Generische Level haben die Aufgabe einem möglichst großen Teil der Spielerschaft eine gute Erfahrung zu bieten. Dabei passen sie sich nicht der aktuellen Situation des Spielers an. Adaptive Level hingegen passen sich dem individuellen Spieler an und können so in ihrer Schwierigkeit variieren. [JT16, TYSB11]

2.3.5 Stochastische und Deterministische PLG

Stochastische PLG nutzt Zufallsfaktoren, um Bestandteile eines Levels zu erstellen. Hierbei können im Regelfall gleiche Level nicht gezielt erneut erstellt werden. Deterministische Verfahren greifen auf Algorithmen zurück, welche bei gleichen Parametern gleiche Level erstellen. [JT16, TYSB11]

2.3.6 Konstruktive und testbasierte PLG

Konstruktive PLG besteht daraus den gewählten Algorithmus ein mal durchlaufen zu lassen. Das Resultat sollte die Regeln auf welchen der Algorithmus beruht erfüllen, wird aber nicht weiter betrachtet. Testbasierte Algorithmen evaluieren selbstständig ihr Ergebnis. Sollte das Resultat ganz oder teilweise nicht den Regeln entsprechen wird der Algorithmus erneut ausgeführt. [JT16, TYSB11]

2.4 Methoden der prozeduralen Levelgenerierung

PLG ist ein Oberbegriff für eine Vielzahl von Verfahren. Grobe Unterschiede in der Systematik wurden bereits beleuchtet. Dieses Kapitel soll einen Überblick über verschiedene häufig genutzte Methoden bieten, welche für PLG genutzt werden.

2.4.1 Suchbasierte PLG

Suchbasierte PLGs sind immer Test basierte PLGs. Sie bestehen grundlegend aus zwei Funktionen. Einem Algorithmus, welcher durch Parameter ein Level generieren kann und eine Bewertungsfunktion. Die Bewertungsfunktion evaluiert die entstandenen Level. Daraufhin werden die Parameter des Algorithmus angepasst und der Prozess so lange wiederholt bis ein erwünschtes Ergebnis entstanden ist. Die meisten dieser Verfahren fallen in die Kategorie der evolutionären Algorithmen. [JT16, TYSB11]

2.4.2 Konstruktive PLG

Nach Togelius et al. [JT16] ist eine konstruktive PLG mit einem abgeschlossenem Raum verknüpft. Als Beispiel wird ein "Dungeon" angebracht, welcher verschiedene Herausforderungen und Gegner bietet, dabei aber immer ein gleichbleibendes Pacing (Flow des Levels) beibehalten sollte. Die Verfahren hierbei sind drei geteilt. Der erste Teil beschäftigt sich mit der abstrakten Erstellung eines Layouts für das Gebiet. Daraufhin wird das abstrakte Model in ein konkreteres repräsentatives Model umgewandelt. Zum Schluss wird daraus wirkliche Geometrie erzeugt und ein Level erstellt. Diese Phasen können je nach Design Aufgaben, NPCs oder Herausforderungen einbauen. Die Verfahren generieren ein fertiges Level als Output, welches nicht weiter evaluiert wird.

2.4.3 Stochastische PLG

Stochastische Methoden sind oft eingesetzt um Terrain in Videospielen zu generieren. Bekannte Techniken sind dabei Noise Funktionen oder Fraktale. Mathematische Zufallsbasierte Funktionen werden dabei genutzt, um die Verformung von Terrain oder Meshes zu bestimmen und anzupassen. [TYSB11, JT16]

2.4.4 PLG über Formale Grammatik

Eine Möglichkeit PLG durchzuführen besteht durch die Nutzung formaler Grammatik. Die Grammatik kann genutzt werden um als Zeichenanweisung interpretiert

zu werden [RS91]. Auch wenn diese Nutzung oftmals zur prozeduralen Generierung von Pflanzen genutzt wird, kann sie für die PLG verwendet werden. [JT16]

2.5 Wave Function Collapse Algorithmus

Der WFC Algorithmus wurde ursprünglich von Maxim Gumin entwickelt und über ein Github Repository veröffentlicht [Gum16]. Der Algorithmus wurde ursprünglich zur Generierung von Bildern genutzt. Dazu wurden Bilder als Eingabe Daten genutzt. Der Algorithmus gab daraufhin Bilder mit ähnlichen Mustern im Vergleich zu den Originalen aus. Seitdem hat sich der Open Source Algorithmus weiterentwickelt und wurde vielfach modifiziert. Dieses Kapitel soll den Algorithmus in seiner Grundfunktion erklären und einordnen.

2.5.1 Terminologie

Um die Vorgänge des WFC Algorithmus möglichst einfach zu beschreiben wird hier Bezug genommen auf den Begriff ”Pattern“. Ein Pattern ist dabei ein zusammenhängender Teil einer Input Bitmap [Gum16, KS22]. In einigen anderen Quellen wird anstatt diesen Begriffs auch ”Tile“ genutzt.

2.5.2 Einordnung

Der WFC Algorithmus fällt in den Bereich der ”greedy constraint satisfaction“ Problemlösungsalgorithmen für Bildsynthese. Sein Vorgehen entspricht dabei der Lösung eines Problem über ein Set an bekannten Variablen und Wahrscheinlichkeitsberechnung. Er kann zu den stochastischen PLG Verfahren gezählt werden. [KS22]

2.5.3 Funktionsweise

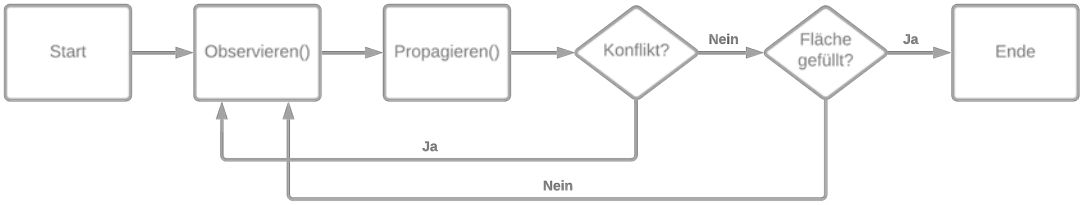

Abbildung 2.3: Ablauf des WFC Algorithmus [Gum16, KLL+19]

Der Ablauf des WFC Algorithmus wird in 2.3 dargestellt. Der Algorithmus braucht als Input eine Bitmap. Der Output des Algorithmus garantiert, dass ausschließlich Pattterns des Inputs zur Generierung benutzt wurden. Der Ablauf wird hier noch einmal genauer erklärt.

1. **Start**
   Während des Startes oder auch der Initialisierung wird als Input eine Bitmap gelesen, die Größe eines NxN Patterns festgelegt und das Ausgabe Array (auch ”wave“ genannt) erstellt. Jedes Element des Ausgabe Arrays wird mit allen verfügbaren Patterns aus dem Eingabe Array gefüllt. Ein Pattern entspricht dabei einem NxN Quadrat aus dem Eingabebild. Alle Patterns eines Elements des Ausgabe Arrays sind die möglichen Patterns, welche das Element annehmen kann. [KS22, LWB18]

2. **Observieren**
   Das Ausgabe Array wird observiert. Dabei wird nach dem Array Element gesucht, welches die kleinste Entropie größer Null besitzt (die wenigsten noch verfügbaren Patterns zur Auswahl). Sobald dieses gefunden ist, wird zufällig eines der verfügbaren Elemente ausgewählt und als einziges verfügbares Pattern markiert (dies wird auch ”collapse“ genannt). Dieser Prozess ist nur pseudo zufällig und basiert auf keinem echten Zufall. [KS22, LWB18]

3. **Propagieren**
   In diesem Schritt wird die Änderung des zu letzt geänderten Feldes an alle anderen Felder übertragen. Nicht alle Pattern sind miteinander kompatibel (die Pixel an ihren Rändern stimmen nicht überein). Daher kann es dazu kommen,

dass benachbarte Felder nicht mehr alle ihrer verfügbaren Patterns platzieren können. Diese werden aus dem Ausgabe Array ausgeschlossen. [KS22, LWB18]

4. **Konflikt**
   Nach dem propagieren wird überprüft, ob es dazu kam, dass ein Feld keine Verfügbaren Patterns mehr besitzt. Sollte dieser Fall eintreten, bricht der Algorithmus entweder ab oder führt den "Observieren" Schritt erneut durch. Im zweiten Fall entsteht im Ausgabe Array ein leeres Feld. Die Entscheidung welche der beiden Varianten genutzt wird, hängt von der Implementation ab. [KS22, LWB18]

5. **Fläche gefüllt**
   Sollte es keinen Konflikt gegeben haben, wird geprüft ob die Fläche gefüllt wurde oder nicht. Ist die Fläche gefüllt, also jedes Element des Ausgabe Arrays mit einem Pattern ausgestattet, ist der Algorithmus zu Ende. Sollte die Fläche nicht gefüllt sein, wird der Schritt "Observieren" erneut ausgeführt. [KS22, LWB18]

6. **Ende**
   Das Ausgabe Array wird in eine Bitmap umgewandelt und ausgegeben. [KS22, LWB18]

2.5.4 Eigenschaften des WFC Algorithmus

Der WFC Algorithmus bringt eine Reihe von Eigenschaften mit sich. Diese werden hier näher erläutert und sind für den weiteren Verlauf der Arbeit wichtig.

1. Gitter
   Aufgrund der Herkunft des Algorithmus aus der Bildsynthese, verlangt dieser in einfachen Implementationen immer ein Gitter. Dadurch ist der Algorithmus in seiner Tätigkeit etwas eingeschränkt, da jedes Feld des Gitters die gleiche Größe besitzt. Einzelne Patterns sind daher immer gleich groß. [KS22]

2. Eingreifbarkeit
   Ist der Algorithmus gestartet, so kann man auf ihn keinen weiteren Einfluss nehmen, mit Ausnahme der Eingabe der Input Parameter. Eine gezielte Erstellung des gleichen Bildes wird so nahezu unmöglich gemacht. [KS22, LWB18]

3. Analyse und Pattern Recognition
   Der Algorithmus arbeitet mit Patterns. Diese werden nicht durch einen Pattern Recognition Algorithmus gefunden. In dem man eine Pattern Größe von NxN angiebt, erlangt man eine Fläche. Diese Fläche wird nacheinander aus der Input Bitmap eingelesen und als Pattern verwertet. Ein hochauflösendes Foto wäre ein schlechter Input, hingegen eine niedrig aufgelöste Bitmap ein besserer. [KS22, LWB18]

4. Ähnlichkeit von Input und Output
   Ein Ausgegebenes Bild stellt immer eine Ähnlichkeit zu dem eingegebenen Bild dar. Es besteht nur aus Pattern, welche im ursprünglichen Bild zu finden sind. Außerdem sollte die Verteilung der Pattern in der Ausgabe etwa der Verteilung in der Eingabe entsprechen. [KS22, LWB18]

2.6 Einsatzmöglichkeiten des Wave Function Collapse Algorithmus

In diesem Kapitel wird der Einsatz des Algorithmus in Bezug auf den Einsatz von Leveldesign Möglichkeiten gelegt. Ziel ist es den Weg als Bildsynthese Algorithmus zum Werkzeug in andere Bereiche abzubilden.

2.6.1 Mehrdimensionaler Einsatz in Videospielen

Obwohl der WFC Algorithmus ursprünglich zur Bildsynthese genutzt wurde, ist er durch seine Funktionsweise auch im Kontext des Leveldesigns anwendbar. Die Patterns bestehen aus einer Anzahl von einem bis mehreren Sprites im 2D Bereich oder einer Ansammlung von Modellen im 3D Bereich. Die Kanten der Patterns sind dementsprechend die äußeren Pixel der Sprites bzw. Vertices entlang der Seitenfläche eines Modells (gerade im 3D Bereich kann es je nach Implementierung auch andere Ausprägungen geben). Der Algorithmus ist für theoretisch jede Dimension anwendbar, allerdings steigt die Laufzeit, je größer das Gitter und je anspruchsvoller die Vergleiche ausfallen. [Gum16, Mor21]

2.6.2 Offline und Online nutzen

Der WFC Algorithmus kann dabei online und offline genutzt werden. Als Beispiel einer Online Nutzung können die Videospiele ”Bad North“ und ”Townscaper“ von Oskar Stålberg genannt werden [Stå18]. Zur Laufzeit des Spiels wird die Spielwelt generiert. Der Spieler bekommt dabei das Gefühl eine individuelle Spielerfahrung zu bekommen während er sich trotzdem in einer ähnlichen Spielwelt befindet.

Eine offline Nutzung entspricht dem Einsatz während des Entwicklungsprozesses. Dabei kann der Algorithmus vielfältig genutzt werden. Er kann viele Varianten eines Levels entwerfen oder Level teilweise weiterentwickeln. Eine andere Möglichkeit ist der Einsatz um die Umgebung für ”Points of Interest“ (spezielle oder besondere Bereiche in einem Level) herum zu erschaffen [KLL+19]. ”Points of Interest“ werden von Designern entworfen und gestaltet. Damit die Bestandteile befriedigend verbunden werden, wird der WFC Algorithmus genutzt, welcher freie Flächen in der Welt füllt.

2.6.3 Anwendung in der Industrie

Der WFC Algorithmus in Kombination mit neuronalen Netzwerken wurde angewendet um Infrastruktur in China (Whenzhou) zu planen. In dieser Kombination konnte ein Prototyping der Planung vereinfacht werden. Diese Vorplanung muss allerdings von Designern weiterhin evaluiert und verbessert werden und kann nicht als einzige zuverlässige Methode angewendet werden. [BL20]

2.7 Modifikationen des Wave Function Collapse Algorithmus

Der WFC Algorithmus wurde über die Jahre in vielen verschiedenen Varianten angepasst und weiterentwickelt. Dieses Kapitel soll anhand der Beschriebenen Eigenschaften in 2.5.4 zwei Beispiele zeigen, die diese Eigenschaften manipuliert und verändert haben.

2.7.1 Verändern des Gitters

Der WFC Algorithmus nutzt ein Gitter, in dem alle Module platziert werden. Das ist eine Einschränkung, weil es Pattern die Möglichkeit nimmt abstrakte Formen annehmen zu können. Ein Graph kann dafür eine Lösung sein. Er wird anstatt des Gitters als Grundlage der Platzierung von Pattern genutzt. Dazu definiert der Graph die Zusammenhänge aller Pattern und wird für die Platzierung genutzt. Im Vergleich zum ursprünglichen WFC Algorithmus ist es dabei für ein Pattern theoretisch möglich unendlich viele Nachbarn zu haben. Die Anzahl an möglichen Nachbarn kann dabei für jedes platzierte Pattern unterschiedlich sein. Angewandt wurde diese Umsetzung schon bei Navmeshes. [KLL+19]

2.7.2 Verbesserungen des Analysesystems

Das Analysesystem kann vielfältig verbessert werden. Eine der populärsten Varianten besteht darin den Input zu verändern. Dabei wird oftmals zu Programmstart ein weiterer Arbeitsschritt eingefügt. Dieser analysiert Patterns oder ein vorgegebenes Level und leitet sich Regeln für das Zusammenführen der einzelnen Module entsprechend ab. [Nun20] Eine weitere Möglichkeit der Modifikation ist das gezielte Ändern aller verfügbaren Patterns in der Zelle des auszufüllenden Gitters. Dadurch kann ein Designer mehr Einfluss auf die Auswahl des Moduls in einem Bereich nehmen. [Nun20]

3 Konzeption des Systems

Nachdem im vorherigen Kapitel der Algorithmus erklärt wurde und dabei auf seine Anwendung Bezug genommen wurde, wird in diesem Kapitel der Einsatz des WFC Algorithmus erläutert. Der größte Unterschied zur ursprünglichen Implementation besteht darin, dass dieser nicht mehr zweidimensionale Bilder erstellt, sondern benutzt wird um dreidimensionale virtuelle Welten zu erschaffen. Der Fokus bildet dabei die Funktionsweise des Algorithmus in dreidimensionaler Arbeitsweise und welche Eigenschaften schon vorher beachtet werden müssen. Außerdem wird die Analyse der Tiles näher erklärt. Weiterhin werden Anforderungen an ein solches Tool für die Unity Engine dargelegt und mögliche Probleme beleuchtet. Zum Schluss wird erläutert, wie automatische Funktionen die Anforderungen an eben jenes Tool messen und welchen Herausforderungen sie unterliegen.

Der ursprüngliche WFC Algorithmus wurde zur Generierung von zweidimensionalen Bitmaps erstellt. Da das Ziel dieser Arbeit ein Einsatz des Algorithmus als Tool zum Designen von dreidimensionalen Leveln ist, wird sich dieses Kapitel mit den daraus resultierenden Veränderungen beschäftigen.

3.1 Analyse und Input der Module

Module (erklärt in 3.1.1) können im 3D Kontext nicht mehr durch eine Bitmap eingegeben werden. Die möglichen Beziehungen sind durch eine zweidimensionale Darstellung nicht ausreichend darstellbar und ablesbar. In diesem Kapitel werden zwei Methoden näher erklärt, welche benutzt werden können, um Module richtig einzuordnen. Die erste Variante beschäftigt sich mit dem Einlesen einer Datei, welche die Beziehungen beinhaltet. Die zweite Methode beschäftigt sich mit der Möglichkeit die Analyse der Module automatisch durchzuführen.

3.1.1 Terminologie

Im Vergleich zur ursprünglichen Terminologie wird ”Tile“ oder ”Pattern“ nun Modul genannt. Ein Modul ist dabei ein abstrakter Begriff, welcher einen kleinen finiten Bestandteil eines Levels beschreibt und dem WFC Algorithmus als Grundbaustein dient. Dieser Begriff wurde gewählt, da er im 3D Kontext für weniger Verwirrung sorgt. Des weiteren wird der Begriff ”Socket“ verwendet. Dieser Begriff beschreibt den Bestandteil des Moduls, welcher zur Bestimmung der Kompatibilität anliegender Module benötigt wird.

3.1.2 Verfahren der Analyse

Der Input des WFC Algorithmus wird für die Anwendung im 3D Bereich angepasst. In diesem Fall arbeitet der Algorithmus mit 3D Modellen und nicht mit Bitmaps. Daher reicht die Farbe der Pixel am Rand eines Tiles zur Bestimmung nicht aus. Es werden zwei Methoden beleuchtet, die eine Darstellung der Beziehung zwischen den Modulen ermöglichen:

1. Eingabe einer Datei, welche die Zugehörigkeit der Module bereits vorschreibt. Der Algorithmus benutzt dabei ausschließlich die eingegebenen Regeln. In der Datei werden für jeden Socket eines Moduls alle Sockets angelegt, die platziert werden können. Der Designer soll die Verbindung zwischen den Modulen bestimmen und in eine Richtung lenken können.

2. Automatische Analyse der Modelle, um geometrisch passende Module zu ermitteln. Dafür wird überprüft welche Modelle jeweils in einem Socket (am Rand) eines Moduls liegen. Die Vertices werden in dem Socket gespeichert und mit den Positionen der anderen Vertices anderer Sockets verglichen. Sollten passende Sockets gefunden werden, werden die Sockets als passend markiert und können als Regel für den WFC Algorithmus benutzt werden. Der Aufwand dieses Vergleiches ist aufwändiger in der Laufzeit. Da der Vergleich auf geometrischen Grundlagen beruht ist eine äußerst exakte Arbeit eines Designers erforderlich, da die Vergleiche bei kleinen Fehlern fehlschlagen. Die Arbeit mit einer automatischen Analyse kann viel Zeit in Anspruch nehmen, da vorher erstellte Module geplant und getestet werden müssen, damit eine erfolgreiche Analyse auch stattfinden kann.

Ein weiteres Problem in der Arbeit mit der automatischen Analyse besteht in der Unity Engine. Jedes Modul liegt dort als Prefab vor. Nicht alle Prefab Bestandteile bestehen aus Modellen. Ein zu implementierender Algorithmus muss mit allen weiteren Bestandteilen umgehen können. In der weiteren Betrachtung wird sich dafür entschieden alle weiteren Gameobjects bei der Analyse zu ignorieren.

3.2 Platzierung der Module

In diesem Kapitel werden Anforderungen dargelegt, welche für den Algorithmus notwendig sind, damit ein prozedurales Level erstellt werden kann. Es wird auf die Einschränkungen für die Erstellung von Modulen eingegangen. Außerdem wird das Gitter näher erklärt, auf dem der WFC Algorithmus operiert. Zuletzt wird auf das Verhalten während der Generierung eingegangen.

3.2.1 Anforderungen an Module

Jedes Modul wird in seinem Ausmaß in der Unity Standard Einheit bemessen. Ein Modul muss dabei immer das Volumen eines Würfels ausfüllen. Die Ausdehnung des Würfels muss dabei für alle Module, die kompatibel sein sollen, gleich sein. In der Ausführung muss dabei nur auf die Seiten "Links", "Rechts", "Vorne" und "Hinten" geachtet werden. Die Sockets für "Oben" und "Unten" sind bei der Erstellung weniger wichtig, da der Algorithmus vorerst nur planare Level generieren wird.

3.2.2 Anforderungen durch die Nutzung eines Gitters

Der WFC Algorithmus arbeitet im Vergleich zu seiner klassischen Implementation nicht mehr mit Pixeln, sonder im 3D Kontext. Um die Änderung in der höheren Dimension richtig abzubilden, wird ein Gitter eingesetzt, da es der ursprünglichen Struktur sehr ähnlich ist. Jedes Gitter Feld ist zu beginn leer und wird während der Laufzeit mit Modulen gefüllt. Das Gitter ist liefert harte Kriterien, jedes Modul muss die gleiche Ausdehung besitzen. Einstellbar ist, neben der Würfelausdehnung für Module, nur die Ausdehnung des Gitters in x und y Richtung. Das Gitter ist abseits davon nicht anpassbar.

3.2.3 Verhalten während der Laufzeit

Die Sockets der Module stellen den Übergang zu anderen Modulen dar und liegen genau auf der Grenze des Gitters. Sobald Module platziert wurden und ein Modul benachbart dazu platziert werden soll, muss überprüft werden, ob die anliegenden Sockets mit den Sockets eines zu platzierenden Moduls kompatibel sind. Ein zu platzierendes Modul muss dazu drei mal um die y-Achse (der Unity Engine) gedreht werden, damit alle Kombinationen überprüft werden können. Sobald eine Position gefunden wurde, in der alle Sockets der platzierten Module und des neuen Moduls kompatibel sind, kann es platziert werden.

Die Anforderungen an den Algorithmus beziehen keinen Höhenunterschied ein. Die Level von CSGO bieten teilweise Vertikalität, allerdings selten mehr als den Unterschied von ein bis zwei Spielergrößen. Daher wird der Algorithmus nur planare Level erstellen, also nur ein Modul in der Höhe beinhalten. Die kleinen Höhenunterschiede sind damit umsetzbar.

3.3 UI und UX des Tools für die Engine

Das Tool soll es ermöglichen einfach prozedural generierte Level zu erstellen. Die Unity Engine ermöglicht die Implementation des Algorithmus in zwei Verschiedenen Situationen: Playmode und Editmode. Der Editmode ist der normale Arbeitsmodus in der Engine. Der Playmode ist ein eingebauter Emulator, der das entwickelte Produkt möglichst realitätsnah darstellen soll, ohne dass es vollständig export werden muss. Die Herausforderung des Playmodes ist, dass Änderungen am Produkt nach Verlassen nicht gespeichert werden. Der WFC Algorithmus wird daher während des Edit Modes ausgeführt, damit Designer das entstandene Level leichter in Form einer Szene speichern können oder Änderungen vornehmen können. Benötigte Module werden in Form von Prefabs erstellt und abgespeichert. Der bekannte Workflow der Engine soll durch das Tool so wenig wie möglich behindert werden und User sollen sich so wenig wie nötig in neue Arbeitsschritte einarbeiten. Daher sollen sich die genannten Arbeitsschritte so nahtlos wie möglich in die Engine einfügen.

3.4 Ermitteln der Anforderungserfüllung

Um die Erfüllung der Anforderungen zu ermitteln, wird eine automatisierte Kontrolle angestrebt. Der Vorteil wäre die Bestimmung der Ergebnisse, unabhängig von einem menschlichen Fehler. Eventuell auftretende Fehler sind für alle durchgeführten Versuche gleich und die Vergleichbarkeit der Resultate bleibt gewährleistet.

- Jedes Team braucht mindestens einen Spawn Punkt. Dies wird ermittelt, in dem die Anzahl platzierter Spawn Punkte ermittelt wird. Außerdem werden Heatmaps zur Visualisierung Erstellt, anhand derer Abstände zwischen den Spawnpunkten identifiziert werden sollen.
- Es wird überprüft, ob es auf der Karte zwei Objectives gibt.
- Jedes Objective muss ein bis zwei direkte Wege besitzen, welche sie mit jeweils dem Spawnpunkt eines Teams verbindet.
- Es gibt einen Choke Point auf dem Weg von einem Spawn Punkt zu einem Objective.

Für den Test werden drei neue Module erstellt und eingefügt. Spawn Punkte, Objectives und Choke Points werden dadurch leichter zählbar und ihre Position leichter analysierbar.

4 Implementierung

Das Kapitel beschäftigt sich mit der konkreten Ausarbeitung eines Tools für die Unity Engine. Dieses Tool erstellt dreidimensionale Level unter Nutzung des WFC Algorithmus. Das Diagramm 4.1 ist eine schematische Abbildung des Systemdesigns. Des Weiteren ist es in der Lage benötigte Module und Sockets automatisch zu analysieren oder sie durch die Nutzung einer JSON Datei einzulesen. Die Generierung der Level wird im Vergleich zur ursprünglichen Implementation beschrieben, sowie das Aussehen der Module und Sockets näher erklärt. Diese Teile sind in Orange erkennbar. Die Benutzung des Tools wurde dabei so einfach wie möglich gestaltet und eine Bearbeitung während des Edit Modes von Unity ermöglicht. Die UI ist blau markiert und wird in 1 ausführlich präsentiert. In Grün sind weitere Teile beschrieben. Diese wurden nicht für die Evaluation genutzt, sind aber implementiert.

Der Zweite Teil des Kapitels beschäftigt sich mit der Implementation der automatischen Analyse der Erfüllung der Anforderungen, welche an ein Level gestellt wurden.

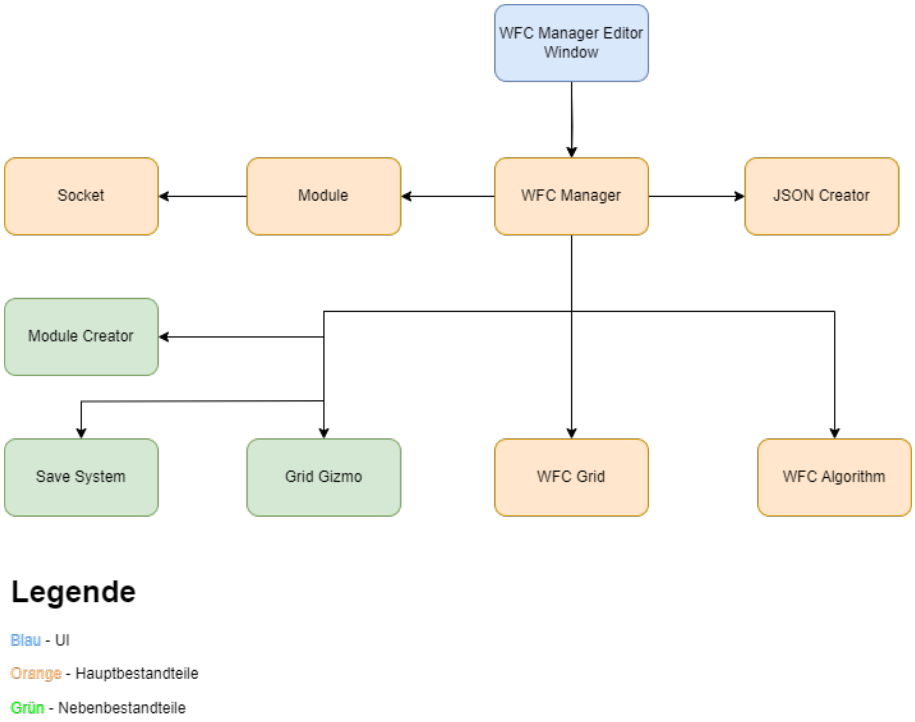

Abbildung 4.1: Schematische Darstellung des Aufbaus in Unity. In Blau ist der Bestandteil für das UI markiert. Orange sind Hauptbestandteile, welche für die Evaluation und die Levelgenerierung wichtig sind. Grün sind Teile des Tools, welche implementiert wurden, aber eine untergeordnete Rolle spielen und für die Evaluation nicht wichtig sind.

4.1 Implementierung der Analysefunktionen

In 3.1 wurde zwei Methoden, den Input der Module und Sockets zu gestalten, beschrieben. Das entwickelte Tool besitzt eine Implementierung beider Varianten. Die automatische Analyse sollte mit Vorsicht genutzt werden, da sie mit unter nicht die erwünschten Ergebnisse liefert. Die Gründe dafür werden in 4.1.2 näher beleuchtet. Folgend werden die Schritte für die Erstellung von Modulen und Sockets beschrieben.

4.1.1 Vorarbeit

Module werden zu Beginn von Hand in der Engine gebaut und als ein Prefab abgespeichert. Diese Prefabs können grundsätzlich frei gestaltet werden, mit der Ausnahme von zwei Bedingungen: Alle Module dürfen das Volumen eines vorher festgelegten Würfels nicht überschreiten und alle Namen der erstellten Prefabs müssen unterschiedlich sein. Sowohl die Prefabs, als auch die Bestandteile der Prefabs werden je in einen konfigurierbaren Ordner in der Engine abgelegt. Dieser Vorgang wird in 4.3.3 näher erklärt. Die Prefabs können daraufhin vom Tool eingelesen werden und dienen dem Algorithmus als Module.

4.1.2 Automatische Analyse eines Moduls und der Sockets

Jedes Modul wird als Würfel betrachtet. Die sechs Außenseiten bilden die Sockets des Moduls. An den Seiten des Würfels wird jeweils überprüft welche 3D Modelle des Moduls anliegend sind. Danach werden von jedem Modell die Vertices ermittelt, die auf einer Außenseite liegen und dem entsprechenden Socket zugeordnet. Das Prefab wird als Modul im Tool gespeichert. Jeder Socket muss intern einmalig sein, daher werden neue Sockets mit allen vorhandenen Sockets verglichen. Sollte es zu einer Übereinstimmung kommen, wird der neue Socket gelöscht und der vorhandene Socket im Modul referenziert. Der Vergleich basiert auf dem Positionsvergleich der Vertices beider Sockets. Stimmen die Anzahl der Vertices und die Positionen auf der gleichen Ebene überein, sind beide Sockets gleich. Sollte kein Vergleich zu einer Übereinstimmung kommen wird der neue Socket in dem Modul als Referenz gespeichert und dient als einmaliger Socket für weitere Vergleiche. Leichte Unterschiede in der Modellierung der Objekte oder nicht exakte Platzierungen in der Unity Engine führen zu Fehlern in der Analyse. Dieser Bestandteil sollte daher vor einer ausgiebigen Nutzung verbessert werden und individuellen Anforderungen stärker angepasst werden.

4.1.3 Input der Modulbeziehung durch eine Json Datei

Neben der automatischen Analyse gibt es die Möglichkeit Module, Sockets und die Beziehungen auch manuell einzugeben. Dafür gibt es eine JSON Datei (das Einlesen der Datei ist in 4.3.3 beschrieben). Die Datei der Evaluation ist in .2 angehangen.

Die Datei beginnt mit einem Array dessen Name ”Modules“ ist. Darin befinden sich alle Module, welche im Verlauf der Datei beschrieben werden. Die einzelnen Modulbeschreibungen werden über einen Identifier aus dem ”Modules“ Array benannt (jeder Name muss einmalig sein). In einer Modulbeschreibung gibt es ein Array für seine Sockets. Die Reihenfolge ist dabei wichtig und gibt die Sockets der Reihe nach von ”links“, ”hinten“, ”rechts“, ”vorne“, ”oben“ und ”unten“ an. Der zweite Teil einer Modulbeschreibung ist das Attribut ”Name“. Der Inhalt gibt den Dateinamen eines Prefabs an. Es ist wichtig das die Namen übereinstimmen und einmalig sind, damit Modulbeschreibungen in der Datei mit den Prefabs der Enginge übereinstimmend verwendet werden. Der letzte Bestandteil der Datei ist ein Array mit dem Namen ”Sockets“, darin müssen alle individuellen Sockets mit einer eindeutigen ID erwähnt werden. Darunter befinden sich alle Sockets (mit dem Identifier aus dem ”Sockets“ Array) als eigenes Array. Die Liste ergibt jeweils alle Sockets die an dieses Array platziert werden können. Durch diese Angabe kann tritt eine einseitige Zuordnung auf, welche in 4.2.1 näher erklärt wird.

4.1.4 Ausgabe der Daten

Nach dem einlesen der Daten sind Module und Sockets einmalig. Sie werden in einem Dictionary gespeichert, in dem Sockets auf alle Module zeigen, welche die Sockets benutzen. Sockets werden verfierfacht. Jedes Duplikat wird um 90°, 180° oder 270° um die y-Achse gedreht. Die Reihenfolge der Module wird dabei geändert und die Duplikate sind nicht untereinander oder mit dem original gleich. Durch diese Drehung und Rotation wird in der späteren Platzierung die Wahl des passenden Moduls für einen Platz erleichtert.

4.2 Generierung der Level

Die Generierung der Level erfolgt während des ”Eidtmode“ in Unity. Die Prefabs werden dabei in einer Szene instantiiert. Die Module werden in einem Grid platziert, welches durch den Anwender angepasst werden kann. Außerdem wird das Grid visualisiert, damit die Ausdehnung des entstehenden Levels und die Größer der Module erkannt werden kann. Näheres zur Anwendung wird in 4.3.3 beschrieben.

Im folgenden Abschnitt wird der konkrete Ablauf des WFC Algorithmus in dem Tool beschrieben. Immer wenn hierbei auf einen Zufall hingewiesen wird, ist damit die pseudo Zufallsmethode der Unity Engine gemeint. Im zweiten Teil des Kapitels wird kurz auf die verwendeten Module eingegangen und ihre Bedeutung erklärt.

4.2.1 Veränderungen am WFC Algorithmus

Der Ablauf des WFC Algorithmus wurde in seinen wichtigsten Schritten unverändert in den 3D Kontext umgesetzt. Alle Änderungen sind klein und werden hier im Detail erklärt.

1. Zu erst wird der Schritt "Observe" ausgeführt. In diesem Schritt wird überprüft, welches Gitterfeld die geringste Entropie[1] aufweist. Sollten mehrere (oder zu Beginn alle) Felder die gleiche Entropie besitzen wird zufällig ein Feld für den "Collapse" Schritt gewählt. Sind alle Felder belegt, bricht der Algorithmus ab. Sollte es dazu kommen, dass ein Feld eine Entropie von 0 besitzt, muss der Algorithmus nicht abbrechen. Sollte die Option "Use Fallback Prefab" aktiviert sein, wird an der Position ein festgelegtes Prefab platziert. Ein Fehler tritt auf, wenn es Kombinationen von Nachbarsockets gibt, die kein Modul erfüllen kann.

2. Nachdem ein Feld ausgewählt wurde, wird auf diesem ein zufälliges Modul der verfügbaren Module platziert.

3. Der Schritt "Propagate" untersucht, basierend auf dem letzten "Collapse" Feld, anliegende Felder auf eine Änderung ihrer Liste der Module, welche sie platzieren können. Die Änderung basiert dabei auf den Sockets der platzierten Module. Sockets legen (wie in 4.1.3) fest, wer die möglichen Nachbarn sind. Es kann im Falle des JSON Inputs entschieden werden, dass eine solche Nachbarschaft nur einseitig erreicht werden soll (die Reihenfolge für die Platzierung der Module ist dabei entscheidend). Sollten bei Nachbarn, eines eben platzierten Moduls, einige der möglichen Module ausgeschlossen werden, werden sie aus dieser Liste entfernt. Damit verringert sich die Entropie (Anzahl der platzierbaren Module für ein Feld) der Module weiter. Sobald die Änderungen zu allen Feldern propagiert wurden, wird der Schritt "Observe" erneut ausgeführt.

[1] Entropie ist hierbei die Anzahl der insgesamt verfügbaren Modulen, die platziert werden können.

4. Zuletzt werden die Module instantiiert. Der User kann danach am erstellten Level weiterarbeiten, es speichern oder es löschen und ein neues generieren.

4.2.2 Bedeutung der Module

Für die Durchführung der Evaluation wurden insgesamt neun Module erstellt. Die ersten sechs davon sind erstellt für die Optik eines Levels. Jedes der Module stellt dabei einen Teil dar, der in einem Level benutzt wird. .3.1 ist normaler Boden. Dieser kann an .3.2, .3.4, .3.5 oder .3.6 angelegt werden. Die Module .3.2 bis .3.6 sind Bestandteile Wände oder Bestandteile von diesen. Die verschiedenen Ausprägungen ermöglichen eine größere Vielfalt der Levelgestaltung und ermöglichen sowohl offene Flächen, als auch Gänge und Korridore. Das Modul .3.7 stellt einen Spawn Punkt dar. .3.8 ist ein Objective und .3.9 ist ein Choke Point. Choke Points können nur an ein Objective platziert werden. Dadurch sollen nach Möglichkeit Choke Points immer in Verbindung und am besten auch auf den Wegen zu Objectives stehen. Objectives sind mit allen anderen Modulen kompatibel. In CSGO liegen diese sowohl an Wänden, als auch an offenen Flächen, daher kann bei ihnen keine klare Zuordnung getroffen werden. Spawn Punkte verhalten sich in ihrer Kompatibilität so wie normaler Boden. Die exakten Zuordnungen können .2 mit der Erklärung von 4.1.3 entnommen werden.

4.3 UI und UX des Tools

Das Tool ist ein ”Monobehaviour“ Skript in der Unity Engine. Dieses wurde durch einen ”Custom Inspector“ erweitert. Um das Tool zu benutzen muss das ”WaveFunctionCollapseManager“ Skript einem Gameobject hinzugefügt werden. Das folgende Kapitel beschäftigt sich mit dem Aufbau und der Bedienung des Tools.

4.3.1 Aufbau

Das Tool ist Grundsätzlich in 3 Teile aufgeteilt: Modul Einstellungen, Einstellungen für weitere Teile des Tools und Button mit Funktionen. Es gibt mehrere Pfadanga-

ben, welche alle relative Pfade zum ”Resources“ in Unity darstellen. Die Abbildung 1 zeigt den Aufbau der UI.

4.3.2 Modul Einstellungen

Die Modul Einstellungen sind auf drei verschiedene Zahlen beschränkt. Die ”Width“ legt die Ausdehnung eines Moduls im Raum fest. ”Units X“ und ”Units Y“ legen dabei die Größe des Gitters auf der xz-Koordinatenebene von Unity fest.

4.3.3 Erstellung von Konfigurationen

Der Menüpunkt ”Creator“ befasst sich mit der automatischen Erstellung von Modulen. Dieser kann Prefabs für die Nutzung als Module analysieren, wie in 4.1 beschrieben wurde. Dafür werden in das Array ”Possible Prefabs“ alle Prefabs für die Modulanalyse gespeichert. ”Base Prefabs“ sind alle Bestandteile der Module. Beide Listen können über den Button ”Load Resources“ aus ihren Ordnern mit den Pfadangaben automatisch hinzugefügt werden. Der Wert für ”Vertex Difference“ soll die Findung passender Sockets verbessern. Dabei werden Vertices die Abweichend sind bis zur angegebenen Zahl ignoriert (eine nähere Beschreibung ist unter 4.1.2 zu finden).

Sollte die Konfiguration über eine JSON Datei genutzt werden kann, muss ein Pfad für das Attribut ”Json Path“ angegeben werden. Außerdem müssen die ”Possible Prefabs“ wie in der automatischen Analyse dem Creator hinzugefügt werden. Mit dem Button ”Read from Json“ wird die Datei eingelesen. Module und Sockets werden erstellt und nötige Datenstrukturen für den WFC Algorithmus bereitgestellt (4.1.3).

Wird unter dem Attribut ”Parent“ ein Gameobject zugewiesen, werden Prefabs nach ihrer Erstellung diesem als ”Child“ Objekte zugewiesen.

4.3.4 Visualisierung

Das Gitter kann in der Unity Engine visualisiert werden. Dafür muss nur der Button ”Show Grid“ betätigt werden, diese Aktion fügt ein weiteres Script dem Gameobject

hinzu. Die Modul Einstellungen werden daraufhin auf das Gitter übertragen. Einstellungen dazu finden sich unter dem Punkt "Grid". Es kann die Farbe des Grids geändert werden (weitere Komponenten werden automatisch erkannt). Sollte man, nachdem der Button gedrückt wurde, das Gitter verstecken wollen, so kann man "Draw Grid" auf false setzen. Für die Visualisierung wurde das "Aline"[2] Package aus dem Unity Asset Store verwendet.

4.3.5 Ausführung des WFC Algorithmus

Der WFC Algorithmus wird über das Drücken des "Start Wfc" Buttons gestartet. Er benutzt dabei alle Einstellungen aus dem darüber liegenden Editor.

Sollte man eine langsamere Variante bevorzugen, kann man den "Step Mode" aktivieren. Dabei wird der Algorithmus Schritt für Schritt ausgeführt. Sollte dies aktiviert sein, wird der Algorithmus mit der "Continue" Taste fortgesetzt.

Bei Problemen kann optional über "Use Fallback Prefab" ein Modul "Fallback Prefab" angegeben werden, welches bei einem Fehlschlag des Algorithmus platziert wird.

Sollte man ein generiertes Level löschen wollen, kann man die "Clear Grid" Taste drücken.

4.3.6 Speichern eines Levels

Sollte man ein generiertes Level speichern wollen, so kann ein ScriptableObject vom Typ "Save Level" benutzt werden. Dieses wird in das Attribut "Save Level Data" gezogen. Über die Taste "Save Level" wird es abgespeichert. Ein Level zu laden funktioniert über ein vorher befülltes "Save Level" ScriptableObject indem die Taste "Load Level" betätigt wird.

[2] https://assetstore.unity.com/packages/tools/gui/aline-162772

4.4 Erkennung der Anforderungserfüllung

Die gestellten Anforderungen aus 2.2.1 werden automatisch analysiert. Dazu wurde ein weiteres Tool in Unity geschrieben, welches hier näher beleuchtet wird. Alle Daten werden über zwei CSV Dateien exportiert.

4.4.1 UI der Anforderungserfüllung

Das Interface des Scriptes zur Ermittlung der Anforderungserfüllung ist unter 2 abgebildet. Zur Durchführung ist die Zuweisung einiger Attribute im Inpektor notwendig. Das "Navmesh Surface" muss eine Referenz auf ein Skript "Navmesh Surface" der Unity Engine besitzen. Der "Path Tester" muss das Prefab "Tester beinhalten", dieser beinhaltet ein "Capsule" Gameobject und eine Navmeshagent. Außerdem ist eine Referenz auf den "Wave Function Collapse Manager", sowie eine Pfadangabe zum speichern der CSV Datei unter "CSV Path". "Count" kann angegeben werden um iterative Tests in der CSV zu nummerieren. Die eingegebene Zahl wird dabei nach jedem Durchlauf incrementiert. Der Button "Log Results" wechselt zum "Playmode" der Engine und sammelt die benötigten Daten. Danach werden die Daten in die CSV Datei geschrieben und Unity wechselt in den "Editmode".

4.4.2 Anforderungsermittlung der Spawn Punkte und Objectives

Die entwickelten Module für Spawn Punkte und Objectives ermöglichen es ihre Anzahl zu ermitteln, indem ihnen ein Skript zum Zählen anhängt. Die ermittelte Anzahl sollte demnach nicht die Anzahl in den Anforderungen von 2.2.1 überschreiten.

4.4.3 Anforderungsermittlung der Wege zwischen Spawn Punkt und Objectives

Um die Wege zu ermitteln, werden von jedem Spawnpunkt Pfade über das integrierte Navmesh System von Unity generiert. Gibt es mögliche Wege für alle Spawn Punkte zu allen Objectives, müssen auch alle Spawn Punkte miteinander verbunden sein. Die Anzahl der Fehlschläge wird in der CSV gespeichert.

4.4.4 Anforderungsermittlung der Choke Points auf den Wegen

Die Pfade der Wege geben ein Array von Vektoren zurück. Daraus können zwischen den Vektoren Geraden gebildet werden. Liegt ein Punkt auf einer dieser Geraden innerhalb der halben Ausdehnung eines Moduls (gemessen vom Mittelpunkt des Moduls), verläuft der Weg durch einen Choke Point. Die Anzahl der Wege mit Choke Point werden daraufhin in der CSV festgehalten.

5 Evaluation

Das implementierte Tool wird in diesem Kapitel evaluiert. Dabei werden die Ergebnisse visualisiert und mit den Anforderungen eines Levels von CS:GO verglichen. Vorher wird die Laufzeit und die Fehlerrate für Generierung und Analyse der Level ermittelt, um mögliche technische Probleme in der Nutzung aufzudecken. Daraufhin wird interpretiert wie sinnvoll der Einsatz bei immer größeren Leveln ist. Zum Schluss werden Vor- und Nachteile des Tools und seiner Ergebnisse beleuchtet.

5.1 Versuchsaufbau

Der WFC Algorithmus ist größtenteils zufallsbasiert. Für die Ermittlung der Anforderungserfüllung wurden daher drei verschiedene Ausdehnungen des Grids über die Variablen "Units X" und "Units Y" verwendet. Damit soll ermittelt werden, ob diese Einstellungen einen Unterschied in der Qualität der Level zur Folge haben. Um die Zuverlässigkeit des Algorithmus zu testen, werden für jeden Versuch 30 Level generiert. Mögliche Abweichungen können dadurch klassifiziert werden und wenn nötig in die Betrachtungen eingeschlossen werden. Die Einstellungen sehen wie folgt aus:

1. Versuchsdurchlauf 4x4: "Units X" und "Units Y" sind beide auf den Wert vier eingestellt. Ein generiertes Level besteht aus 16 Modulen.
2. Versuchsdurchlauf 10x10: "Units X" und "Units Y" sind beide auf den Wert zehn eingestellt. Ein generiertes Level besteht aus 100 Modulen.
3. Versuchsdurchlauf 20x20: "Units X" und "Units Y" sind beide auf den Wert 20 eingestellt. Ein generiertes Level besteht aus 400 Modulen.

Alle anderen Einstellungen sind gleich, damit keine neuen Fehlerquellen auftreten können oder bestehende Fehlerquellen jeweils die gleiche Auswirkung auf alle Testergebnisse besitzen.

Die Größe des Gitters zu verändern, bringt den Vorteil der Unabhängigkeit des Parameters. Die anderen veränderbaren Parameter sind die Veränderung der JSON (also Änderung der Socket- und Modulbeziehungen) oder die Veränderung der Anzahl und der Art der Sockets. Die Qualitätsänderung eines Levels durch Größenänderung eines Gitters sollte aufgrund der in 4 erwähnten Ähnlichkeit unbeeinflusst durch andere Parameter sein. Obwohl der Einfluss des gewählten Parameters, im 2D Kontext durch eine schwache Ähnlichkeit bei großen Ausgabe Bildern, eher gering ausfällt [Gum16]. Im 3D Kontext dieser Zusammenhang aufgrund des Inputs über eine JSON Datei überprüft.

5.2 Laufzeit- und Fehlerentwicklung

In diesem Kapitel werden grundlegende Daten der Ausführung des Algorithmus in den drei Versuchsreihen präsentiert und ausgewertet. Das Ziel ist es dabei die Skalierbarkeit des Algorithmus zu ermitteln und Probleme für die weitere Betrachtung zu erkennen oder ausschließen zu können. Dazu werden Fehler des Algorithmus der Größe eines Levels gegenübergestellt.

5.2.1 Vergleich der Laufzeit für die Generierung eines Levels in Abhängigkeit der Größe eines Levels

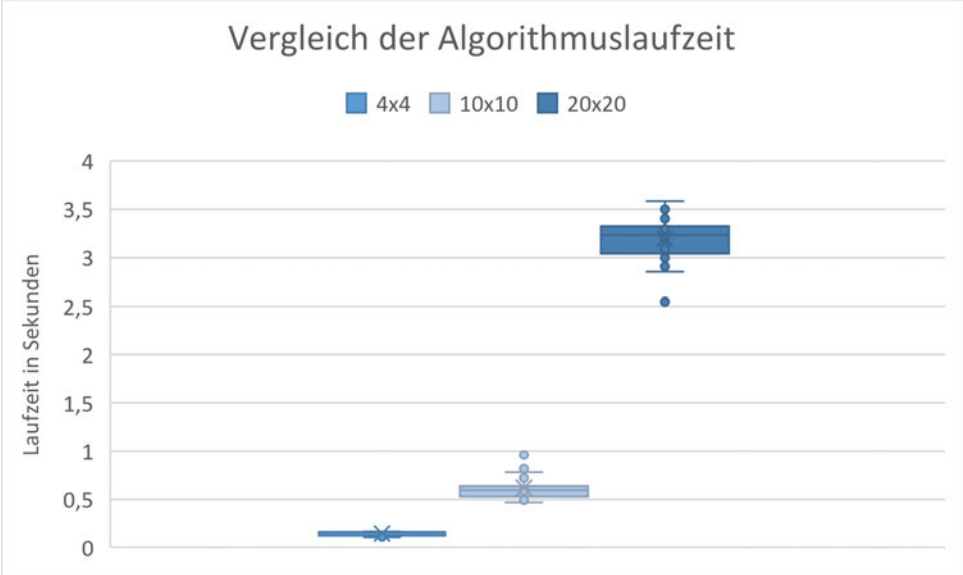

Abbildung 5.1: Durchschnittliche Laufzeit für die Generierung eines Levels der drei Versuchsreihen in Abhängigkeit der Größe des generierten Levels gegenüber gestellt.

Wie in 5.1 zu erkennen ist, steigt die Laufzeit des Algorithmus nahezu linear an im Vergleich zur Größe des generierten Levels. Der Aufwand bleibt daher abschätzbar und ein Einsatz in Größeren Anwendungen ist möglich.

5.2.2 Vergleich der Laufzeit für die automatische Analyse eines generierten Levels in Abhängigkeit der Größe eines Levels

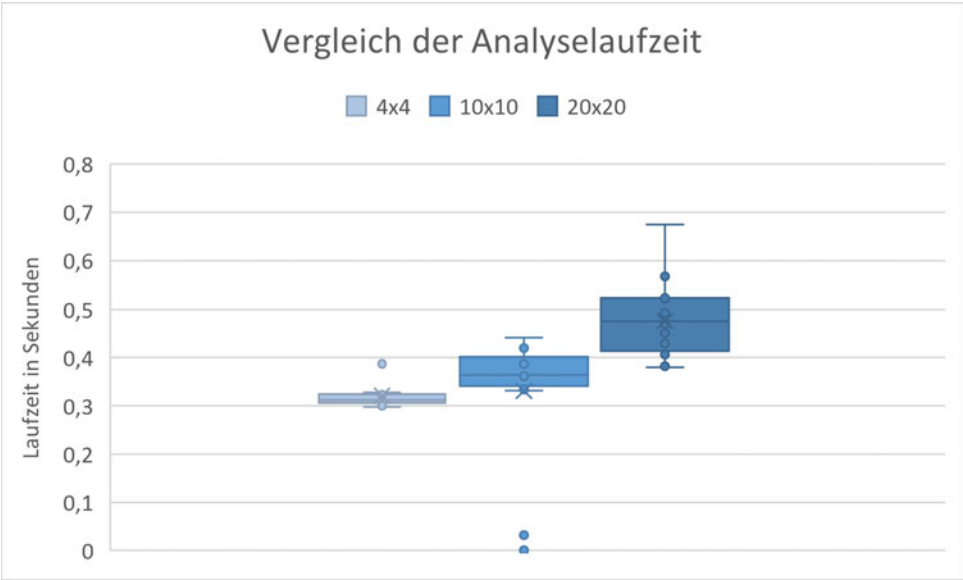

Abbildung 5.2: Durchschnittliche Laufzeit für die automatische Analyse eines Levels der drei Versuchsreihen in Abhängigkeit der Größe des generierten Levels gegenüber gestellt.

Die Laufzeiten für die Analyse der Level (5.2) sind signifikant kürzer als die Laufzeiten welche für die Generierung der Level nötig waren. Die Entwicklung der Laufzeit lässt sich bei der Analyse wie bei der Generierung als linear steigend einordnen.

5.2.3 Vergleich der Fehlerrate in Abhängigkeit der Größe eines Levels

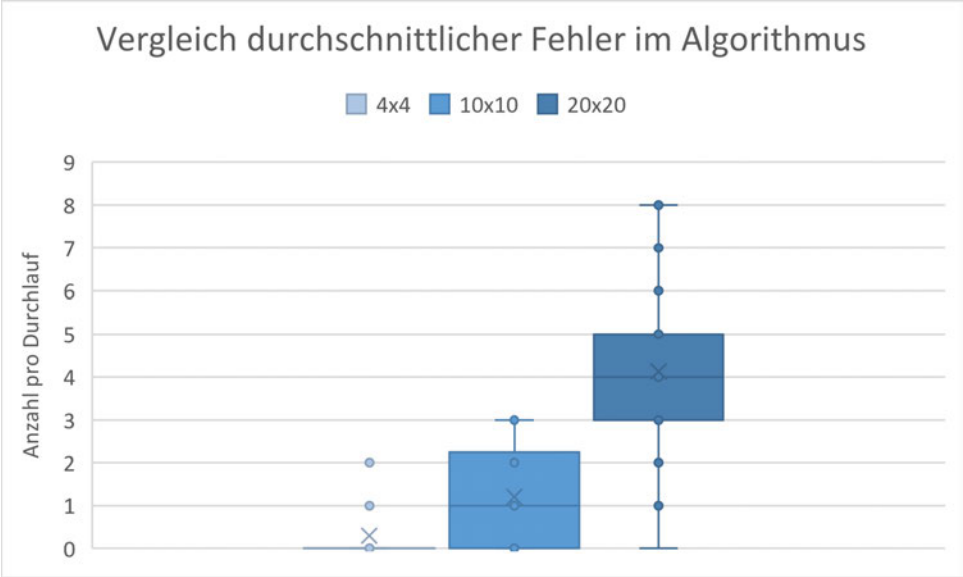

Abbildung 5.3: Durchschnittliche Anzahl von Fehlern pro generiertem Level in Abhängigkeit von der Größe des generierten Levels.

Die Anzahl der entstandenen Fehler für die Generierung eines Levels steigt in etwa linear über alle Versuchsreihen an. Ein Fehler tritt auf, falls der WFC Algorithmus ein Modul im Gitter nicht besetzt und durch die Platzierung seiner Nachbarmodule keine Auswahl für die Platzierung eines Moduls besitzt (4.2.1). In diesem Fall wurde das einfache "Boden" Modul platziert. Die durchschnittliche Fehlerrate im 1. Durchgang beträgt 0%, im 2. Durchgang 1% und im 3. Durchgang 4%. Sie steigt in den Durchgängen nicht stark an und ermöglicht einen Vergleich der Level. Der Einfluss auf die Ergebnisse der Versuchsreihen bleibt gering und die Erfüllung der Anforderungen kann evaluiert werden.

Die Ursache für die Fehler liegt in der Zusammenstellung der Modul- und Socketbeziehungen. Weitere Anpassungen könnten die Fehlerrate beeinflussen. Durch die geringe Auswirkung auf die Ergebnisse wird für die Evaluation darauf verzichtet.

5.2.4 Fazit

Die Generierung eines Levels und die automatische Analyse sind lösbare Probleme und können mit linear steigendem Aufwand abgeschätzt werden. Der Einsatz des erstellten WFC Tools wird nicht durch die Implementation aufgehalten oder behindert. Die Fehlerrate des Algorithmus steigt über alle Versuchsreihen nahezu konstant und beeinträchtigt die Ergebnisse nur gering.

5.3 Auswertung Anforderungserfüllung

Nachdem zuvor die Fehlerrate und Laufzeit des Algorithmus für die drei Versuchsreihen analysiert wurde, wird dieses Kapitel die quantitative Analyse der Ergebnisse behandeln. Die Erfüllung der Anforderungen steht im Zentrum. Heatmaps werden eingesetzt um die Verteilung von Generierten Strukturen zu visualisieren.

5.3.1 Platzierung der Spawn Punkte

Die Platzierung der Spawn Punkte wird für jede Versuchsreihe in .6.1, .6.2 und .6.3 gezeigt. Dabei können teilweise Felder erkannt werden, welche häufiger einen Spawn Punkt zugewiesen bekommen, allerdings ist die Wahrscheinlichkeit dafür nie über 50%. Es sollte daher nicht davon ausgegangen werden, dass diese Felder für einen Versuch besonders günstig liegen. Das Diagramm .6.5 zeigt, dass jeder Gitterpunkt im Verlauf der Versuchsreihe zwischen 2 und 5 (bzw. 6) Spawn Punkte zugewiesen bekommt. Dieser Bereich variiert nur leicht zwischen den Versuchsreihen. Die Abbildung .6.4 gibt an, wie viele Spawn Punkte durchschnittlich bei einer Ausführung in einer Versuchsreihe platziert werden. Anhand dieser Daten kann ein linearer Anstieg im Vergleich zur Größe des Levels festgestellt werden.

Zusammenfassend ergibt sich, dass die Platzierung der Spawn Punkte verteilt über fast alle Gitterpunkte geschieht. Es ist kein eindeutiges Muster in der Platzierung erkennbar. Die Anforderung kann teilweise von dem Versuchsdurchlauf 4x4 erfüllt werden. Die durchschnittliche Anzahl generierter Spawn Punkte liegt bei 2,03. Der Wert schwankt dabei zwischen 1 und 3. Die Anzahl kann in diesem Fall im Mittel erfüllt werden. Gegen die vollständige Erfüllung spricht das Fehlen eines Musters

in der Platzierung. Spawn Punkte können sowohl nebeneinander liegen, als auch gegenüber. Die Anforderung kann daher nicht zuverlässig von dem Durchlauf 4x4 erfüllt werden. Die Versuchsreihen 10x10 und 20x20 sind noch unzuverlässiger, da die Anzahl der generierten Spawn Punkte insgesamt steigt. Die Größe des generierten Level scheint dabei einen Einfluss zu nehmen, auf die Anzahl der Spawn Punkte, jedoch nicht auf die Platzierung.

5.3.2 Platzierung der Objectives

Die Abbildungen .6.6, .6.7 und .6.8 zeigen die Heatmaps der platzierten Objectives für jede der drei Versuchsreihen. Daraus ergeben sich viele Gemeinsamkeiten mit den Spawn Punkten, auch hier ist kein klares Muster identifizierbar.

Anhand von .6.9 kann festgestellt werden, dass die Anzahl der platzierten Objectives über alle Testreihen linear ansteigt im Vergleich zum Größenanstieg des Levels. Trotz des Anstieges bleibt die Wahrscheinlichkeit, bei der ein Gitterfeld mit einem Objective belegt wird fast gleich (vgl. .6.10). Es gibt einige stärker abweichende Punkte, bei denen Felder öfter oder seltener belegt wurden, allerdings sind diese Felder Einzelfälle und können nicht genutzt werden, um tatsächlich ein Muster zu erkennen.

Die Anforderung wird auch in diesem Durchlauf am ehesten von dem Versuchsdurchlauf 4x4 gelöst. Hier werden im Mittel 4 Objectives platziert. Das ist im Vergleich zur Anforderung doppelt so viel, aber die Abweichung ist im Vergleich zu den anderen Testläufen nur sehr gering. Dabei lassen sich auch hier Gemeinsamkeiten im Vergleich zu den Spawn Punkten finden. Der Durchlauf 4x4 kommt der Erfüllung der Anforderung am nächsten, scheitert aber vorrangig an der Platzierung und der Anzahl der platzierten Module. Der Unterschied liegt darin, dass Objectives häufiger als Spawn Punkte generiert werden.

5.3.3 Verbindung der Spawn Punkte und Objectives

Die Verbindung zwischen Spawn Punkten und Objectives ist in allen Versuchsreihen möglich gewesen. Es konnten keine Fälle beobachtet werden, in denen es nicht

möglich war. Das Diagramm .6.11 zeigt dies. Die Anforderung kann damit erfüllt werden.

5.3.4 Choke Points zwischen Spawn Punkten und Objectives

Die Platzierung von ChokePoints kann wie bei Objectives und Spawn Punkten aus den drei Heatmaps .6.12, .6.13 und .6.14 entnommen werden. Dabei fällt auf, dass wesentlich weniger Fläche genutzt wird, als bei den anderen beiden generierten Modulen. Dies beweist auch .6.15 und .6.16. Die Anzahl der generierten Choke Points pro Gitterfeld, sowie die Anzahl insgesamt generierter Gitterfelder mit Choke Points fallen wesentlich geringer aus. Die Ursache dafür liegt vermutlich in der Socketbeziehung des Moduls. Es erfordert ein anliegendes Objective. Bei der Betrachtung der Heatmaps der Objectives und der Choke Points fällt auf, dass es eine kleinere Korrelation von häufigen Objectives und häufigne Choke Points in der Umgebung gibt.

Die Anforderung der Choke Points auf dem Weg zwischen Spawn Punkte und Objectives kann nicht erfüllt werden. Es konnte in keinem Durchlauf diese Anforderung erfüllt werden .6.17.

5.4 Fehleranalyse

In den Durchführungen und Auswertungen der Versuchsreihen konnte festgestellt werden, dass die Anforderungen zum Platzieren der Spawn Punkte, Objectives und Choke Points (vgl. 2.2.1) nicht erfüllt werden konnten. Die Anforderung der Verbindung zwischen Spawn Punkten und Objectives konnte erfüllt werden. Dieses Kapitel beschäftigt sich mit den Ursachen und möglichen Ansätzen zur Verbesserung der Werte.

5.4.1 Ursachen

Es gibt zwei Ursachen für die nicht Erfüllung der Anforderungen.

1. Die Einstellung der kompatiblen Sockets und Module.

2. Die Funktionsweise des Algorithmus.

Die Konfiguration der kompatiblen Sockets kann dem Anhang dieser Arbeit und die Beschreibung aus 4.1.3 entnommen werden. Sie nimmt Einfluss auf Möglichkeiten der Platzierung von Modulen. Choke Points müssen immer neben einem Objective platziert werden, für Objectives gilt diese Bedingung nicht. Würde man an den Einstellungen Änderungen vornehmen, wäre es indirekt möglich die Anzahl an platzierten speziellen Modulen zu beeinflussen. Dadurch könnten Nebeneffekte wie höhere Fehlerraten des Algorithmus auftreten. Um gewünschte Ergebnisse zu erzielen, wäre umfassende Studie zum Einfluss von Parametern in der Konfiguration nötig. Eine einfache Lösung bietet das verändern der Einstellungen nicht, da das gesamte Gefüge der Beziehung sich ändern kann und Platzierungen eventuell stark beeinflussen würde. Die Einstellungen zu ändern würden das Ergebnis sehr wahrscheinlich beeinflussen, aber sie stellen keinen einfachen Regler dar, um die Qualität zuverlässig zu verbessern.

Die Funktionsweise des Algorithmus ist im Kern zufallsbasiert. Es gibt nur sehr eingeschränkt die Option mit ihm zu kommunizieren. Beeinflussung funktioniert nur über die Größe des Level oder die Einstellung kompatibler Sockets und Module. Die gegebenen Parameter sind nicht in der Lage zuverlässig ein Level mit den Anforderungen zu sinnvoller Positionierung und Anzahl zu erfüllen.

5.4.2 Verbesserung der generierten Level

Um die Anforderungen besser zu erfüllen wären mehr Parameter notwendig, um den Vorgang des Algorithmus zu beeinflussen. Diese Wege wurden teilweise in [Nun20] erforscht. Hierbei wären zwei wichtige zusätzliche Anforderungen die Implementation der Parameter ”Abstand“ und ”Anzahl“. Damit sollten Distanzen und Häufigkeiten bei der Platzierung berücksichtigt werden. Zu nah aneinander liegende Level-Bestandteile würden vermieden. Zu viele Bestandteile einer bestimmten Kategorie würden dadurch nicht in einem Level sein. Zusätzlich könnte auch der Parameter für Position eingeführt werden, um bestimmte Bereiche eines Levels für einen Bestandteile schon vorher zu blockieren oder ihn gezielt auf eine Position im Gitter setzen.

6 Zusammenfassung und Ausblick

Der WFC Algorithmus wurde implementiert in der Unity Engine und evaluiert in Hinsicht auf seinen Einsatz im Leveldesign. Um die Zuverlässigkeit zu Prüfen wurden Anforderungen an ein Level des Spiels CSGO genutzt. Der Algorithmus ist nicht in der Lage innerhalb von drei Versuchsreihen alle gestellten Anforderungen zu erfüllen. Zuverlässig erwies er sich nur in einer einzigen Anforderung. Er wäre für den untersuchten Zweck kein geeignetes Hilfsmittel. Um die Arbeit abzuschließen wird in diesem Kapitel noch ein mal auf Möglichkeiten eingegangen, wie der Algorithmus mit Verbesserungen eine Einsatzmöglichkeit in dem untersuchten Szenario darstellen könnte. Außerdem wird die Arbeit zusammengefasst und ein finales Fazit gezogen.

6.1 Verbesserungen für zukünftige Einsätze

Einen Teil der Verbesserungen wurden schon in 5.4.2 dargestellt. Allgemein fehlt es dem Algorithmus an Möglichkeiten seine Entscheidungen zu beeinflussen. Mehr Parameter wären eine Möglichkeit seine Entscheidungsfreiheit einzuschränken. Eine weitere Möglichkeit wäre die Erweiterung mit einem Leveldesign Tool. Das Tool könnte es ermöglichen bestimmte Punkte vor dem Erstellen eines Levels zu platzieren und den WFC Algorithmus als Werkzeug zum ausfüllen freier Flächen zu benutzen. Bei einer solchen Nutzung würde der Algorithmus weniger aktiv in den Design Prozess treten und eher eine Nachbereitung der geleisteten Arbeit durchführen. Bei diesen Verbesserungen muss allerdings immer die Fehlerrate des Algorithmus bedacht werden. Sollte diese bei zu vielen Modifikationen steigen, könnten die Ergebnisse für den Anwender unbrauchbar werden. Abschließend wäre eine solche Weiterentwicklung auch von seiner konkreten Aufgabe abhängig. Der Algorithmus könnte in einer Weiterentwicklung die Anforderungen dieser Arbeit erfüllen. Daraus würde vermutlich eine Version resultieren, welche nicht in der Lage wäre unterschiedliche Aufgaben zu erfüllen. Allgemeine Problemstellungen sollten eventuell mit anderen

Algorithmen untersucht werden oder den WFC Algorithmus nur in kleinen Schritten modifizieren und seine Weiterentwicklung beobachten.

6.2 Fazit

Der Wave Function Collapse Algorithmus ist ein für die Bildsynthese entwickelter Algorithmus. Durch seine spezielle Funktionsweise und einfache Implementation scheint es ein geeignetes Werkzeug für die prozedurale Levelgenerierung zu sein. Um dies zu prüfen wurden Anforderungen an ein Level des Videospiel Counter Strike: Global Offensive verwendet und getestet, ob diese von dem WFC Algorithmus erfüllt werden könnten. Durch die Evaluation konnte die Erfüllung nur einer von vier Anforderungen nachgewiesen werden. Die Anforderungen sind sehr grundlegend an ein Level gewählt. Durch das Verfehlen der meisten Grundlagen, kann ein Einsatz in dem gewählten Themenfeld nicht empfohlen werden. Der WFC Algorithmus schafft es nicht zuverlässig zielgerichtete Aufgaben im Leveldesign zu übernehmen. Durch Modifikationen könnte er zu einem Werkzeug werden, mit dem diese Aufgaben erfüllbar wären.

Glossar

Choke Point Ein Choke Point ist im Kontext eines Shooter Games (wie CSGO) ein Bereich in dem das Level einen schmalen Durchgang besitzt und oftmals einen besonders umkämpften Punkt in einem Level darstellt. 10

Counter Strike: Global Offensive Counter Strike: Global Offensive ist ein Multiplayer Shooter Videospiel entwickelt von Valve. 2

Objective Ein Objective ist ein Ort im Level eines Videospiel, der mit einer Aufgabe für den Spieler in Verbindung steht. 10

prozedurale Levelgenerierung Erstellen eines virtuellen Videospiellevels durch Benutzung eines Algorithmus. 2

Spawn Punkt Ein Spawn Punkt (auch Spawn Point genannt) ist ein Ort oder Bereich in einem Videospiel, an dem Spieler das Spiel betreten. 10

Steam Ein digitaler Online Shop für Videospiele auf dem PC. 12

Wave Function Collapse Der Wave Function Collapse Algorithmus ist ein Bildsynthese Algorithmus und wird im Zusammenhang dieser Arbeit als Werkzeug zum erzeugen prozeduraler Level benutzt. 2

Literaturverzeichnis

[BL20] Rongdong Diao Bo Lin, Wassim Jabi: *Urban Space Simulation Based on Wave Function Collapse and Convolutional Neural Network*, *SimAUD2020*, 2020.

[csg] *Dust II*, URL: https://counterstrike.fandom.com/wiki/Dust_II, besucht am 22.11.2023.

[Gal13] Alex Galuzin: *How to Design Multiplayer Gameplay Map Layouts (Complete In-Depth Guide)*, Dez. 2013, URL: https://www.worldofleveldesign.com/categories/csgo-tutorials/csgo-how-to-design-gameplay-map-layouts.php, besucht am 20.09.2023.

[Gum16] Maxim Gumin: *WaveFunctionCollapse*, Sept. 2016, URL: https://github.com/mxgmn/WaveFunctionCollapse/, besucht am 22.09.2023.

[JT16] Mark J. Nelson Julian Togelius, Noor Shaker: *Procedural Content Generation in Games*, Springer, 2016.

[KLL+19] Hwanhee Kim, Seongtaek Lee, Hyundong Lee, Teasung Hahn und Shinjin Kang: *Automatic Generation of Game Content using a Graph-based Wave Function Collapse Algorithm*, *2019 IEEE Conference on Games*, S. 1–4, 2019.

[KS22] Isaac Karth und Adam M. Smith: *WaveFunctionCollapse: Content Generation via Constraint Solving and Machine Learning*, *IEEE Transactions on Games*, Bd. 14(3):S. 364–376, 2022.

[LWB18] Sylvain Lefebvre, Li-Yi Wei und Connelly Barnes: *Informational Texture Synthesis*, März 2018, working paper or preprint.
URL https://inria.hal.science/hal-01706539

[Men14] Enzo Menegazzi: *A CS:GO Level Design concept:the pathways*, Sept. 2014, URL: https://www.gamedeveloper.com/design/a-cs-go-level-design-concept-the-pathways, besucht am 20.09.2023.

[Mor21] Quentin Edward Morris: *Modifying Wave Function Collapse for more Complex Use in Game Generation and Design*, 5 2021.

[Nun20] Nuno J. Nunes (Hg.): *Automatic Generation of Game Levels Based on Controllable Wave Function Collapse Algorithm*, Bd. 19 von *14*, The organization, Springer Nature Switzerland AG, Gewerbestrasse 11, 6330 Cham, Switzerland, Nov. 2020, an optional note.

[RS91] Olaf Pfeiffer Reinhard Scholl: *Natur als Fraktale Grafik*, Markt&Technik Verlag AG, 1991, ISBN 3-87791-013-0.

[Stå18] Oskar Stålberg: *Wave Function Collapse in Bad North*, Juli 2018, URL: https://www.youtube.com/watch?v=0bcZb-SsnrA, besucht am 19.11.2023.

[TYSB11] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley und Cameron Browne: *Search-Based Procedural Content Generation: A Taxonomy and Survey*, *IEEE Transactions on Computational Intelligence and AI in Games*, Bd. 3(3):S. 172–186, 2011.

Anhang

Grundlagen

.1 Anforderungen an das Leveldesign

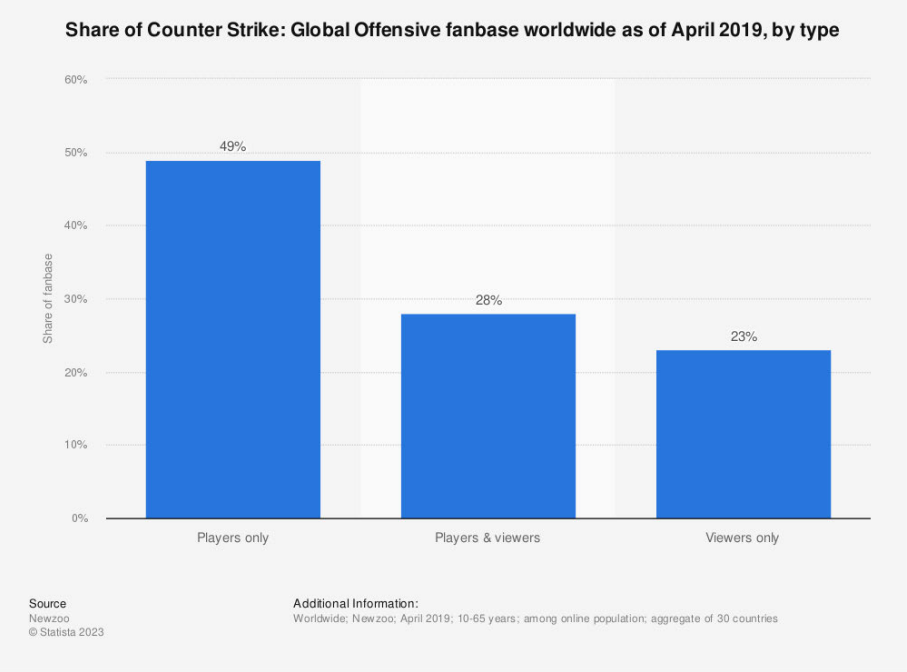

Abbildung .1.1: CSGO besitzt eine breite Interessengemeinschaft. Fast ein Viertel sind nur Zuschauer. Es ist nicht nur unter Spielern beliebt, sondern erreicht auch darüber hinaus eine Fangemeinschaft.

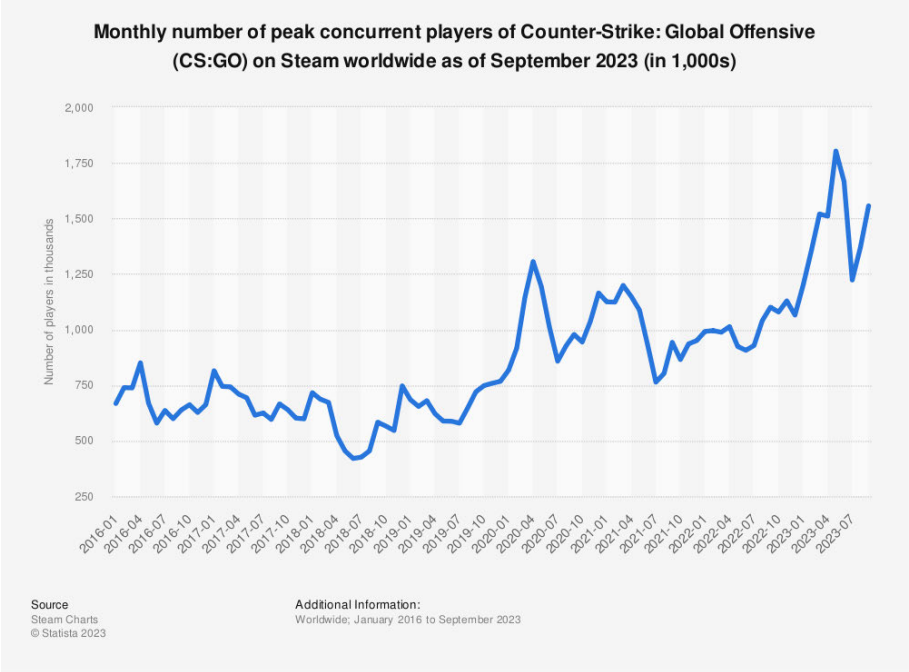

Abbildung .1.2: Die Spielerzahl steigt seit 2016 an und erreicht mittlerweile oft über eine Million gleichzeitige Spieler zur Bestzeit im Monat.

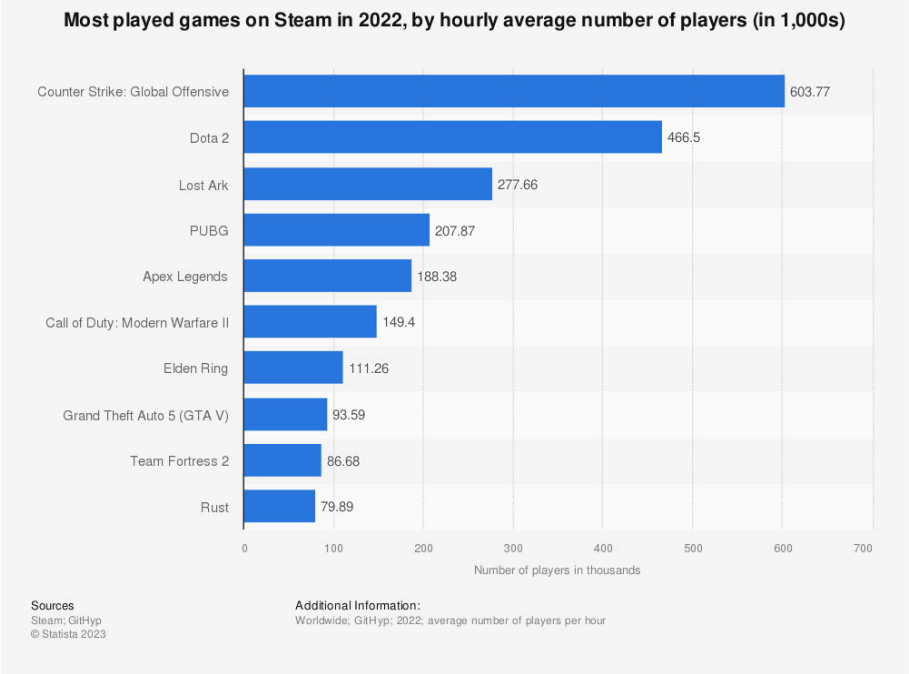

Abbildung .1.3: CSGO ist das Spiel in das die gesamte Spielerschaft die meisten addierten Stunden auf Steam investiert hat.

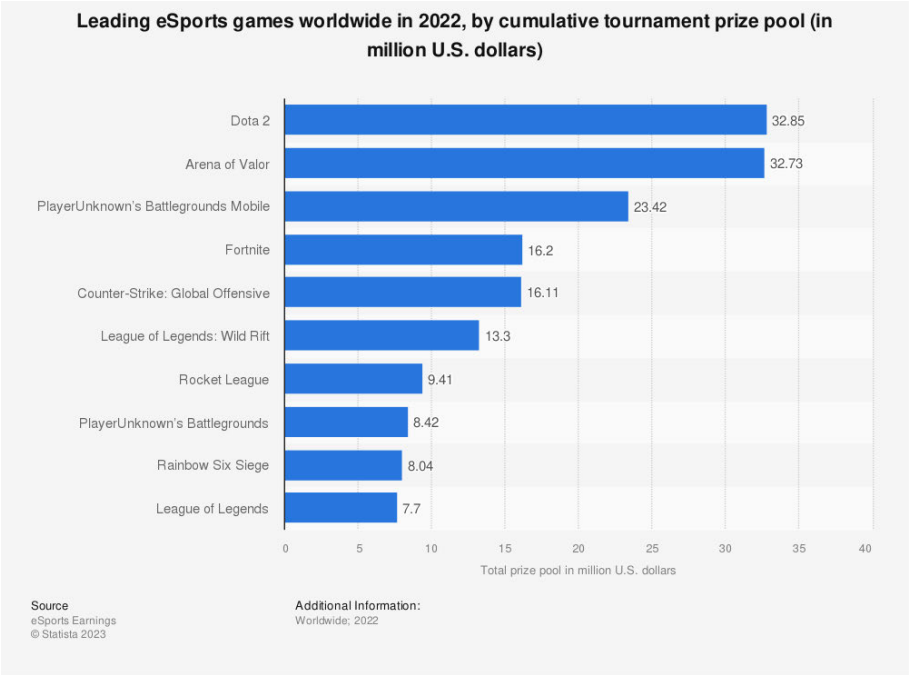

Abbildung .1.4: CSGO ist eines der E-Sport Spiele mit den höchsten Preisgeldern.

A4

Implementierung

.2 Implementierung der Analysefunktion

```
{
  "Modules": [
    "Prf1",
    "Prf2",
    "Prf3",
    "Prf4",
    "Prf5",
    "Prf6",
    "Prf7",
    "Prf8",
    "Prf9"
  ],
  "Prf1": {
    "Sockets": [
      "S1",
      "S1",
      "S1",
      "S1",
      "S0",
      "S0"
    ],
    "Name": "Prf1"
  },
  "Prf2": {
    "Sockets": [
      "S1",
```

```
27            "S5",
28            "S3",
29            "S6",
30            "S0",
31            "S0"
32          ],
33          "Name": "Prf2"
34      },
35      "Prf3": {
36          "Sockets": [
37            "S2",
38            "S4",
39            "S3",
40            "S3",
41            "S0",
42            "S0"
43          ],
44          "Name": "Prf3"
45      },
46      "Prf4": {
47          "Sockets": [
48            "S1",
49            "S1",
50            "S7",
51            "S8",
52            "S0",
53            "S0"
54          ],
55          "Name": "Prf4"
56      },
57      "Prf5": {
58          "Sockets": [
59            "S1",
60            "S1",
61            "S9",
62            "S6",
```

```
63          "S0",
64          "S0"
65        ],
66        "Name": "Prf5"
67      },
68      "Prf6": {
69        "Sockets": [
70          "S1",
71          "S5",
72          "S10",
73          "S1",
74          "S0",
75          "S0"
76        ],
77        "Name": "Prf6"
78      },
79      "Prf7": {
80        "Sockets": [
81          "S11",
82          "S3",
83          "S12",
84          "S3",
85          "S0",
86          "S0"
87        ],
88        "Name": "Prf-Choke"
89      },
90      "Prf8": {
91        "Sockets": [
92          "S13",
93          "S13",
94          "S13",
95          "S13",
96          "S0",
97          "S0"
98        ],
```

```
99        "Name": "Prf-Objective"
100     },
101     "Prf9": {
102       "Sockets": [
103         "S1",
104         "S1",
105         "S1",
106         "S1",
107         "S0",
108         "S0"
109       ],
110       "Name": "Prf-Spawn"
111     },
112     "Sockets": [
113       "S0",
114       "S1",
115       "S2",
116       "S3",
117       "S4",
118       "S5",
119       "S6",
120       "S7",
121       "S8",
122       "S9",
123       "S10",
124       "S11",
125       "S12",
126       "S13"
127     ],
128     "S0": [
129       "S0"
130     ],
131     "S1": [
132       "S1",
133       "S9",
134       "S10",
```

```
135         "S13"
136     ],
137     "S2": [
138         "S4",
139         "S5",
140         "S7",
141         "S13"
142     ],
143     "S3": [
144         "S3",
145         "S9",
146         "S10",
147         "S13"
148     ],
149     "S4": [
150         "S2",
151         "S6",
152         "S8",
153         "S13"
154     ],
155     "S5": [
156         "S6",
157         "S2",
158         "S8",
159         "S13"
160     ],
161     "S6": [
162         "S4",
163         "S5",
164         "S7",
165         "S13"
166     ],
167     "S7": [
168         "S2",
169         "S6",
170         "S8",
```

```
171         "S13"
172     ],
173     "S8": [
174         "S5",
175         "S4",
176         "S7",
177         "S13"
178     ],
179     "S9": [
180         "S3",
181         "S1",
182         "S10",
183         "S13"
184     ],
185     "S10": [
186         "S3",
187         "S1",
188         "S9",
189         "S13"
190     ],
191     "S11": [
192         "S1",
193         "S6",
194         "S5",
195         "S7",
196         "S8",
197         "S13"
198     ],
199     "S12": [
200         "S13"
201     ],
202     "S13": [
203         "S1",
204         "S2",
205         "S3",
206         "S4",
```

```
207         "S5",
208         "S6",
209         "S7",
210         "S8",
211         "S9",
212         "S10",
213         "S11",
214         "S12"
215     ]
216 }
```

.3 Generierung der Level

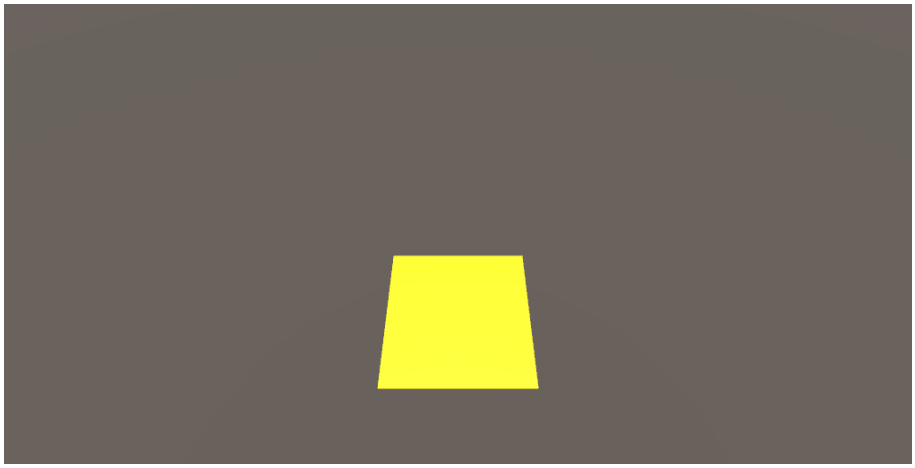

Abbildung .3.1: 1. Modul in der Unity Engine. Es stellt flachen Boden dar.

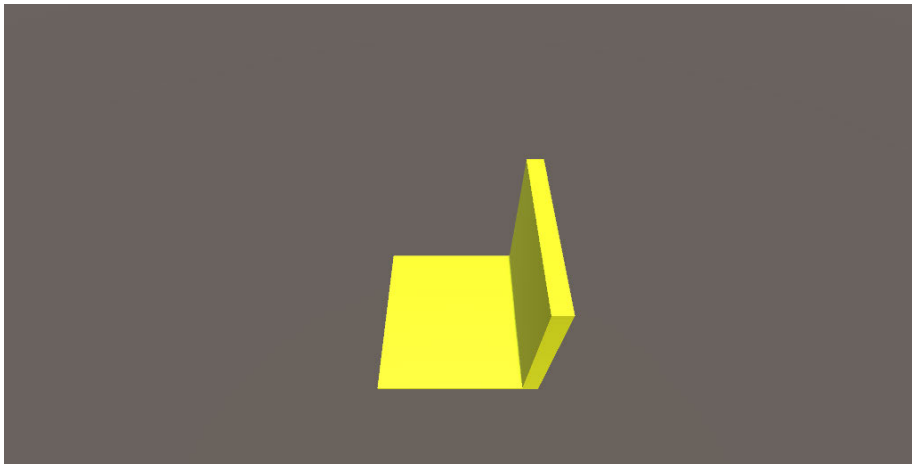

Abbildung .3.2: 2. Modul in der Unity Engine. Es stellt flachen Boden mit einer Wand an einer Seite dar.

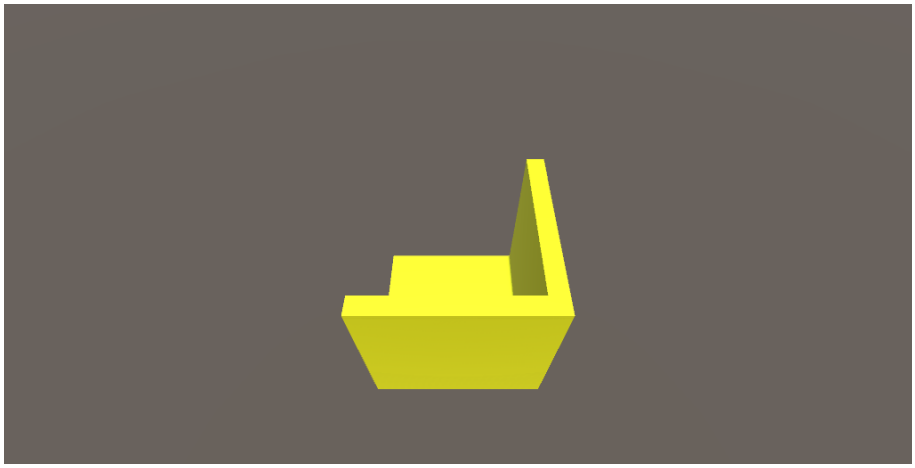

Abbildung .3.3: 3. Modul in der Unity Engine. Es stellt flachen Boden mit zwei Wänden dar, welche einen Eckpunkt zweier Wände bilden.

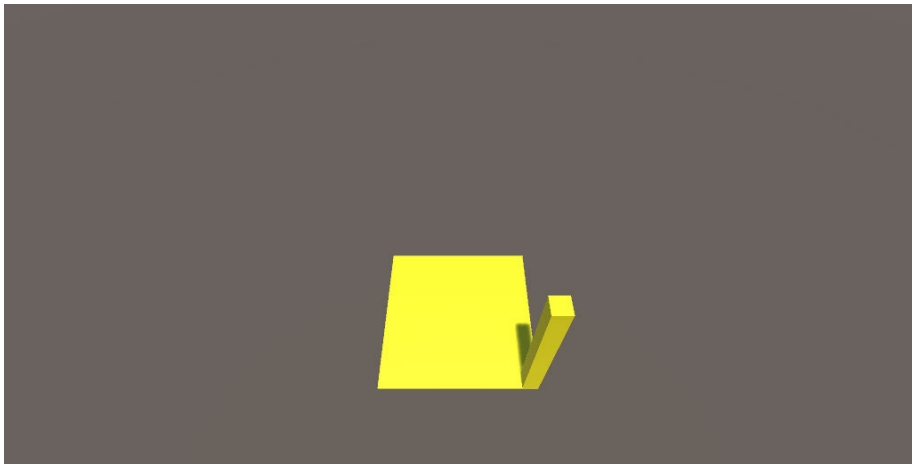

Abbildung .3.4: 4. Modul in der Unity Engine. Es stellt flachen Boden mit einer Säule in einer Ecke dar, diese wird benutzt um Wände zu vereinigen.

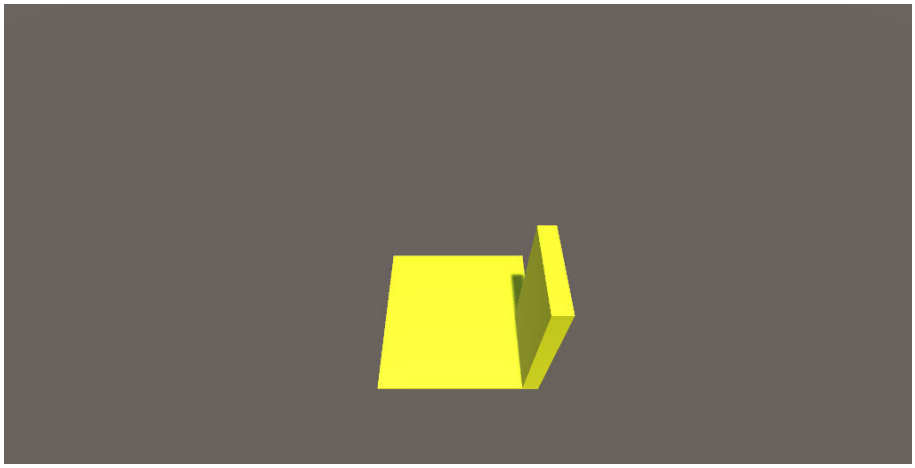

Abbildung .3.5: 5. Modul in der Unity Engine. Es stellt flachen Boden mit einer halben Wand dar.

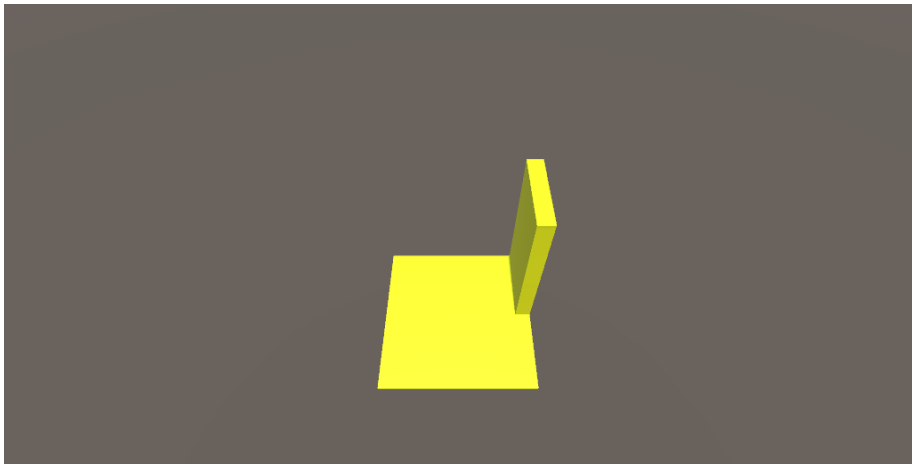

Abbildung .3.6: 6. Modul in der Unity Engine. Es stellt flachen Boden mit einer halben Wand dar. Im Vergleich zum 5. Modul ist die Wand auf der anderen Seite.

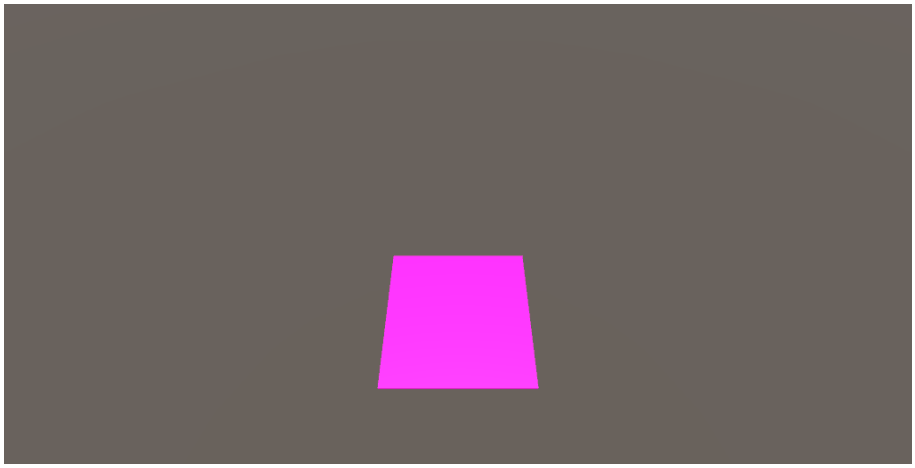

Abbildung .3.7: Spawn Modul in der Unity Engine. Es stellt flachen Boden dar, auf dem Spieler starten sollen.

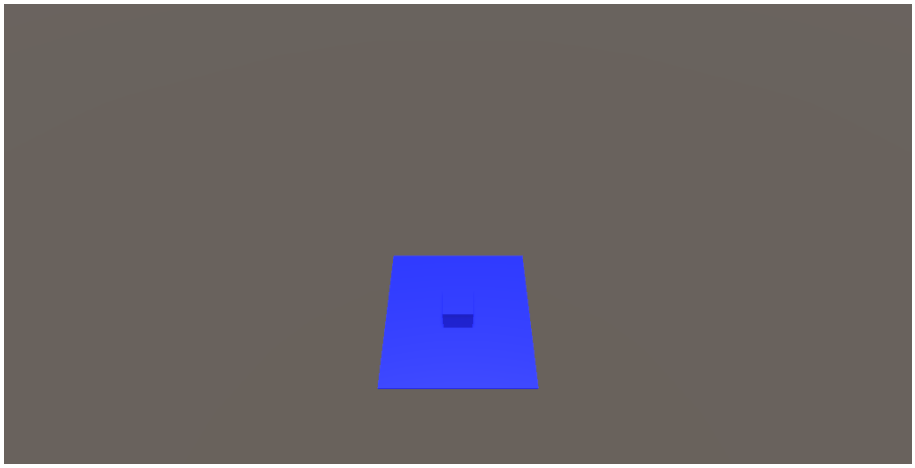

Abbildung .3.8: Objective Modul in der Unity Engine. Es stellt flachen Boden dar und besitzt in der Mitte einen Würfel, der den Punkt für Interaktion eines Objective darstellen soll.

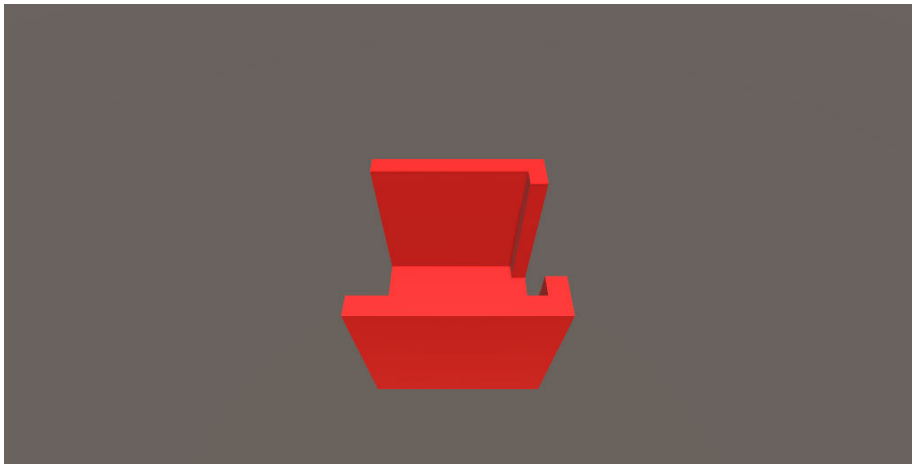

Abbildung .3.9: Choke Point Modul in der Unity Engine. Es stellt flachen Boden dar und bildet einen kleinen Corridor. Es soll mit der rechten Seite in ein Objective münden.

.4 UI und UX des Tools

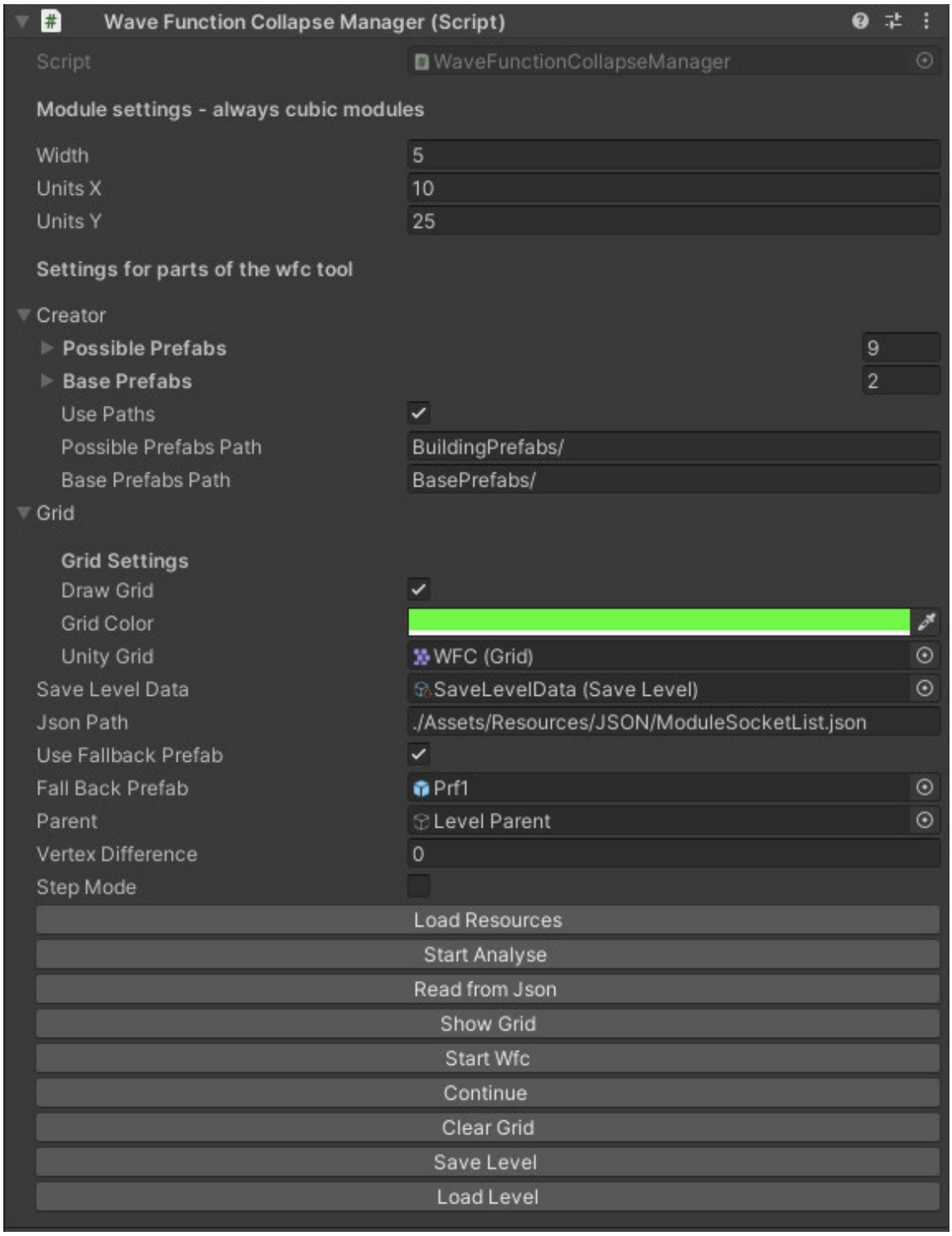

Abbildung 1: User Interface des WFC Tools. Es gibt im oberen Teil viele Einstellungen für den Algorithmus, die Analyse und Visualisierung. Die Buttons sind für die Interaktion zuständig.

.5 Erkennung der Anforderungserfüllung

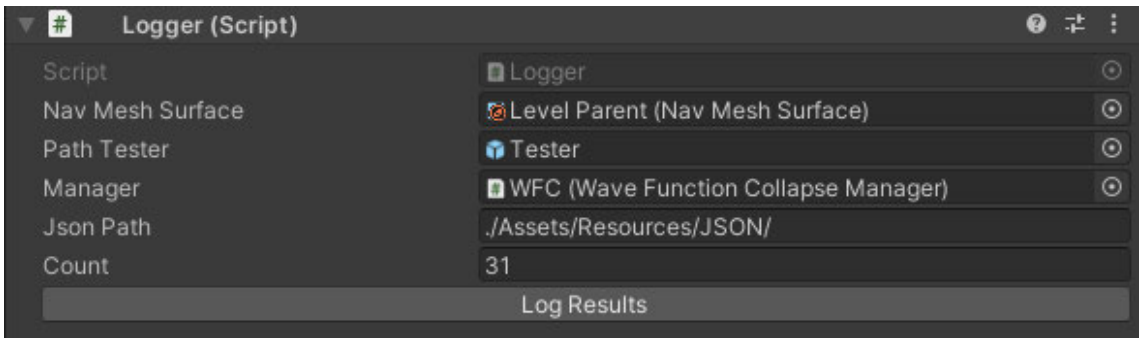

Abbildung 2: User Interface des Analyse Tools für den WFC Aglorithmus.

Evaluation

.6 Auswertung Anforderungserfüllung

.6.1 Platzierung der Spawn Punkte

0	x	1	2	3	4
y	0	2	7	12	17
1	2	2	3	6	2
2	7	3	3	7	3
3	12	6	2	7	2
4	17	5	6	2	2

Abbildung .6.1: Heatmap zur Verteilung der Spawn Punkte in der Versuchsreihe 4x4. In grau ist der Index der Module in der Heatmap markiert. Die grünen Achsen ist der Mittelpunkt der Platzierung eines Moduls in der Unity Engine. Die blauen Flächen zeigen die Module und die Anzahl der Platzierungen in den 30 Durchläufen.

0	x	1	2	3	4	5	6	7	8	9	10
y	0	2	7	12	17	22	27	32	37	42	47
1	2	3	2	1	2	4	4	6	2	3	2
2	7	2	1	3	2	1	3	2	2	3	4
3	12	6	2	4	2	3	7	4	1	3	4
4	17	4	6	3	3	2	7	5	7	4	6
5	22	4	3	5	0	7	3	3	7	7	7
6	27	1	4	5	5	3	1	6	5	6	7
7	32	6	3	7	4	6	2	4	5	7	4
8	37	1	1	4	3	1	4	3	2	1	0
9	42	3	5	0	3	3	4	4	6	5	7
10	47	1	1	4	4	3	3	4	3	6	8

Abbildung .6.2: Heatmap zur Verteilung der Spawn Punkte in der Versuchsreihe 10x10. In grau ist der Index der Module in der Heatmap markiert. Die grünen Achsen ist der Mittelpunkt der Platzierung eines Moduls in der Unity Engine. Die blauen Flächen zeigen die Module und die Anzahl der Platzierungen in den 30 Durchläufen.

0	x	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
y	0	2	7	12	17	22	27	32	37	42	47	52	57	62	67	72	77	82	87	92	97
1	2	5	5	3	4	5	4	4	4	6	6	2	1	1	4	2	4	2	3	1	1
2	7	3	6	5	3	5	5	3	3	3	5	5	2	4	2	1	2	4	5	4	4
3	12	1	1	6	7	1	3	6	5	3	6	9	3	5	3	3	4	3	6	4	4
4	17	2	0	3	1	4	3	4	3	3	0	2	4	4	2	3	2	0	1	0	4
5	22	5	2	5	7	4	3	3	1	3	6	5	6	4	5	1	4	3	7	6	3
6	27	5	1	5	6	4	6	3	1	3	1	1	5	2	4	5	4	3	5	4	4
7	32	2	0	5	3	3	5	5	2	2	5	3	3	3	2	3	4	3	5	6	3
8	37	3	5	5	4	4	4	3	5	3	6	2	6	3	6	3	4	1	2	6	4
9	42	0	8	2	4	4	3	3	1	0	5	3	4	3	0	0	1	1	5	5	7
10	47	3	5	4	7	5	1	2	3	6	6	4	2	2	3	4	3	5	2	4	3
11	52	4	6	3	3	5	4	2	7	3	1	3	4	2	7	6	4	3	3	5	1
12	57	4	3	4	3	5	2	1	7	4	6	6	8	7	1	2	4	4	3	7	8
13	62	3	4	5	0	2	3	4	3	2	6	3	2	4	2	4	0	2	4	4	4
14	67	2	3	3	6	5	3	7	2	7	6	2	1	5	3	4	3	2	1	1	3
15	72	4	4	5	8	7	2	5	5	6	3	2	4	5	5	2	4	4	1	3	5
16	77	5	2	3	3	0	9	0	2	4	1	3	3	2	3	2	3	6	5	3	5
17	82	6	3	7	9	3	3	5	4	2	0	5	3	4	6	2	7	3	5	4	2
18	87	2	4	1	3	3	4	2	4	3	6	3	3	2	2	2	7	1	2	7	3
19	92	4	3	2	3	4	2	2	6	5	3	6	5	4	4	2	2	5	5	5	6
20	97	6	5	1	4	4	4	1	3	3	4	4	5	6	5	6	4	3	7	3	1

Abbildung .6.3: Heatmap zur Verteilung der Spawn Punkte in der Versuchsreihe 20x20. In grau ist der Index der Module in der Heatmap markiert. Die grünen Achsen ist der Mittelpunkt der Platzierung eines Moduls in der Unity Engine. Die blauen Flächen zeigen die Module und die Anzahl der Platzierungen in den 30 Durchläufen.

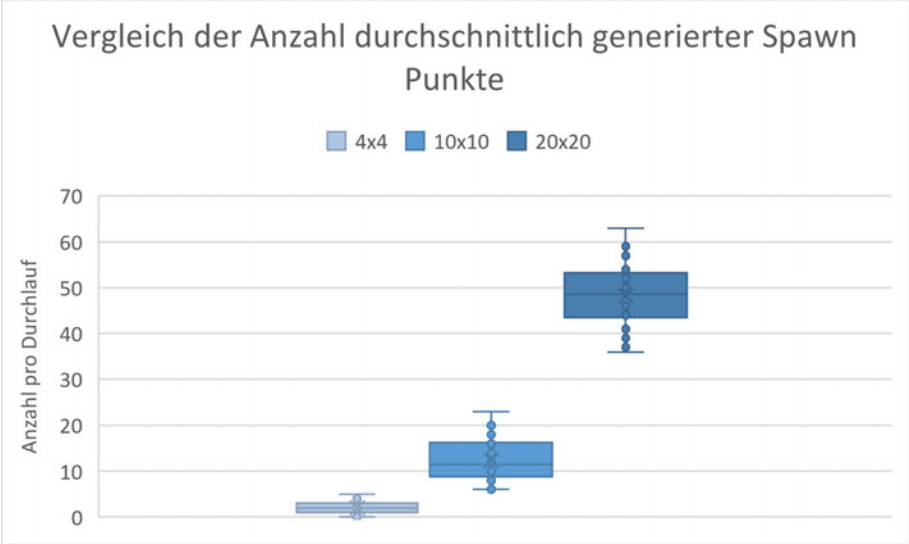

Abbildung .6.4: Vergleich der Anzahl durchschnittlich platzierter Spawn Punkte in einen Durchlauf im Vergleich mit allen Testläufen. Die Anzahl der Spawn Punkte im Durchlauf 4x4 kommt im Mittel sehr nah an die Anforderung. Die Anzahl steigt linear, wodurch größere Level eher ungeeignet sind für den hier betrachteten Use Case sind.

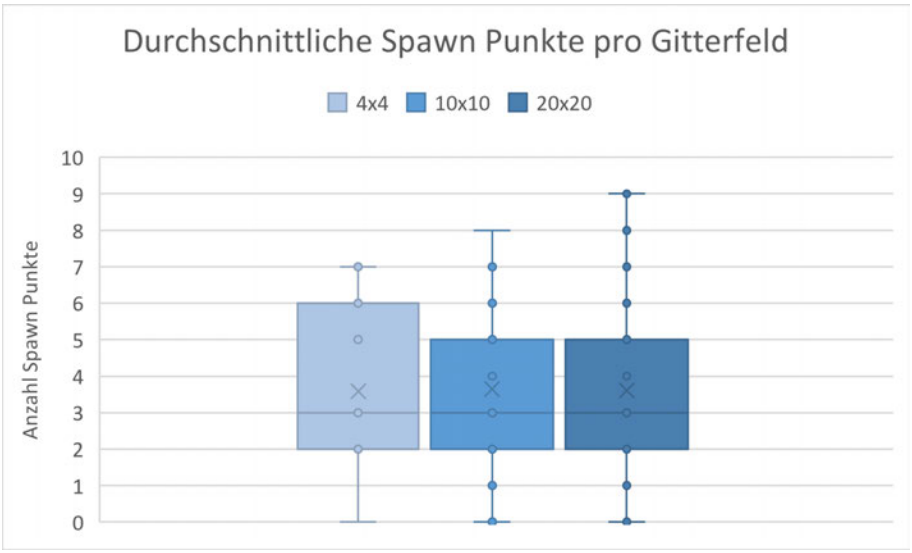

Abbildung .6.5: Vergleich der Anzahl platzierter Spawn Punkte pro Gitterfeld über die 30 Testläufe in den verschiedenen Testreihen. Die Werte weichen nur minimal voneinander ab. Die Größe hat keinen Einfluss auf die Wahrscheinlichkeit für die Platzierung eines Spawn Punktes auf einem Gitterfeld.

0	x	1	2	3	4
y	0	2	7	12	17
1	2	8	6	6	12
2	7	5	6	7	5
3	12	5	13	5	4
4	17	6	8	10	7

Abbildung .6.6: Heatmap zur Verteilung der Objectives in der Versuchsreihe 4x4. In grau ist der Index der Module in der Heatmap markiert. Die grünen Achsen beziehen sich auf den Mittelpunkt der Platzierung eines Moduls in der Unity Engine. Die blauen Flächen zeigen die Module und die Anzahl der Platzierungen in den 30 Durchläufen.

0	x	1	2	3	4	5	6	7	8	9	10
y	0	2	7	12	17	22	27	32	37	42	47
1	2	4	8	3	11	4	11	4	6	8	11
2	7	9	7	18	2	8	6	12	3	10	7
3	12	8	6	3	14	4	10	4	8	6	6
4	17	6	5	9	5	14	4	8	7	7	8
5	22	4	11	7	8	6	9	7	4	8	7
6	27	8	5	7	6	7	9	7	6	10	4
7	32	10	8	6	10	6	9	7	10	6	6
8	37	7	8	7	6	9	8	10	7	6	3
9	42	5	4	8	9	7	7	9	6	7	8
10	47	6	10	4	11	10	6	8	10	8	5

Abbildung .6.7: Heatmap zur Verteilung der Objectives in der Versuchsreihe 10x10. In grau ist der Index der Module in der Heatmap markiert. Die grünen Achsen beziehen sich auf den Mittelpunkt der Platzierung eines Moduls in der Unity Engine. Die blauen Flächen zeigen die Module und die Anzahl der Platzierungen in den 30 Durchläufen.

0	x	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
y	0	2	7	12	17	22	27	32	37	42	47	52	57	62	67	72	77	82	87	92	97
1	2	7	8	7	8	7	6	4	6	6	5	11	7	8	9	7	10	11	6	6	10
2	7	8	5	9	7	13	7	10	9	6	10	6	7	6	5	10	11	5	6	5	10
3	12	6	10	5	8	11	8	8	2	13	8	7	6	6	9	10	5	6	8	3	5
4	17	9	6	6	8	7	10	3	12	1	8	10	7	6	9	6	6	7	7	9	7
5	22	7	7	7	9	4	6	8	9	9	6	8	8	4	10	9	7	9	6	8	8
6	27	13	10	8	3	12	3	6	8	7	10	8	5	8	4	8	8	2	11	5	10
7	32	6	5	10	9	7	8	8	8	6	6	9	9	2	14	5	10	12	1	9	5
8	37	8	6	5	7	7	9	3	9	9	7	6	4	12	7	7	3	6	10	6	9
9	42	10	3	14	5	9	5	11	8	11	4	12	4	7	7	7	11	9	4	8	5
10	47	9	5	4	10	5	11	4	9	6	7	7	7	10	2	10	8	10	7	9	6
11	52	8	9	7	7	8	7	9	4	6	8	3	11	8	7	6	11	4	8	9	8
12	57	6	5	7	9	5	8	10	8	6	4	11	5	8	7	16	4	9	9	5	5
13	62	3	7	9	8	10	4	8	6	8	13	4	8	7	8	3	12	7	9	9	7
14	67	16	4	10	4	6	12	5	6	8	5	7	12	3	6	6	10	9	7	7	9
15	72	6	10	5	8	4	4	8	7	8	9	6	8	12	6	10	4	6	9	6	4
16	77	7	11	7	9	8	6	9	7	8	8	10	6	10	6	8	7	8	6	8	9
17	82	3	11	4	5	6	6	6	7	8	7	2	11	6	5	7	8	5	7	8	8
18	87	11	2	13	3	9	8	10	5	11	8	10	5	8	7	9	5	8	10	8	9
19	92	7	12	2	10	4	7	6	8	3	8	5	6	8	7	5	12	4	6	6	3
20	97	5	6	6	7	10	8	7	6	9	10	8	4	9	7	8	2	12	5	7	9

Abbildung .6.8: Heatmap zur Verteilung der Objectives in der Versuchsreihe 20x20. In grau ist der Index der Module in der Heatmap markiert. Die grünen Achsen beziehen sich auf den Mittelpunkt der Platzierung eines Moduls in der Unity Engine. Die blauen Flächen zeigen die Module und die Anzahl der Platzierungen in den 30 Durchläufen.

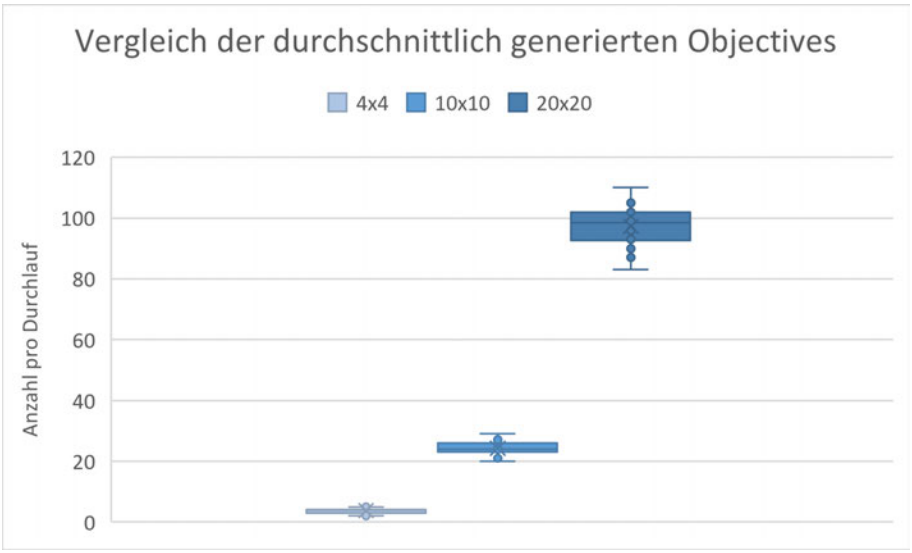

Abbildung .6.9: Vergleich der Anzahl durchschnittlich platzierter Objectives in einen Durchlauf im Vergleich mit allen Testläufen. Die Anzahl der Objectives im Durchlauf 4x4 kommt im Mittel den Anforderungen am nächsten. Der lineare Anstieg führt zu schlechteren Ergebnissen für die Durchläufe 10x10 und 20x20.

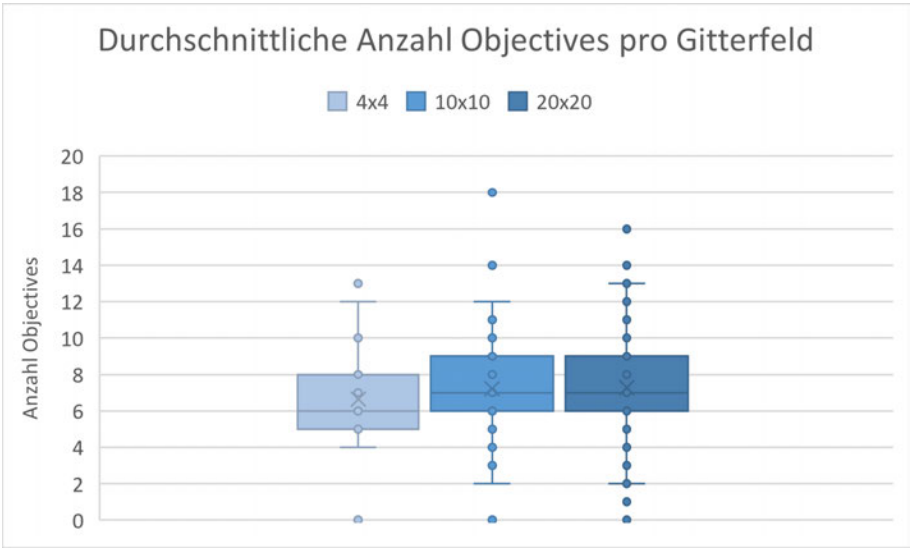

Abbildung .6.10: Vergleich der Anzahl insgesamt platzierter Objectives pro Gitterfeld über die 30 Testläufe in den verschiedenen Testreihen. Die Werte sind sich sehr ähnlich und die Größe des generierten Levels scheint keinen großen Einfluss auf das Ergebnis zu haben.

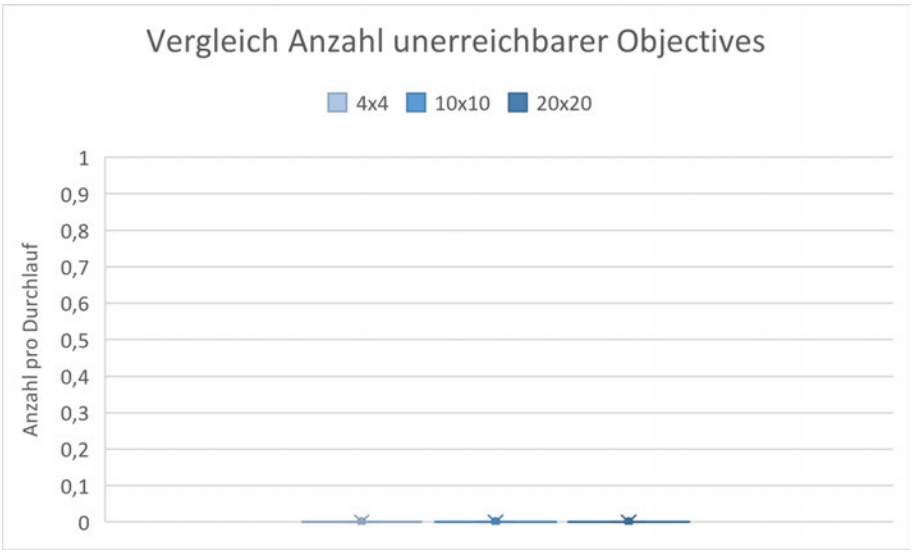

Abbildung .6.11: Anzahl der Objectives, welche nicht von einem Spawn Punkt erreicht werden können. Der Wert ist bei allen Testläufen null. Die Levelgröße scheint dabei keinen Einfluss auf die Erreichbarkeit der Objectives darzustellen.

0	x	1	2	3	4
y	0	2	7	12	17
1	2	1	1	1	0
2	7	1	0	1	0
3	12	1	1	1	0
4	17	2	0	2	1

Abbildung .6.12: Heatmap zur Verteilung der Choke Points in der Versuchsreihe 4x4. In grau ist der Index der Module in der Heatmap markiert. Die grünen Achsen beziehen sich auf den Mittelpunkt der Platzierung eines Moduls in der Unity Engine. Die blauen Flächen zeigen die Module und die Anzahl der Platzierungen in den 30 Durchläufen.

0	x	1	2	3	4	5	6	7	8	9	10
y	0	2	7	12	17	22	27	32	37	42	47
1	2	0	0	2	0	1	0	2	0	1	0
2	7	1	2	0	0	1	1	0	1	1	3
3	12	1	0	2	2	2	0	2	1	0	1
4	17	0	0	1	0	2	0	0	1	2	0
5	22	1	0	1	0	0	1	0	0	0	0
6	27	1	1	0	1	0	0	0	0	0	0
7	32	0	0	1	0	1	0	1	1	0	0
8	37	0	0	1	0	1	1	1	2	0	1
9	42	1	0	0	0	0	1	1	0	1	1
10	47	1	0	1	0	1	0	2	2	2	1

Abbildung .6.13: Heatmap zur Verteilung der Choke Points in der Versuchsreihe 10x10. In grau ist der Index der Module in der Heatmap markiert. Die grünen Achsen beziehen sich auf den Mittelpunkt der Platzierung eines Moduls in der Unity Engine. Die blauen Flächen zeigen die Module und die Anzahl der Platzierungen in den 30 Durchläufen.

0	x	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
y	0	2	7	12	17	22	27	32	37	42	47	52	57	62	67	72	77	82	87	92	97
1	2	0	2	1	0	1	2	0	0	1	1	2	1	1	2	0	1	1	0	2	1
2	7	0	0	0	2	1	0	1	0	1	0	0	1	2	2	0	1	0	0	0	0
3	12	3	4	2	0	1	1	1	0	0	1	1	1	2	1	0	0	0	0	0	1
4	17	1	0	0	0	1	0	0	1	1	0	1	1	0	1	0	0	0	2	0	0
5	22	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	1	0
6	27	1	0	1	0	1	2	0	1	2	3	1	0	1	0	1	0	1	0	1	0
7	32	0	0	1	0	0	1	1	0	1	1	0	2	0	0	1	0	0	0	0	1
8	37	0	0	0	0	0	1	0	0	0	1	0	0	1	0	0	0	1	0	1	0
9	42	0	0	1	3	0	2	1	1	1	0	0	2	0	0	0	1	0	2	1	3
10	47	0	0	0	0	0	0	1	2	0	0	0	2	1	0	0	0	0	0	0	1
11	52	2	0	1	0	2	3	1	0	1	1	0	0	0	0	0	0	2	2	1	0
12	57	1	0	0	0	1	1	0	1	1	1	0	0	0	2	0	0	3	1	2	0
13	62	1	0	1	1	1	0	0	0	1	2	0	0	0	1	2	0	4	0	0	0
14	67	0	3	0	0	0	1	0	0	0	0	0	1	1	2	0	0	0	2	1	0
15	72	2	1	1	0	0	1	0	1	1	0	0	0	0	2	1	0	0	0	0	0
16	77	0	1	1	1	0	1	1	0	0	0	1	0	1	2	1	1	1	1	2	0
17	82	1	0	0	0	1	2	1	0	2	2	0	2	1	0	1	0	0	0	1	0
18	87	1	1	1	0	0	1	1	1	0	1	2	0	0	0	0	1	1	0	0	0
19	92	0	0	2	0	1	1	0	1	1	0	1	0	0	0	1	2	0	1	0	1
20	97	3	1	0	1	1	0	0	0	1	0	1	2	1	0	1	3	1	1	0	1

Abbildung .6.14: Heatmap zur Verteilung der Choke Points in der Versuchsreihe 20x20. In grau ist der Index der Module in der Heatmap markiert. Die grünen Achsen beziehen sich auf den Mittelpunkt der Platzierung eines Moduls in der Unity Engine. Die blauen Flächen zeigen die Module und die Anzahl der Platzierungen in den 30 Durchläufen.

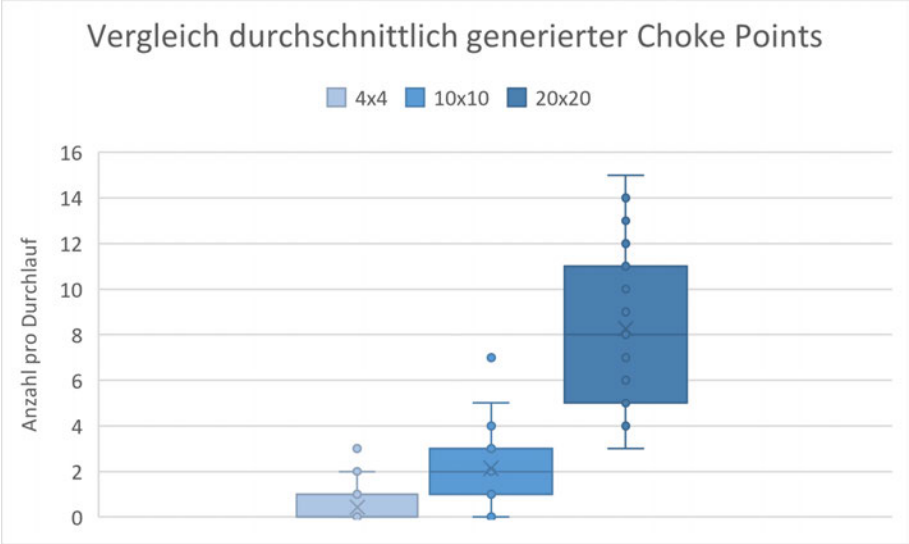

Abbildung .6.15: Vergleich der Anzahl durchschnittlich platzierter Choke Points in pro Durchlauf in einem Level. Hier lässt sich ein linearer Anstieg erkennen.

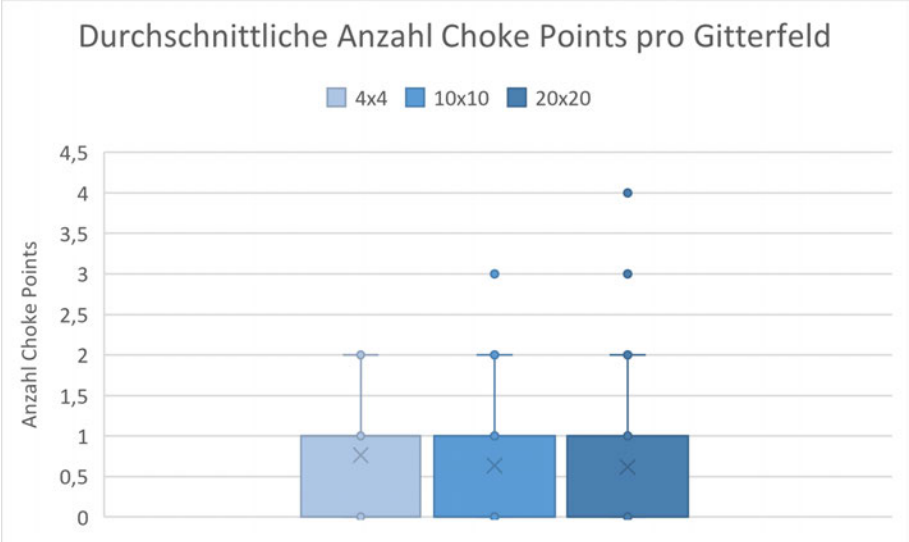

Abbildung .6.16: Vergleich der durchschnittlichen Anzahl insgesamt platzierter Choke Points pro Gitterfeld in den drei Versuchsreihen. Die Anzahl bleibt über alle Versuchsreihen nahezu konstant.

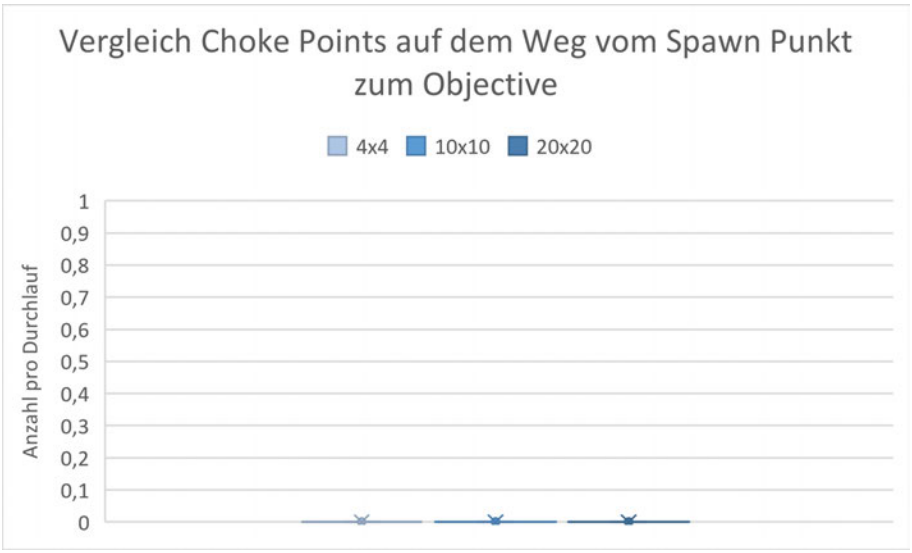

Abbildung .6.17: Vergleich der Anzahl durchschnittlicher Choke Points die während eines Durchlaufs auf dem Weg zwischen einem Spawn Punkt und einem Objectiv lagen. Die Anzahl betrug immer null.

Selbstständigkeitserklärung

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

Mittweida, den 29. November 2023

Johannes Bätz