Major Project

„Zur Implementation eines Pathfinding-Algorithmus für einen schwerelosen Voxel basierten 3D-Raum in Unity“
„Praxisprojekt am Beispiel eines Prototypen im Minecraft-typischen Würfel-Raster“

Modulnummer:	*BSc SAE6302*
Modulname:	*Major Project*
Abgabedatum:	*27.08.2021*
Abschluss:	*Bachelor of Science Games Programming*
Abschnitt:	*März 2021*
Name:	*Erik Thor Damisch*
Campus:	*Hamburg*
Land:	Deutschland
Wortanzahl:	*10100*

Selbstständigkeitserklärung:
Hiermit bestätige ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken (dazu zählen auch Internetquellen) entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht.

Hamburg, 23.08.2021 E. T. Damisch

Ort, Datum — Unterschrift Student/in

Rechtevereinbarung:
Der/die Student/in räumt dem SAE Institut das nicht exklusive jedoch zeitlich und örtlich unbeschränkte Recht ein, die vorliegende Arbeit zum Zweck der Ausbildung, sowie der Darstellung von Ausbildungsinhalten, zu speichern und für Personen des SAE Instituts zugänglichen zu machen.

Hamburg, 23.08.2021 E. T. Damisch

Ort, Datum — Unterschrift Student/in

Inhalt

Abbildungsverzeichnis

1 Einleitung

1.1 Thema

Die Idee hinter der Bachelorarbeit, ist es einen A*-Algorithmus für 3D-Pathfinding in einen Prototyp in Unity zu implementieren. Ein Non-Player-Charakter (NPC) soll mithilfe der Wegfindung vom Start zum Endpunkt navigieren können. Abgesehen von einigen Hindernissen soll die Bewegung in alle Richtungen möglich sein. Ein möglichst effizientes Ressourcenmanagement steht im Vordergrund.

1.2 Zielsetzung

Das Ziel des Praxisprojekts ist die erfolgreiche Umsetzung eines 3D-Pathfinding-Algorithmus in Unity. Der Prozess soll auf dem A*-Algorithmus basieren, welcher in Hinsicht auf Schnelligkeit und Performanz modifiziert werden soll. Das Ergebnis wird mithilfe eines in Unity erstellten Prototypen präsentiert und visualisiert.

Um den Erfolg des Prototyps zu gewährleisten, muss ein Pathfinding-Algorithmus für den 3-Dimensionalen-Raum implementiert und modifiziert werden, welcher immer einen nahezu perfekten Weg mit möglichst wenig Ressourcen findet. Demnach muss der zu navigierende Raum für den Algorithmus vereinfacht werden. Außerdem muss die Heuristik des Algorithmus auf Performanz angepasst werden. Am Ende soll ein NPC dem gefundenen Pfad folgen und so das Ziel erreichen. Der Prototyp soll ähnlich wie Minecraft nur aus quadratischen Blöcken bestehen, die Hindernisse sollen als feste Blöcke umgesetzt werden und alle anderen Blöcke, die keine Hindernisse sind, werden durch Gizmos in Unity visualisiert. Der Pfad wird ebenfalls durch Gizmos visualisiert, jedoch in einer anderen Farbe.

Um den Algorithmus für diese Zwecke zu modifizieren, werden verschiedene Techniken benötigt. Zum einen ist es notwendig den zu navigierenden Raum zu vereinfachen. Hierfür wird ein Voxelgrid verwendet, der Raum wird also in viele quadratische Blöcke aufgeteilt, jeder Block ist entweder navigierbar oder ein Hindernis. Bei größeren Blöcken ist der Algorithmus schneller und bei kleineren ist der Pfad dafür genauer und vermutlich kürzer. Um diese Eigenschaft zu nutzen wird ein Octree-Model verwendet. Hier kann jeder Block in acht gleichgroße Blöcke aufgeteilt werden, welches

beliebig oft angewendet werden kann, um einen genaueren Pfad zu finden. Sobald jedoch der zu betrachtende Block ein Hindernis ist, wird er nicht weiter für den Weg berücksichtigt.
Zum anderen muss die Heuristik des A*-Algorithmus modifiziert werden um den Fokus des Algorithmus auf eine schnelle anstatt eine perfekte Wegfindung zu legen.

Der entwickelte Algorithmus soll am Ende für diverse 3D-Pathfinding Probleme und auch in Videospielen mit mehreren Agenten anwendbar sein. Der Entwicklungsprozess soll in dem Prototyp festgehalten werden, demnach sollen verschiedene Stadien des Algorithmus verfügbar sein, um die Unterschiede zu testen. Außerdem soll es möglich sein in der Entwicklung einen Schritt zurück zu machen, da es essenziell sein wird eine gute Mischung aus Schnelligkeit und Genauigkeit zu erzielen.

1.3 Motivation

Künstliche Intelligenz ist ein fester Bestandteil von Videospielen, in nahezu jedem Spiel ist sie vorhanden. Ebenso wie Pathfinding, welches auch zum Bereich Künstliche Intelligenz gehört.

Innerhalb meines Studiums habe ich viel mit Pathfinding Algorithmen und Künstlicher Intelligenz gearbeitet, nicht nur weil es Teil des Studiums ist, sondern auch weil es mich sehr interessiert. In einem früheren Projekt „Ant Simulator“ habe ich mich um das 2D-Pathfinding gekümmert. Ich habe dort einen A*-Algorithmus implementiert, um so Wege zu den zufällig generierten Höhlen der Ameisen zu erstellen. Das hat mir so viel Spaß gemacht, dass ich meine Facharbeit über Pathfinding Algorithmen geschrieben habe. Durch mein Interesse an dem Thema und die Relevanz in der Industrie, bin ich sehr motiviert meine Bachelorarbeit über 3D-Pathfinding zu schreiben. Meine zuvor erlernten Fähigkeiten und mein Vorwissen lassen mich sehr zuversichtlich auf die erfolgreiche Umsetzung meines Projekts blicken.

Pathfinding in Videospielen ist wie schon zuvor erwähnt sehr verbreitet, es gibt kaum Spiele die ohne Pathfinding auskommen. Ob es nun ein Rennspiel ist wo der Non-Player-Character (NPC) eine bestimmte Strecke fahren muss oder ein Massively Multiplayer Online Role-Playing Game (MMORPG), wo sich die NPCs auf der Karte bewegen oder doch ein Role-Playing Game (RPG), indem man sich einen Zielpunkt aussuchen kann und dann automatisch dort hinläuft. Äußerst wichtig ist Pathfinding auch in Real-Time Strategy (RTS) Spielen.

3D-Pathfinding hingegen ist nicht so weit verbreitet, da es hierbei einige Schwierigkeiten gibt und es in vielen Spielen nicht nötig ist in alle Richtungen zu navigieren. In Six degrees of freedom (6DoF) Spielen ist 3D-Pathfinding jedoch nötig, hierfür gibt es eine Software. Diese Software heißt Mercuna AI-Navigation, sie ist ein Produkt der Firma Mercuna. Diese Firma ist sehr interessant, da sie Middleware für Navigation in Videospielen anbietet und es sehr nah an die Idee meines Praxisprojekts rankommt. Der Umfang meines Projekts ist wesentlich geringer, jedoch ist die Idee dahinter sehr ähnlich und die Software ist ein sehr gutes Referenz Produkt in der Industrie.

2 Kontext

2.1 Search Space Representation

Die Search Space Representation (SPR) ist eine Technik den zu navigierenden Raum darzustellen. Pathfinding Architekturen sind eine Abstraktion des Raums, in dem sich die Charaktere des Spiels bewegen können. Eine Abstraktion des Raumes ist nötig, da die zu Grunde liegende Physik zur Simulation des Raumes nicht direkt für das Pathfinding genutzt werden kann (vgl. Rabin 2019: 13–17). Mit der Wahl der SPR versucht man demnach eine Repräsentation zu wählen, die möglichst genau den wirklichen Raum darstellen kann (ebd. 13-17). Es gibt mehrere Möglichkeiten den Raum darzustellen, hierzu werden 3 verschiedene Techniken im folgenden Abschnitt betrachtet.

2.1.1 Waypoint Graphs

Mit der Hilfe von Waypoint Graphs bekommt man eine sehr vereinfachte Darstellung des zu navigierenden Raums. Es werden überall im Raum Waypoints angelegt, entweder manuell oder sie werden automatisch generiert. Waypoints sind navigierbare Punkte in der Welt, ein Charakter kann sich von Waypoint zu Waypoint bewegen. Es gibt also eine zuvor festgelegte Anzahl an navigierbaren Punkten. Die meisten Waypoints müssen vor der Laufzeit gespeichert werden. Waypoint Graphs sind gut, um einfache Welten darzustellen und diese zu navigieren (vgl. Rabin 2019: 16). Durch die überschaubare Menge an Waypoints kann der Algorithmus schnell vom Start-Punkt zum Ziel-Punkt finden. Auf der anderen Seite kann der Pfad durch die geringe Menge an navigierbaren Punkten beschränkt und unnatürlich wirken (ebd. 16). Waypoint Graphs sind schnell und einfach zu implementieren, verbrauchen vergleichsweise wenig Speicherplatz und die Wegplanung ist schnell und günstig (ebd. 16). Die Qualität des Pfades kann jedoch auch darunter leiden, zu viele Waypoints wirken sich negativ auf den Speicher und die Komplexität der Wegplanung aus, zu wenig Waypoints führen zu einer schlechten Qualität des Pfades (ebd. 16).

2.1.2 Sparse Voxel Octree

Ein Sparse Voxel Octree ist eine räumliche Struktur, die beim Rendern von Grafik genutzt wird, hauptsächlich im Bereich Belichtung und Strahlverfolgung. SVOs sind optimiert, um große Räume mit wenig Inhalt zu repräsentieren (vgl. Schwarz/Seidel 2010). Die Struktur ist grundlegend aufgebaut wie ein herkömmlicher Octree. Octrees unterstützen eine schnelle Positionssuche da sie den Raum hierarchisch in acht Teile auf jeder Ebene des Octrees unterteilen. In einem Octree werden alle Teile des Baumes in acht Räume unterteilt, das wird solange weitergeführt bis man die unterste Ebene des Octrees erreicht hat. In einem SVO funktioniert es etwas anders, dieser unterteilt sich nur in untere Ebenen, wenn der Voxel tatsächlich etwas beinhaltet. Wenn ein Voxel etwas beinhaltet wird dieser Voxel wie bei einem herkömmlichen Octree in acht Voxel unterteilt. Das wird solange fortgeführt bis der Voxel mit Inhalt auf der untersten Ebene des Baumes erreicht wird. So müssen viel weniger Zellen als bei einem Octree oder einem Grid gespeichert werden. Ein SVO bietet statt einfachen Kind-Eltern Verbindungen, Verbindungen zu jedem Nachbarn der Zelle. Diese Eigenschaft führt dazu, dass die Struktur schneller durchlaufen werden kann (vgl. Schwarz/Seidel 2010). Außerdem kann man diese Informationen auch gut für Wegfindungszwecke nutzen, weshalb SVOs im Bereich der 3D Wegfindung immer beliebter werden (vgl. Rabin 2019: 273-278).

2.1.3 Navigation Meshes

Navigation Meshes sind eine akkurate Darstellung des zu navigierenden Raums. Sie bestehen aus konvexen Polygonen, diese Polygone werden so in der Welt verteilt, dass jede begehbare Stelle von einem Polygon eingeschlossen wird (vgl. Rabin 2019: 16-17). Navigation Meshes legen sich also über den kompletten zu navigierenden Raum. Durch die Genauigkeit der Repräsentation wirken die gefundenen Wege natürlich (ebd. 17). Die Pfad Findung auf Navigation Meshes ist relativ schnell und trotzdem qualitativ hochwertig, da sich die Charaktere in alle Richtungen bewegen können. Außerdem sind Navigation Meshes nicht so Speicher aufwendig wie etwa Grids. Mit Navigation Meshes kann man selbst nicht Grid basierte Welten darstellen (ebd. 16). Ein Navigation Mesh zu implementieren ist jedoch höchst aufwendig und schwierig. Man kann sie nach der Implementierung auch nicht so leicht modifizieren, da es sehr komplex ist. Navigation Meshes sind für den 3D

Raum ungeeignet. Sie können zwar für kleine 3D Räume benutzt werden indem man mehrere Navigation Meshes übereinanderlegt (ebd. 17), es führt jedoch schnell dazu, dass man zu viel Speicher verbraucht und die Zeit, um einen Weg zu finden, viel zu hoch ist (ebd. 17).

2.1.4 Grids

Mit einem Grid wird die Welt als Array mit begehbaren und unbegehbaren Zellen dargestellt, sie können für 2D und für 3D Welten benutzt werden (vgl. Algfoor et al. 2015: Abs. 2). Grids sind eine sehr vereinfachte Repräsentation der Welt und sind einfach zu implementieren. Die Modifizierung von Grid basierten Repräsentationen ist einfach, man kann sie sogar in einem Text Editor bearbeiten (vgl. Rabin 2019: 15). Die Terrain-Kosten und ob eine Zelle begehbar ist oder nicht kann ebenso schnell angepasst werden. Außerdem ist es einfach eine Grid Zelle zu lokalisieren, man muss lediglich die Koordinaten durch die Grid Auflösung teilen, das Ergebnis ist dann die Zelle im Array wo sich der Charakter befindet (vgl. Algfoor et al. 2015). Grids sind für große Welten nicht so gut geeignet, da jede Zelle einzeln dargestellt wird. Auch wenn in 100x100 Zellen kein Objekt ist wird jede Zelle einzeln dargestellt und gespeichert (ebd. Abs. 2). Dem kann jedoch durch eine Sparse Representation entgegengewirkt werden (ebd. Abs. 2). Die Pfadplanung im Grid kann sehr teuer werden durch die feine Darstellung der Welt, weswegen oft eine weitere Abstrahierung des Grids benötigt wird (ebd. Abs. 2). Eine Glättung des gefundenen Wegs ist nötig, wenn man eine realistische Bewegung haben möchte.

2.1.5 Hierarchical Pathfinding

Je größer die Maps werden desto ungeeigneter werden die 3 vorgestellten Techniken. Hier kommt Hierarchical Pathfinding (HPF) ins Spiel. Beim HPF geht es darum, die Welt in verschiedener Auflösung weiter zu unterteilen. Die Welt muss in mindestens 2 Ebenen aufgeteilt werden. Jede Ebene hat eine andere Auflösung und somit mehr oder weniger Zellen. Man kann sich HPF an dem Beispiel einer Wohnung gut vorstellen. Die erste Ebene mit der niedrigsten Auflösung beinhaltet die Räume, man hat wenig Zellen da jeder Raum in der Wohnung nur eine Zelle ist (vgl. Millington 2019: 257). Die zweite Ebene ist hochauflösend und zeigt jede mögliche Bewegung in den Räumen der

Wohnung (ebd. 257). Wenn man nun versucht von der Küche ins Badezimmer zu gelangen wird zuerst die Raum Ebene benutzt. Hier wird geguckt welche Räume Küche mit Badezimmer verbinden. Alle Räume, die keinen Weg von Küche zu Badezimmer beinhalten werden ab jetzt ignoriert. Als nächstes wird die hochauflösendere Ebene betrachtet (ebd. 257). Hier wird ein Schritt für Schritt Pfad durch die zuvor gefundenen Räume geplant. Diese Technik kann je nach Größe und Komplexität der Welt mit beliebig vielen Ebenen erweitert werden. Außerdem kann jede Ebene auf einer beliebigen SPR basieren. Die Raum Ebene könnte auf einem Waypoint Graph beruhen und die Schritt-für-Schritt Ebene auf einem Grid oder einer Navigation Mesh (ebd. 257). Mit dem HPF kann man auch in großen Welten Ressourcen schonend navigieren.

2.2 A*-Algorithmus

Der A*-Algorithmus ist ein Pathfinding-Algorithmus, welcher auf dem Dijkstra-Algorithmus basiert (vgl. Millington 2019: 215–237). Pathfinding Probleme in Videospielen werden meistens mit dem A*-Algorithmus oder einer Variation des Algorithmus gelöst. Man kann den Algorithmus leicht Implementieren, er ist sehr effizient und man kann ihn gut optimieren (ebd. 215-237). A* im Gegensatz zu Dijkstra benutzt eine Heuristik, um zu bestimmen welche Zelle am vielversprechendsten ist (ebd. 215-237). Die Inbezugnahme der Heuristik ist ein Grund für die Popularität des A*-Algorithmus (ebd. 215-237).

Die Kosten Formel für den A*-Algorithmus sieht wie folgt aus:

$$f(x) = g(x) + (h(x) * Gewichtung)$$

g(x) sind die Kosten, die man bis zu der Zelle x benötigt hat (vgl. Rabin 2019: 7).
h(x) sind die Kosten, die man voraussichtlich benötigt, um das Ziel zu erreichen, sie ist die Heuristik Funktion und deshalb nur eine Schätzung der benötigten Kosten (ebd. 7).
Gewichtung ist ein Faktor, den man beliebig anpassen kann, um die Heuristik Funktion zu gewichten (ebd. 7). So kann man die Heuristik Funktion über- oder unterschätzen lassen.
f(x) stellt die voraussichtlichen Gesamtkosten dar, die benötigt werden, wenn diese Node für den Pfad benutzt wird (ebd. 7).

2.2.1 Heuristik

Die Heuristik Funktion im A*-Algorithmus wird benutzt, um zu schätzen wie weit die derzeitig betrachtete Zelle von der Zielzelle entfernt ist. So können Zellen bevorzugt werden, welche sich näher am Ziel befinden. Die Genauigkeit der Heuristik wirkt sich direkt auf die Schnelligkeit des Algorithmus aus (vgl. Millington 2019: 215–237). Wenn man eine perfekte Heuristik hätte, würde immer sofort der kürzeste Weg gefunden werden, jedoch würde dies auch das Pathfinding-Problem lösen (vgl. Millington 2019: 215–237). Es ist also praktisch unmöglich eine perfekte Heuristik Funktion für jede Möglichkeit zu haben. Es wird versucht so nah wie möglich an eine perfekte Heuristik Funktion heran zu kommen.

2.2.1.1 Unterschätzen der Heuristik

Wenn die Heuristik Funktion die tatsächliche Entfernung unterschätzt, wird der A*-Algorithmus langsamer. Durch die Unterschätzung wird der Algorithmus voreingenommen und wählt eher die Zellen aus, welche näher an der Startzelle sind (vgl. Millington 2019: 230-231). Somit werden mehr Zellen untersucht und es dauert länger einen Weg zu finden. Wenn die Heuristik Funktion immer unterschätzt wird, wird ein perfekter Weg, also der kürzeste mögliche Weg, gefunden welcher identisch mit dem Weg ist den Dijkstra finden würde (ebd. 230-231). Der Dijkstra Algorithmus ist fast identisch mit dem A*-Algorithmus, wenn dieser eine Null-Heuristik benutzt. Sobald die Funktion einmal überschätzt ist, ist die Garantie des perfekten Wegs erloschen (ebd. 230-231). Wenn es wichtiger ist den perfekten Weg zu finden als schnell einen guten Weg zu finden sollte man in jedem Fall unterschätzen.

2.2.1.2 Überschätzen der Heuristik

Das Überschätzen der Entfernung kann zu einem suboptimalen Pfad führen. A* wird sich eher nach der Heuristik richten und so eher Zellen auswählen welche näher am Ziel sind, statt Zellen zu wählen die weniger kosten (vgl. Rabin 2019: 7). So wird A* eher einen Weg finden mit wenigen Zellen, auch wenn die Kosten zwischen den Zellen hoch sind (ebd. 7). Die Suche wird manipuliert, um schneller die Zielzelle zu finden. Mit dieser Technik ist der A*-Algorithmus wesentlich schneller und er findet trotzdem einen akzeptablen Weg. Wenn die am meisten überschätzte Zelle mit Wert x überschätzt

wird, dann wird der gefundene Pfad auch nur maximal um x größer sein als der kürzeste Weg (vgl. Millington 2019: 230). Eine leicht überschätzende Heuristik Funktion kann sehr nützlich sein, da sie schneller zu einem Ergebnis führt und oft sehr gute Wege findet. Überschätzende Heuristiken haben jedoch wenig Spielraum für Fehler. Große Überschätzungen können schnell dazu führen, dass der Algorithmus einen schlechteren Weg findet (ebd. 230).

2.2.1.3 Euclidean Distance

Euclidean Distance (ED) ist eine Heuristik, um die genaue Entfernung zwischen zwei Punkten festzustellen. Diese Heuristik wird beim Pathfinding meistens benutzt, wenn die Bewegung in alle Richtungen möglich ist. ED gibt also die genaue Entfernung zwischen zwei Punkten wieder, Hindernisse und Wege werden hier nicht berücksichtigt, es geht lediglich um die Luftlinie. Das heißt ED kann nie den tatsächlichen Weg überschätzen und ist deswegen eine gute Heuristik, um immer den kürzesten Weg zu finden (vgl. Millington 2019: 232). ED ist außerdem sehr gut, wenn es wenig Wände und Hindernisse gibt, da die Heuristik dann genauer ist und bessere Ergebnisse liefert (ebd. 232). Allerdings ist ED schlecht für Welten mit vielen Wänden und Hindernissen geeignet. Der Weg wird in diesem Fall zu häufig sehr stark unterschätzt und somit werden unnötig viele Zellen untersucht und der Algorithmus braucht wesentlich länger (ebd. 232).

2.2.1.4 Octile Distance

Octile Distance (OD) auch bekannt als Diagonal Distance ist eine Anpassung der Manhattan Heuristik, welche die Bewegung in acht Richtungen auf einem Grid erlaubt (vgl. Amit 2010). Mit OD kann man sich in die vier Standard Richtungen und auch diagonal bewegen. OD gibt ebenfalls den kürzesten möglichen Weg ohne Berücksichtigung der Hindernisse an. OD arbeitet mit einem Skalierungswert D, dieser kann eine beliebige Zahl annehmen. Wenn D der Wert für die geringsten Kosten der Bewegung zwischen zwei Zellen ist, unterschätzt OD den tatsächlichen Weg und garantiert so einen optimalen Weg als Ergebnis (vgl. Amit 2010). Man kann D manipulieren, um den tatsächlichen Weg zu überschätzen und schneller einen Weg zu finden. Diese Verbesserung geht auf Kosten des optimalen Pfads, welcher nun nicht mehr garantiert werden kann (vgl. Amit 2010).

2.2.2 Open List Sorting

Eine Standard Implementierung von dem A*-Algorithmus würde immer die Zelle mit den niedrigsten f-Kosten auswählen. Es kommt häufig vor, dass es mehrere Zellen mit den gleichen f-Kosten gibt. Deshalb sollte man auch alle Zellen mit den gleichen f-Kosten sortieren. Hier bietet es sich an die Zellen zu bevorzugen welche die größten g-Kosten haben (vgl. Rabin 2019: 8-9). Bei den Zellen mit den größten g-Kosten wird erwartet, dass sie sich näher am Ziel befinden.

2.3 Unity 3D Engine

Die folgenden Informationen sind alle der offiziellen Unity Seite entnommen wurden (vgl. Unity o. D.). Unity 3D Engine ist eine Echtzeit 3D-Entwicklungsplattform, die überwiegend benutzt wird, um Videospiele zu entwickeln. Die Unity Engine bietet dem Nutzer kostenlos alle Werkzeuge, um ein Videospiel zu entwickeln, jedoch gibt es auch kostenpflichtige Varianten der Unity Engine. Diese kostenpflichtige Version bietet weitaus mehr Features und erleichtert die Entwicklung, außerdem darf man mit der kostenlosen Version maximal 100.000 Euro Umsatz in den letzten 12 Monaten gemacht haben.

Die Grafik in der Unity 3D Engine basiert auf OpenGL, Direct3D oder Vulkan je nachdem für welche Plattform entwickelt wird. Die Engine bietet außerdem die Möglichkeit der Animation von Charakteren und Partikelsystemen, sie ermöglicht die Implementierung von Musik und Geräuschen und sie erleichtert die Entwicklung von Multiplayer Aspekten. Die in Unity verfügbaren Mechanismen können durch selbst geschriebene Programme (Skripts) erweitert und verändert werden, die Logik kann ebenfalls durch diese Skripts erstellt oder verändert werden. Programmieren in Unity basiert auf Mono und bietet hauptsächlich C# als Programmiersprache. Neben kostenlosen Tutorials und Lernvideos verfügt Unity auch über einen sogenannten Asset Store. Im Asset Store gibt es diverse Charaktere, Skripte, Animationen, Geräusche und Musik und alles was man sonst noch zur Videospielentwicklung benötigt.

2.4 Mercuna

Alle Informationen über Mercuna wurden der offiziellen Seite von Mercuna entnommen (vgl. 3D Pathfinding | AI Middleware for UE4 and Unity 2019). „Mercuna: 3D Navigation“ ist eine kostenpflichtige Lizenz basierte 3D Pathfinding Lösung für Unity und Unreal Engine 4. Die 3D Navigation von Mercuna ist ein Programm für Entwickler, welches sich um das komplette 3D Pathfinding in einem Spiel kümmert. Es kümmert sich um Pathfinding, steering behaviour, Bewegung und dynamic obstacle avoidance. Diese Erweiterung für Projekte arbeitet mit einem optimierten A*-Algorithmus und als SPR wird ein Sparse Voxel Octree benutzt. Dieser Octree wird automatisch von Mercuna für die eigene Spiel Geometrie erstellt. Mercuna bietet diverse Features, um sicherzustellen, dass das Pfadfindung eine gute Performanz zeigt, gut aussieht und schnell ist. Es werden unter anderem die folgenden Features geboten: Asynchrones Pathfinding und Hierarchical Pathfinding. Hierarchical Pathfinding ist, wie schon zuvor erwähnt, eine gute Methode, um die Spielgeometrie zu vereinfachen und schnell einen Pfad zu finden. Asynchrones Pathfinding wird benutzt, um die Performanz des Spiels zu gewährleisten, auch wenn mehrere komplexe Wege gleichzeitig zu finden sind. Mercuna ist eine gute Lösung für Entwickler und Entwicklerstudios, die keine Zeit oder Mittel haben, um selber ein 3D Navigationssystem zu entwickeln. Funcom, Tripwire Interactive und Deep Silver FISHLABS sind Entwicklerstudios, die Mercuna benutzen. Tripwire Interactive benutzt Mercuna in dem Spiel Maneater für die Navigation der Menschen und der Fische. Deep Silver FISHLABS entwickelt momentan ihr neues Spiel Chorus, es handelt sich um einen space-combat shooter welcher ebenfalls mit der Mercuna 3D Navigation arbeitet. Jedoch funktioniert Mercuna momentan nur mit Unity und Unreal Engine, weshalb einige Entwickler und Studios nicht die Möglichkeit haben Mercuna zu nutzen.

2.5 Warframe

Warframe ist ein kostenloser online Co-op Shooter in der Third-Person-Perspektive. Die eigenen Spielfiguren sind Weltraum Ninjas und die Level sind prozedural generiert. Digital Extremes ist mit Warframe Vorreiter in der 3D Navigation in der Spieleentwicklung. Schon 2014 haben sie 3D Pathfinding in Warframe eingebaut (vgl. Brewer 2015). Diverse fliegende AI gesteuerte Gegner Einheiten benutzen 3D Pathfinding in Warframe wie zum Beispiel fliegende Raumschiffe (vgl. Brewer

2015). Diese Raumschiffe müssen durch enge Weltraumtrümmerfelder und Asteroidengürtel navigieren (vgl. Brewer 2015).

Auch in Warframe wird ein Sparse Voxel Octree als Search Space Representation genutzt (vgl. Brewer 2015). Das Team um Lead Programmer Dan Brewer arbeitet mit einem Greedy A*-Algorithmus und einer Größenkompensation für die Nodes, sodass größere Nodes bevorzugt werden (vgl. Brewer 2015). So wird sichergestellt, dass ein guter Weg schnell gefunden wird.

2.6 Aktuelle Trends

3D-Pathfinding wird in den letzten Jahren immer mehr erforscht und genutzt. Es wird auch außerhalb der Videospiel-Industrie genutzt. 3D-Pathplanning erfordert große Mengen an Rechenressourcen, weshalb es erst seit kurzem technisch möglich ist 3D-Pathfinding in Videospielen und anderen Bereichen anzuwenden (vgl. Brewer 2015).

Unmanned Aerial Vehicles (UAVs) können mit 3D-Pfadplanung autonom navigieren. UAVs können für das Militär, Wissenschaftliche Forschung, Tracking-Operationen und Robotik genutzt werden.

In der Videospiel-Industrie wird im Bereich des 3D-Pathfinding mehr und mehr geforscht, optimiert und implementiert. Es gibt viele Personen, Forschungseinrichtungen und Firmen, die sich mit dem Problem der Dreidimensionalen-Weg-Findung auseinandersetzen.

Lei Niu und Guobin Zhuo von der Wuhan University in China haben 2008 ein Paper über mögliche Verbesserungen für den A*-Algorithmus in Hinblick auf 3D-Pathfinding geschrieben. Sie hatten die Idee den zu navigierenden Raum in Regionen aufzuteilen, um so die Speicherkapazität zu schonen (vgl. Niu/Zhou 2008). Außerdem wird in jeder Region parallel der optimale Weg berechnet. Mit diesen Verbesserungen kann man die benötigte Zeit signifikant verkürzen.

Die Universiti Teknologi Malaysia hat 2015 eine Studie über Pathfinding Techniken veröffentlicht („A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games“). In dieser Studie geht es hauptsächlich um verschiedene Grids und hierarchische Techniken, die bis dahin entwickelt wurden. Die Studie fasst gut den Fortschritt in dem Bereich zusammen und zeigt wie viel Arbeit in 3D-Pathfinding gesteckt wird (vgl. Algfoor et al. 2015: Abs. 3). Außerdem wird in der Studie

gesagt, dass in den nächsten Jahren mehr Menschen forschen werden, um die Probleme die 3D-Pathfinding mit sich bringt, zu lösen.

An der Lublin University of Technology haben Jakub Smołka, Kamil Miszta, Maria Skublewska-Paszkowska, und Edyta Łukasik 2019 ein Paper über die Modifizierung des A*-Algorithmus für eine 3D-Engine veröffentlicht. In vielen Fällen ist die Rückgabe eines perfekten Wegs wichtiger als die Performanz des Algorithmus, in Videospielen ist es jedoch nicht immer so. Das Forscher Team hat den A*MOD entwickelt, sie verwenden statt der Standard Heuristik die chebyshev distance, außerdem haben sie vertex penalties implementiert, des Weiteren arbeitet der A*MOD mit post-processing des Pfads (vgl. Smołka et al. 2019).

2018 haben Daniel Brewer und Nathan R. Sturtevant das Paper „Benchmarks for Pathfinding in 3D Voxel Space“ veröffentlicht. Das Paper beschreibt wie das Pathfinding im 3D-Raum in Warframe umgesetzt wurde. Zu 3D-Pathfinding Problemen in Videospielen gibt es wenig Informationen, weshalb die Autoren ihre Arbeit und zugrunde liegende Daten veröffentlicht haben. Daniel Brewer ist der Lead Programmer von Warframe bzw. Digital Extremes.

Warframe ist ein Free-to-play Action-Rollenspiel, die Perspektive des Spielers ist Third-Person und es ist ein Online Multiplayer Shooter. Warframe ist eines der wenigen Spiele mit komplexem 3D-Pathfinding. Seit 2014 hat Warframe mit dem Arch Wing Update eine 3D Navigation für Künstliche Intelligenzen (vgl. Brewer/Sturtevant 2018). Daniel Brewer und das Team um ihn herum hat für das Pathfinding in Warframe einen Sparse Voxel Octree benutzt. Warframe ist immer noch sehr beliebt und wird weiterhin geupdatet. Die Entwickler von Warframe sind weiterhin dabei das Pathfinding zu verbessern, neue Techniken zu evaluieren und zu implementieren.

Mercuna 3D ist, wie schon zuvor erwähnt eine Lizenz basierte Middleware, welche von Mercuna entwickelt und Ende 2016 veröffentlicht wurde. Mercuna 3D hilft Entwicklerstudios dabei 3D-Pathfinding für ihre Spiele zugänglich zu machen. Mercuna 3D wird von einigen Entwickler Studios verwendet, unter anderem Funcom, Tripwire Interactive und Deep Silver FISHLABS. Viele Spiele, die mit Mercuna entwickelt werden, sind noch in der Entwicklungsphase. Mercuna ist eine gute Lösung für Indie Firmen und auch für etablierte Publisher/Firmen, da ihr Monetarisierungsmodell sich dem Umsatz der Firma anpasst (vgl. 3D Pathfinding | Mercuna o. D.). Schon seit der Gründung 2016 haben über 10 Videospielentwickler sich entschieden ihr Spiel mit Mercuna 3D zu entwickeln (ebd.). Daran kann man gut erkennen, dass der Trend 3D-Pathfinding in Videospielen einzusetzen gerade erst angefangen hat.

3 Methodik

3.1 Vorgehensweise

Zur erfolgreichen Bearbeitung des Projekts wurde folgender Plan bestehend aus mehreren Meilensteinen erstellt.

1. Vorbereitung des Projekts
2. Search Space Representation und Hierarchical Pathfinding Strukturen implementieren
3. A* implementieren und evaluieren
4. A* optimieren und evaluieren
5. Weitere Optimierungsmöglichkeiten in Betracht ziehen
6. Erstellen des Prototyps

Jeder Meilenstein muss erfüllt sein, um den jeweils nächsten Schritt zu bearbeiten, so wird sichergestellt, dass das Projekt Schritt für Schritt bearbeitet wird. Und der Fokus immer nur auf einem Meilenstein liegt. Die Meilensteine 2, 3, 4 und 5 werden nach erfolgreichem Beenden evaluiert. Nach der Evaluation der Meilensteinergebnisse wird für jeden Meilenstein Feedback eingeholt. Nach den Feedbacks wird entschieden ob das Ergebnis erneut evaluiert werden muss. Wenn das Feedback positiv ist und keine Kritik angebracht wird, wird der nächste Meilenstein bearbeitet.

3.2 Vorbereitung

3.2.1 Versionskontrolle

Um Datenverlust vorzubeugen wird ein online Repository (GitHub) genutzt. In GitHub kann man ein Projekt erstellen und dieses dann fortlaufend updaten. So ist selbst bei defektem Computer oder Datenverlust aus diversen Gründen sichergestellt, das keine oder nur die Daten seit dem letzten Hochladen verloren gehen. Diese Uploads des aktuellen Projekts nennt man Commits. Github bietet auch die Möglichkeit alte Commits mit den neuen zu vergleichen oder den alten Stand der Commits wiederherzustellen. Außerdem gibt es die Möglichkeit verschiedene Zweige für die

verschiedenen Entwicklungsschritte zu erstellen. Wodurch das Projekt und die Commits übersichtlicher und leichter zuzuordnen sind.

3.2.2 Entwicklungsumgebung

Für das Projekt wurde die Entwicklungsumgebung Unity Engine gewählt. Unity ist eine für Studenten und Hobby Programmierer kostenlose Engine. Unity bietet alle Tools, um das Projekt erfolgreich zu erstellen. Unity Scripts sind eine gute Möglichkeit, um die Engine auf die Bedürfnisse des Projekts anzupassen.

3.3 Search Space Representation

Die Auswahl der SPR ist ein wichtiger Teil des Projekts, es muss sichergestellt werden, dass die SPR komplex genug ist, um 3D Pathfinding zu erlauben und gleichzeitig nicht zu komplex ist, um im Rahmen des Projekts implementiert zu werden. Außerdem muss die SPR ermöglichen sich frei im Raum zu bewegen, weshalb beispielsweise ein Waypoint Graph ungeeignet ist. Die Recherche hat ergeben, dass es zwei gute Möglichkeiten gibt den Raum darzustellen. Zum einen könnte man herkömmliche Grids und zum anderen „Sparse Voxel Octrees“ implementieren. Grids teilen den Raum in viele kleine gleichgroße Würfel auf, jeder Würfel kann begehbar oder unbegehbar sein, begehbare Nodes können jedoch einen Kosten-Modifizierer besitzen. Nodes zu durchqueren kann verschieden teuer sein, je nachdem welches Material durchquert werden muss, können verschiedene Kosten entstehen. Es wäre zum Beispiel teurer sich durch Wasser zu bewegen als auf Land. Das Problem bei Grid basiertem 3D-Pathfinding ist, dass ein großer Raum sehr viele Nodes benötigt, um dargestellt zu werden, was sich auch stark auf die Performanz auswirkt.

„Sparse Voxel Octree“ ist eine verbreitete Grafik Struktur welche hauptsächlich für Belichtung und Ray-Tracing genutzt wird. SVO’s erleichtern eine schnelle Positionssuche und unterteilen hierarchisch den Raum in acht Teile auf jedem Level des Octree. Die Datenstruktur beinhaltet Verbindungen zwischen Nachbarn, jede Node hat Informationen über all ihre Nachbarn, anstatt der herkömmlichen Eltern-Kind Verbindungen. Die Verbindungen, die im SVO gegeben sind, können sehr gut für 3D-Pathfinding verwendet werden.

Um diese beiden SPR's bezogen auf das Projekt zu Vergleichen wurden folgende Kriterien erstellt:

1. Geschwindigkeit
2. Genauigkeit
3. Ressourcennutzung

Eine SPR im Rahmen des Projekts ist ausschließlich vorhanden, um einen Algorithmus darauf laufen zu lassen. Deshalb kann die SPR nur mit einem Pathfinding Algorithmus evaluiert werden. Der ausgewählte Algorithmus wird dann auf beiden SPR's laufen und mit den drei Kriterien analysiert. Um beide SPR's zu vergleichen werden mehrere Start- und Endpunkte ausgewählt. Der Algorithmus soll dann durch beide SPR's laufen und es wird geguckt mit welcher SPR der Algorithmus schneller läuft. Genauso wird auch ermittelt welche SPR eine bessere Genauigkeit erzielt, wobei es durchaus möglich ist das der Algorithmus durch beide SPR's den gleichen Weg findet. Bei der Genauigkeit geht es darum, dass der Algorithmus den kürzesten oder einen möglich kurzen Weg findet. Je näher die Länge des Weges an der Luftliniendistanz zwischen Start und Ziel ist desto genauer ist er. Die Ressourcennutzung kann man in zwei Teile aufteilen. Der erste Teil ist ohne Bezug auf den Algorithmus, es geht lediglich darum wie viel Speicher die SPR benötigt, ohne dass ein Weg gesucht wird. Beim zweiten Teil geht es darum wie der Algorithmus mit dem SPR die Ressourcen nutzt. Es wird also wieder der gleiche Algorithmus auf beide SPR's mit dem gleichen Start- und Endpunkt angewendet und dann wird ermittelt wie viele Ressourcen verwendet wurden. So wird herausgefunden welche SPR mit dem genutzten Algorithmus Ressourcen schonender ist.

3.4 Algorithmus Evaluieren

3.4.1 Analyse

Der beliebteste und weitverbreitetste Algorithmus für Videospiele ist der A*-Algorithmus. Es gibt jedoch auch andere Algorithmen die man in Betracht ziehen könnte wie den Dijkstra-Algorithmus. Im Rahmen dieses Projekts werden nur die beiden genannten Algorithmen betrachtet, da sie in Videospielen am häufigsten verwendet werden. Um herauszufinden welcher Algorithmus am besten für das Projekt geeignet ist, müssen beide Algorithmen analysiert und verglichen werden. Ein Kriterium was der Pathfinding Algorithmus erfüllen muss, ist die Funktionalität auf Graphen. Wenn der Algorithmus nicht auf Graphen funktioniert kann er nicht für das Projekt verwendet

werden und muss somit nicht weiter evaluiert werden. In diesem Fall könnte der Algorithmus nicht auf eine SPR angewendet werden.

Ein weiteres Kriterium was für das Projekt unbedingt erfüllt werden muss, ist die Inbezugnahme von Heuristiken. Heuristiken sind äußerst wichtige Werkzeuge, um den Algorithmus zu verändern und an die Bedürfnisse des Projekts anzupassen. Die Heuristik ist nötig, um den Algorithmus zu gewichten. In dem Projekt ist es wichtig schnell und ressourcenschonend einen Weg zu finden. Also braucht man eine Heuristik Funktion, die genau darauf abzielt. Ohne eine Heuristik Funktion wäre ein Algorithmus also unbrauchbar für dieses Projekt.

Weitere Kriterien sind Geschwindigkeit und Genauigkeit. Diese beiden Kriterien sind abhängig voneinander und müssen deshalb gemeinsam betrachtet werden bzw. das Verhältnis beider Kriterien zueinander. Bei der Geschwindigkeit ist gemeint, wie lange der Algorithmus für ein Wegfindungsproblem braucht bzw. wie viele Knotenpunkte untersucht werden mussten, um das Ziel zu finden. Umso näher die Zahl der untersuchten Knotenpunkte an der Zahl der Knotenpunkte des gefundenen Wegs ist desto besser. Die Genauigkeit misst, wie nah der gefundene Weg an dem kürzesten Weg dran ist oder wie nah der gefundene Weg an der Distanz zwischen Start- und Endpunkt dran ist.
Außerdem kann man den Algorithmus in seiner Komplexität bewerten. Komplexität meint hier wie viele Möglichkeiten es gibt den Algorithmus zu verändern. Ein Algorithmus, der lediglich einen Weg findet und dies immer nach den gleichen Kriterien macht, ohne diese Kriterien beeinflussen zu können, ist minimal komplex.

3.4.2 Vergleich

Da die ersten beiden Kriterien entweder gegeben oder nicht gegeben sind und der Algorithmus, wenn die Kriterien nicht erfüllt sind unbrauchbar ist, werden diese Kriterien nicht zum Vergleich benötigt. Die letzten drei Kriterien Geschwindigkeit, Genauigkeit und Komplexität sind für den Vergleich notwendig. Geschwindigkeit und Genauigkeit müssen in Relation zu einander betrachtet werden. Ein Algorithmus schneidet im Vergleich besser ab, wenn Geschwindigkeit und Genauigkeit höher als bei dem anderen sind. Für dieses Projekt ist die Geschwindigkeit wichtiger als die Genauigkeit weshalb die Geschwindigkeit schwerer gewertet wird als die Genauigkeit. Das Ziel des Projekts ist es, möglichst schnell gute Wege in einem großen 3D-Raum zu finden, weshalb die Geschwindigkeit wichtiger ist als die Genauigkeit.

Die Komplexität eines Algorithmus ist nur dann wichtig, wenn die Geschwindigkeit und Genauigkeit zweier Algorithmen sehr ähnlich sind. Dann wird die Komplexität als Vergleichskriterium genutzt. Außerdem ist die Komplexität von Bedeutung, wenn der Algorithmus zu komplex ist, um ihn im Rahmen des Projekts/Zeitplans zu implementieren.

4 Durchführung

4.1 Versionskontrolle

Zur Versionskontrolle wird ein Git-Repository bei dem Anbieter GitHub eingerichtet, hierzu wird die Desktop App von GitHub heruntergeladen und anschließend ein Repository für das Projekt erstellt. Der nächste Schritt ist, regelmäßig den Fortschritt hochzuladen umso Datenverlust vorzubeugen. Es werden lediglich die Dateien hochgeladen, die sich seitdem letzten hochladen geändert haben, neu hinzugefügt oder gelöscht wurden.

4.2 Entwicklungsumgebung

Zur Entwicklung des Projekts wird die neuste Version von Unity gewählt, Version 2021.1.10f1. Unity besitzt alle erforderlichen Aspekte, um das Projekt umzusetzen. Als Code-Editor wird Visual Studio 2019 gewählt, Visual Studio arbeitet sehr gut mit Unity zusammen.

4.3 Erstellung der Search Space Representation

Der erste Schritt des Projekts ist es eine passende Repräsentation des Raumes zu finden und zu implementieren. Zunächst werden alle Repräsentationsmöglichkeiten betrachtet und die passenden für 3D-Räume ausgewählt. Ich habe mich für den gewöhnlichen Octree und den Sparse Voxel Octree entschieden.

4.3.1 Erstellung des Octree (Grid)

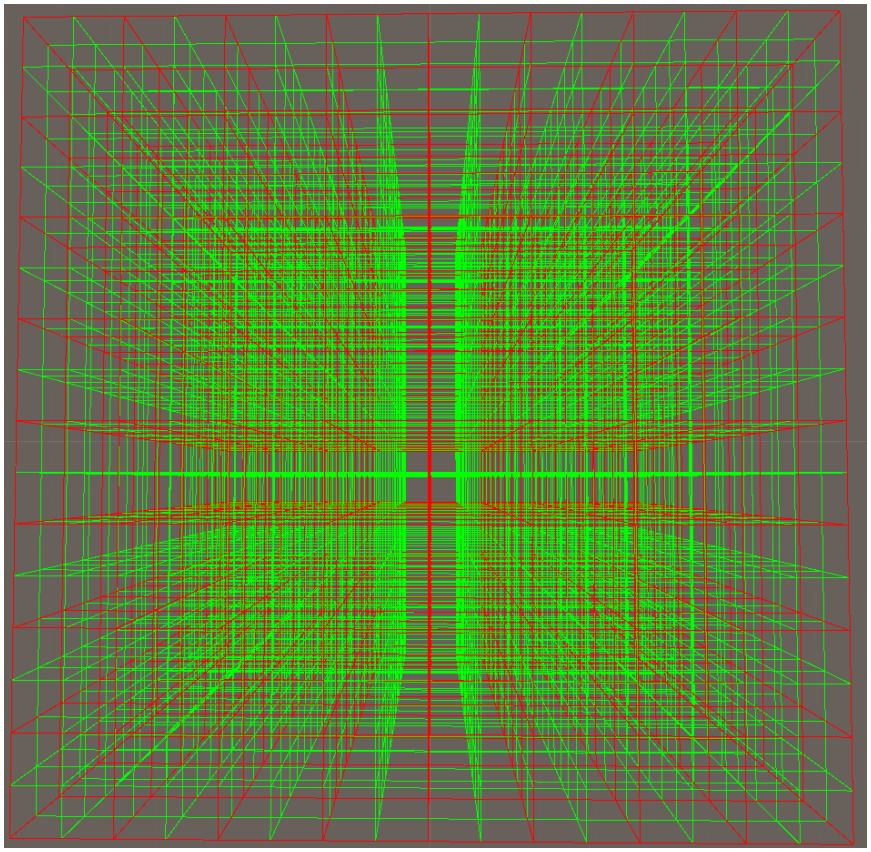

Abbildung 1 - Gewöhnlicher Octree

Wie in Abbildung 1 zu sehen ist, funktioniert ein gewöhnlicher Octree ähnlich wie ein Gitternetz. Der Octree teilt den gesamten Raum in gleichgroße Würfel auf. Die Größe der Würfel kann man so setzen wie man möchte. Das besondere an einem Octree ist, dass er mit einer zuvor festgelegten Anzahl an Schichten arbeitet. Jede Schicht wird nacheinander erstellt, es fängt an bei der Wurzel des Baums und geht dann Schritt für Schritt eine Schicht weiter nach oben. Die Erstellung des gewöhnlichen Octree ist nicht so komplex. Wir benötigen zwei Variablen, zum einen die Größe des Raums und zum anderen die Anzahl der Schichten. Zunächst teilen wir den gesamten Raum in acht gleichgroße Knotenpunkte auf. Diese acht Knotenpunkte sind jetzt die direkten Kinder des Wurzel-Knotenpunktes. Der nächste Schritt ist alle acht Kinder jeweils in acht Knotenpunkte zu unterteilen. Dieser Vorgang wird so oft wiederholt bis die Anzahl an zuvor festgelegten Schichten erreicht ist. Bei einer Anzahl von sechs Schichten wären es also insgesamt $(8+8^2+8^3+8^4+8^5+8^6) = 299.592$ Knotenpunkte. Der Grund warum sechs Schichten für das Beispiel ausgesucht wurden, ist dass der Octree für sechs Schichten noch in einigen Sekunden erstellt werden kann sobald man mehr als

sechs Schichten erstellen will dauert es äußerst lange oder Unity hängt sich auf. Mir ist es mehrmals passiert, dass Unity bei mehr als sechs Schichten abgestürzt ist. Bei acht Schichten verbraucht Unity den gesamten Arbeitsspeicher von 32 GB und kann selbst nach 10 Minuten kein Octree erstellen. Das verwundert aber auch nicht da es über 19 Millionen zu erstellende Nodes sind.

4.3.2 Optimierung des Octree (Sparse Voxel Octree)

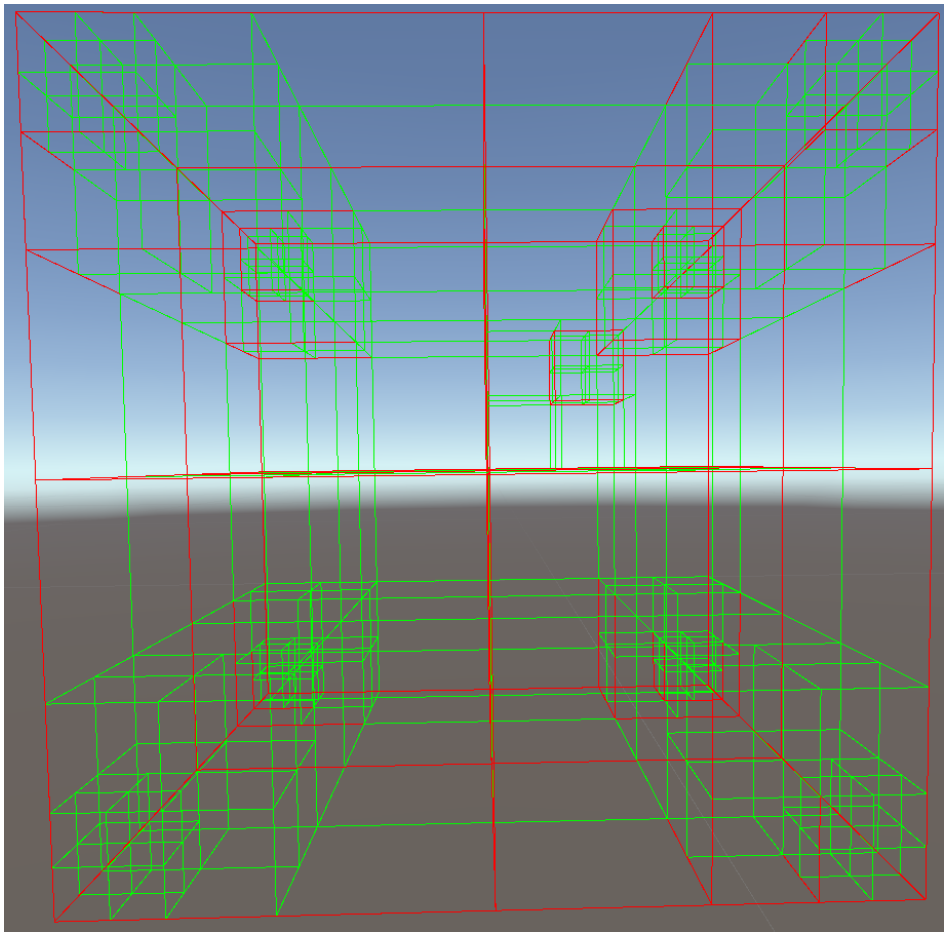

Abbildung 2 - Sparse Voxel Octree

Nach den ernüchternden Ergebnissen des gewöhnlichen Octree und der riesigen Menge an benötigten Knotenpunkten, habe ich die Erstellung eines Sparse Voxel Octree in Angriff genommen. Um einen SVO zu erstellen benötigt man zunächst dieselben zwei variablen Größen und Schichten. Der Wurzel-Knotenpunkt (Schicht 0) wird in der gewollten Größe erstellt und in acht gleichgroße Knotenpunkte unterteilt. Außerdem werden alle Hindernisse, die sich in dem Raum befinden in eine Liste eingefügt. Der nächste Schritt ist, zu Testen in welchen der acht Knotenpunkte sich

Hindernisse befinden. Jeder Knotenpunkt der ein oder mehrere Hindernisse beinhaltet wird wieder in acht Knotenpunkte unterteilt. Jeder Knotenpunkt, der keine Hindernisse beinhaltet, wird nicht weiter unterteilt und als kinderloser Knotenpunkt gekennzeichnet. Die neu erstellten Kinder-Knotenpunkte der Eltern mit Hindernissen, werden wiederum darauf getestet ob sie Hindernisse beinhalten und wenn ja werden sie wieder in acht aufgeteilt. Jedes Mal, wenn ein Knotenpunkt in acht unterteilt wird, werden diese acht Knotenpunkte als Unterpunkte in dem Eltern-Knotenpunkt gespeichert. Diesen Vorgang führen wir solange fort bis die höchste Schicht erreicht ist oder keine Knotenpunkte mehr Hindernisse beinhalten. Wo ein Hindernis existiert, wird immer bis auf die detailreichste Schicht runtergebrochen. Nach der Erstellung des Octree müssen wir den Knotenpunkten noch einige Informationen zufügen.

In diesem Projekt gibt es keine raumdiagonale Bewegung, weshalb es bis zu 18 verschiedene Bewegungsmöglichkeiten gibt. Abbildung 3 zeigt die möglichen Bewegungen des Algorithmus. Jede rote Spitze steht für eine Bewegungsrichtung, wie zu sehen ist, gibt es keine raumdiagonale Bewegung. Die Rückseite des Würfels in der Abbildung sieht genauso aus wie die Vorderseite, die Bewegungsmöglichkeiten sind gespiegelt. Es müssen also bis zu 18 Nachbarn in jedem Knotenpunkt gespeichert werden, um alle Bewegungen abzudecken. Bei den Nachbarn muss darauf geachtet werden, dass ein Knotenpunkt immer nur Nachbarn besitzt, die gleichgroß oder größer sind.

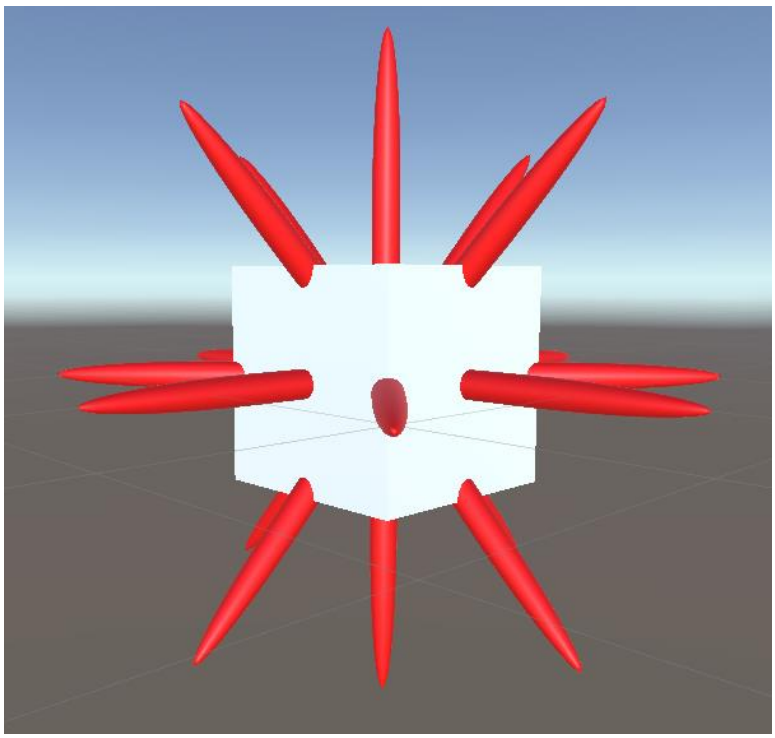

Abbildung 3 - Bewegungsmöglichkeiten

Jeder Knotenpunkt speichert die eigene Größe und Position im Raum. Außerdem speichern wir für jeden Knotenpunkt den Eltern-Knotenpunkt ab. Für die Knotenpunkte in der detailreichsten Schicht wird gespeichert, ob sie ein Hindernis beinhalten und somit blockiert sind. Die Knotenpunkte in der höchsten Schicht werden entweder als blockiert oder nicht blockiert gekennzeichnet. Ich habe mich dazu entschieden den SVO zu vereinfachen. Man könnte jeden Knotenpunkt in der höchsten Schicht so speichern, dass die Position jedes teils eines Hindernisses genau vorliegt und somit eine noch genauere Darstellung der Welt vorhanden wäre.

4.4 Implementierung der Algorithmen

Der Sparse Voxel Octree ist nun erstellt und als nächstes müssen die Algorithmen implementiert werden. Die Implementierung des A*-Algorithmus auf einen Octree unterscheidet sich leicht von einer Standard-Implementierung. Der SVO wurde so angepasst, dass der Algorithmus ohne Probleme implementiert werden kann. Da die beiden zu evaluierenden Algorithmen (A* und Dijkstra) sich lediglich in der Heuristik unterscheiden, wird erst der A* implementiert. Um dann den Dijkstra Algorithmus zu erhalten, muss man nur die Heuristik Funktion weglassen. Die Implementierung erfolgt anhand eines Pseudocodes für den Algorithmus, dieser wird dem Buch AI for Games Third Edition von Ian Millington entnommen.

Im Folgenden wird die komplette Implementierung des A*-Algorithmus Schritt für Schritt erklärt. Um einen Algorithmus zu implementieren wird zunächst ein Graph benötigt, der den Raum repräsentiert in diesem Fall ist das der Sparse Voxel Octree.

Jede Node muss die folgenden Werte speichern: Position; Größe; Schicht; ob die Node blockiert ist; F-Cost; G-Cost; H-Cost; einen Array mit allen Nachbarn; einen Array mit acht Child-Nodes, falls vorhanden; eine Parent-Node und die Verbindung zu einer anderen Node, um den gefundenen Pfad zurückzuverfolgen.

Zum Start der Funktion müssen außerdem einige Variablen deklariert werden. Zuerst werden Start-Node und End-Node deklariert, sie stehen für den Start und das Ziel des Algorithmus. Außerdem wird eine Open- und eine Closed-List deklariert und eine Current-Node, die immer die zu betrachtende Node ist. Im nächsten Schritt werden alle zuvor genannten Variablen initialisiert, also mit einem Wert versehen. Jetzt wo alle Variablen einen Wert haben kann der eigentliche Algorithmus implementiert werden.

Der Hauptteil des Algorithmus ist eine While-Schleife die solange läuft bis die Current-Node gleich der End-Node ist oder die Open List keine weiteren Nodes beinhaltet. Am Anfang der Schleife wird die Open List sortiert, sodass die Nodes mit der niedrigsten F-Cost vorne in der Liste sind. Dann wird die vorderste Node aus der Open List gleich der Current-Node gesetzt.

Als nächstes wird jeder Nachbar der Current-Node betrachtet, die Nachbarn werden geprüft ob sie Null oder blockiert sind. Wenn eins von beiden zutrifft, wird der nächste Nachbar betrachtet andernfalls geht die Betrachtung weiter. Da mit einem Octree gearbeitet wird, muss der nächste Test sein, ob der Nachbar Child-Nodes besitzt. Wenn ja, muss ermittelt werden welche der Child-Nodes an die Current-Nodes grenzen und diese Nodes müssen der Open List hinzugefügt werden. Dies wird wiederholt bis die detailreichste Schicht erreicht wird. Wenn es sich um eine diagonale Bewegung handelt, muss als nächstes überprüft werden, ob wenigstens eine der Nodes die die diagonale Bewegung eingrenzen, frei sind. Wenn keine der eingrenzenden Nodes frei ist, wird der nächste Nachbar betrachtet, da keine diagonale Bewegung möglich ist.

Wenn der Nachbar weiter betrachtet wird, hat er sich als möglicher Weg teilqualifiziert und ihm wird eine G-Cost und H-Cost hinzugefügt oder aktualisiert. Die H-Cost wird mit der Heuristik Funktion ermittelt und die G-Cost wird berechnet, indem die G-Cost der Current-Node mit dem Weg zwischen Current-Node und Nachbar addiert wird. Nachdem G-Cost und H-Cost ermittelt wurden, wird abgefragt ob der Nachbar in der Closed List vorhanden ist. Wenn der Nachbar in der Closed List vorhanden ist wird geprüft, ob die Instanz des Nachbarn in der Closed List eine kleinere G-Cost hat als die Instanz des Nachbarn die momentan betrachtet wird. Ist dies der Fall wird der nächste Nachbar betrachtet, wenn nicht wird der Nachbar von der Closed List gelöscht. Wenn die Closed List den Nachbar nicht beinhaltet, wird geprüft ob er in der Open List vorhanden ist. Wenn er vorhanden ist wird auch hier wie bei der Closed List geprüft ob die Instanz in der Open List eine niedrigere G-Cost hat. Ist dies der Fall, wird der nächste Nachbar betrachtet, wenn nicht geht es weiter zum nächsten Schritt.

Nun wird dem Nachbarn die aktualisierte G-Cost zugewiesen und es wird die Current-Node als Verbindung auf dem Nachbarn gespeichert. Zum Schluss wird erneut geprüft ob der Nachbar in der Open List vorhanden ist, wenn nicht wird er der Open List hinzugefügt ansonsten wird der nächste Nachbar betrachtet. Wenn alle Nachbarn der Current-Node betrachtet wurden, wird die Current-Node von der Open List entfernt und der Closed List hinzugefügt.

Sobald die While-Schleife vorbei ist, wird geprüft ob die Current-Node gleich der End-Node ist, wenn dies der Fall ist, geht es weiter, ansonsten wurde kein Weg gefunden und es wird Null zurückgegeben.

Wenn ein Pfad gefunden wurde wird eine Liste erstellt, die den Pfad beinhalten soll und es wird die Current-Node hinzugefügt, die Current-Node ist in dem Fall die End-Node.

Nun gehen wir in eine weitere While-Schleife, die solange läuft bis die Current-Node gleich der Start-Node ist, so können wir den Weg zurückverfolgen. In der While-Schleife passieren nur zwei Dinge, einmal wird dem Pfad die Verbindung der Current-Node hinzugefügt und dann wird die Current-Node zu der Verbindung der Current-Node. So wird immer wieder die Verbindung dem Pfad hinzugefügt und der Pfad wird zurückverfolgt. Nachdem die Schleife durchgelaufen ist wird der Pfad umgekehrt, damit der Pfad den Weg vom Start zum Ziel zeigt und nicht den Weg vom Ziel zum Start. Am Ende wird der Pfad zurückgegeben.

Um daraus einen Dijkstra-Algorithmus zu erstellen müssen nur die H-Cost und F-Cost entfernt werden, und die Open List muss nach G-Cost sortieren.

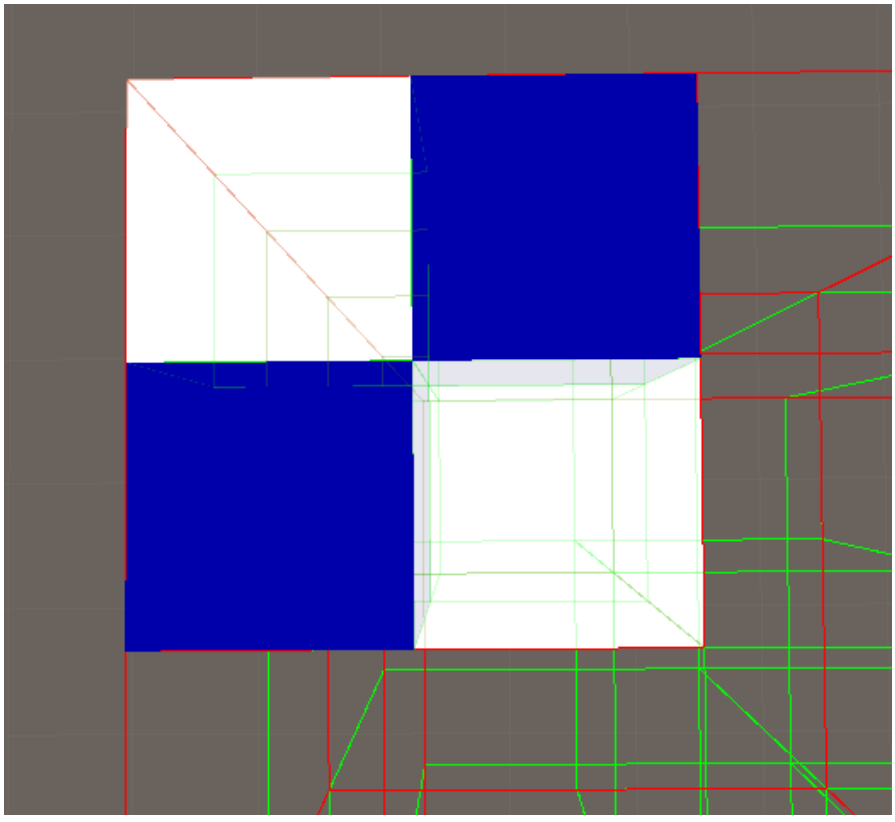

Abbildung 4 - Fehlschlag der Prüfung auf diagonale Bewegung

Die Implementierung hat sehr gut funktioniert bis auf den Teil, der auf diagonale Bewegung prüft, da ich nicht genau wusste wie ich ermittle welche Nodes die diagonale Bewegung eingrenzen. Zuerst habe ich gedacht, ich könnte prüfen welche Nachbarn die Current-Node und der zu betrachtende Nachbar gemeinsam haben, jedoch ist das keine Lösung. Das Problem ist, das nicht nur die eingrenzenden Nodes gleich sind, sondern auch andere Nodes die Nachbarn von beiden Nodes sein können. In Abbildung 4 ist zu sehen wie der Algorithmus einen diagonalen Weg gefunden hat, obwohl er versperrt ist. Dieses Problem ist durch die fehlerhafte Prüfung auf diagonale Bewegung entstanden.
Nach dem Fehlschlag habe ich mich für eine andere Herangehensweise entschieden. Zuerst prüfe ich in welche diagonale Richtung sich bewegt werden soll, dies wird anhand der Positionen der beiden Nodes ermittelt. Wenn sich zum Beispiel der x Wert und der Y Wert unterscheiden weiß ich, dass die Bewegung in XY Richtung ist. Wenn sich nur ein Wert unterscheidet kann man sicher sein, dass es keine diagonale Bewegung ist. Wenn wir bei unserem Beispiel mit XY-Richtung bleiben, wird als nächstes ermittelt welche Node in X und in Y Richtung angrenzt. Dann wird die gesamte angrenzende Strecke in Abständen der kleinsten Nodes entlang der Z-Achse überprüft. Sollten alle angrenzenden Nodes blockiert sein, ist keine Diagonale Bewegung möglich.

4.4.1 Evaluierung der Algorithmen

4.4.1.1 Funktionalität

Die Evaluierung der Algorithmen beginnt damit zu testen ob die Algorithmen mit einem Graphen funktionieren, in diesem Fall der optimierte Octree. Beide Algorithmen werden auf den Octree angewendet. Bis auf ein paar kleine Änderungen der Standard Implementierung lassen sich beide Algorithmen problemlos auf den optimierten Octree anwenden. Das erste Ausschlusskriterium ist damit erfüllt und beide Algorithmen können weiter evaluiert werden.

4.4.1.2 Heuristik

Das zweite Ausschlusskriterium ist die Inbezugnahme einer Heuristik. Der A*-Algorithmus benutzt eine Heuristik Funktion. Diese Funktion gibt die voraussichtliche Distanz einer Node zum Ziel an, die Funktion kann angepasst werden um schneller einen Weg oder um einen kürzeren Weg zu finden. Diese Funktion kann die Menge an besuchten Nodes deutlich verringern und ist deshalb gut geeignet für dieses Projekt.
Der Dijkstra-Algorithmus hingegen besitzt keine Heuristik Funktion und ist deshalb ungeeignet für das Projekt. Der Dijkstra Algorithmus hat den Vorteil, dass er immer den kürzesten Weg vom Start zum Ziel findet.

4.4.1.3 Geschwindigkeit und Genauigkeit

Geschwindigkeit und Genauigkeit werden getestet, indem der Algorithmus auf den optimierten Octree angewendet und die Zeit gestoppt wird. Obwohl der Dijkstra Algorithmus nicht für das Projekt geeignet ist, kann man ihn benutzen um die Genauigkeit des A* zu beurteilen, da Dijkstra immer den kürzesten Weg findet. Außerdem kann der Dijkstra Algorithmus nützlich sein, um die Geschwindigkeit zu vergleichen. Für den Test der Genauigkeit und Geschwindigkeit wurde ein 3D-Raum von Tausend mal Tausend mal Tausend Unity Einheiten gewählt, der Octree wurde mit acht Schichten erstellt. Es wird ein Weg diagonal durch den Raum gefunden, um einen möglichst langen Weg für den Vergleich zu haben. Außerdem werden in dem Raum 1000 Hindernisse zufällig generiert.
Der A*-Algorithmus ist schon ohne jegliche Optimierungen recht schnell, er benötigt ca. 5 Sekunden, um einen Weg durch den Test-Raum zu finden und ist durchschnittlich nur wenige Nodes von dem kürzesten Weg entfernt. Je nach Platzierung der Hindernisse kann das Ergebnis abweichen, doch selbst bei ungünstiger Lage der Hindernisse ist der A*-Algorithmus sehr genau. Zum Vergleich braucht der Dijkstra-Algorithmus ungefähr 2 Minuten, um einen Weg durch den Test-Raum zu finden. Die Genauigkeit des Dijkstra ist dafür unschlagbar, da er immer den kürzesten Weg findet. Der nicht optimierte A*-Algorithmus ist also ungefähr 24-mal so schnell wie der Dijkstra Algorithmus.

4.4.1.4 Evaluierung der Search Space Representation (SPR)

Bei dem Vergleich der beiden Search Space Representations, Octree und Grid gibt es einige Probleme, da es nicht möglich ist ein derart großes Grid für Unity zu erstellen und mit Gizmos darzustellen. Es wird somit für den Vergleich einmal der optimierte und einmal der nicht optimierte Octree genutzt.

Der optimierte Octree (Sparse Voxel Octree) besitzt 35.000 Nodes, da nur bei Hindernissen weiter unterteilt wird. Der normale Octree hingegen hat bei acht Schichten in dem Test-Raum 19.173.960 Nodes und ein Grid hätte bei einer Node Größe von 1x1x1 genau 1.000.000.000 Nodes. Der normale Octree besitzt also 550-mal so viele Nodes wie der optimierte Octree und ein Grid hat 28.000-mal so viele Nodes wie der optimierte Octree. Einen Vergleich der Geschwindigkeit für acht Schichten ist nicht möglich, da ein 8-schichtiger Octree mit 19 Millionen Nodes in Unity nicht zu erstellen ist. Man kann aber davon ausgehen, dass der optimierte Octree wesentlich schneller ist, da es viel weniger Nodes zu erkunden gibt. Wenn der normale und der optimierte Octree mit sechs Schichten erstellt wird, benötigt der Algorithmus ungefähr 3-mal so viel Zeit auf dem normalen Octree wie auf dem optimierten. Die Genauigkeit der Raum Repräsentation hängt davon ab wie groß die Nodes sind, in unserem Fall hängt die Größe der Nodes direkt mit der Anzahl an Schichten zusammen, je mehr Schichten desto genauer die Positionen.

Der normale Octree hat den Vorteil, dass an jeder Stelle die Genauigkeit der kleinsten Nodes gegeben ist. Bei dem optimierten Octree gibt es viele größere Nodes was den Weg etwas weniger „Echt“ aussehen lässt. Solche Probleme könnte man durch Path Smoothing und kleine Veränderungen lösen. Es könnte also sein, dass der Start und Zielpunkt in der gleichen Node sind. Hierfür habe ich eine Lösung gefunden. Sobald die beiden Punkte in der gleichen Node sind weiß man, dass es zwischen den beiden Punkten keine Hindernisse gibt. Also wird in dem Fall der Weg mit der Luftlinie zwischen den beiden Punkten gleichgesetzt.

4.4.2 Optimierung des A*-Algorithmus

Nach der Evaluierung des Algorithmus und der Raum Repräsentation wurde der A*-Algorithmus und der optimierte Octree für das Projekt ausgewählt, da sie die besten Ergebnisse geliefert haben. Der nächste Schritt ist, den Algorithmus zu optimieren. Hierfür wird die Heuristik Funktion des Algorithmus verändert.

4.4.2.1 Heuristik

Die Heuristik Funktion des A*-Algorithmus lautet wie folgt:

$$f(x) = g(x) + h(x)$$

h(x) steht für die geschätzte Distanz von einer Node zu der Ziel Node,

g(x) steht für die Distanz von der Start Node zu der betrachteten Node und

f(x) steht für die gesamte Strecke von der Start Node bis zur End Node, wenn der Weg über die betrachtete Node führt.

Um den Algorithmus zu beschleunigen kann der Heuristik-Teil der Funktion h(x) verändert werden. Der Heuristik-Teil kann durch eine Gewichtung angepasst werden:

$$f(x) = g(x) + (h(x) * Gewichtung)$$

Wenn man die Gewichtung auf 0 setzt hat man eine stark unterschätzende Heuristik die immer Null ist, das würde den A*-Algorithmus zu einem Dijkstra-Algorithmus umwandeln. Die Unterschätzung der Heuristik führt dazu, dass mehr Nodes untersucht werden und es wahrscheinlicher ist, den kürzesten Weg zu finden, jedoch wird dadurch auch die Zeit erhöht, die der Algorithmus benötigt, um einen Weg zu finden. Für dieses Projekt ist eine Unterschätzung ungeeignet.

Der Algorithmus soll so schnell wie möglich einen möglichen kurzen Weg finden. Hierzu wird die Überschätzung der Heuristik Funktion genutzt. Für das Projekt werden Gewichtungen zwischen 1.1 und 2 getestet. Es ist wichtig eine gute Gewichtung zu finden die nicht zu hoch ist, da sonst sehr schlechte Wege gefunden werden könnten. Eine Gewichtung in dem zuvor genannten Rahmen sollte es ermöglichen einen guten Weg in kurzer Zeit zu finden. Durch die Tests wird eine Gewichtung von 1.5 gewählt da sie die besten Ergebnisse liefert. Eine Gewichtung die höher als 1.5 ist bringt keine nennenswerten Zeitvorteile und sorgt nur dafür, dass ein schlechterer Weg gefunden wird. Der optimierte A*-Algorithmus benötigt nun statt 5 Sekunden nur noch durchschnittlich 7 Millisekunden um den gleichen Weg zu finden, wie zuvor in der ersten Evaluierung des Algorithmus. Der optimierte A*-Algorithmus ist drei Größenordnungen schneller als der nicht optimierte, also fast tausendmal schneller. Diese Optimierung ist ein Erfolg und wird für den Rest des Projekts beibehalten. Diese Zahlen kommen aus den Tests hervor, die Zahlen können jedoch variieren je nachdem wie komplex das Pfadfindungsproblem ist, bei weniger komplexen Problemen können die Zahlen sehr viel näher aneinander sein.

4.4.2.2 Hierarchische Pfadfindung

Durch die Verwendung des optimierten Octree ist eine Hierarchische Pfadfindung automatisch gegeben. Durch die verschieden großen Schichten werden immer kleinere Nodes in einer größeren Node zusammengefasst. Sobald eine Node die andere Nodes beinhaltet navigiert werden soll, müssen die Nodes betrachtet werden die die Kinder von der großen Node sind. Wenn eine größere Node keine Kinder hat wird sie einfach durchlaufen. Diese Methode hat den Vorteil, dass nicht jeder Teil des Raums in kleinstmögliche Nodes unterteilt werden muss. Mit diesem System ist eine Optimierungsmöglichkeit, den Algorithmus größere Nodes bevorzugen zu lassen. Um die größeren Nodes zu bevorzugen, wird bei der Zuweisung der G-Kosten die Hälfte der Größe der Nodes abgezogen. Diese Veränderung führt zu der gleichen Zeitersparnis wie die Gewichtung der Heuristik Funktion, jedoch nur dann, wenn die Heuristik Funktion nicht verändert wird. Wenn beide Optimierungen gleichzeitig implementiert sind, bleibt es bei einer Durchschnittszeit von 7 Millisekunden. Man kann also beide Verbesserungen einzeln nutzen, beide gleichzeitig sind aber unnötig.

Für das Projekt wird nur die Optimierung der Heuristik genutzt, da durch die Bevorzugung von größeren Nodes auch schlechtere Wege entstehen könnten.

4.5 Visualisierung

Um den Octree und den Algorithmus in Unity zu visualisieren werden Gizmos genutzt. Der Octree wird mit sogenannten Wire Cubes visualisiert. Die Umrandungen jeder Node werden dargestellt, so kann man sehen wie der Raum aufgeteilt ist. Der Start- und Endpunkt wird als leeres Objekt in Unity dargestellt, diese Punkte können an beliebige Positionen im Raum gezogen werden und symbolisieren Start und Ziel. Die Hindernisse sind Würfel, die der Größe der kleinsten Nodes entsprechen. Sobald der Algorithmus angewendet und ein Weg gefunden wurde, werden die gefundenen Nodes, die den Weg symbolisieren, mit einer blauen Farbe ausgefüllt. So kann man sehen welchen Weg der Algorithmus gefunden hat.

Nachdem ein Weg gefunden wurde bewegt sich eine Kugel, der eine Kamera folgt entlang des gefundenen Weges. So wird dargestellt wie sich ein NPC durch den Raum navigiert. In Abbildung 5 ist gut zu sehen wie der gefundene Pfad Visualisiert wird. Außerdem kann man in der Abbildung auch gut erkennen wie die einzelnen Nodes sichtbar gemacht wurden.

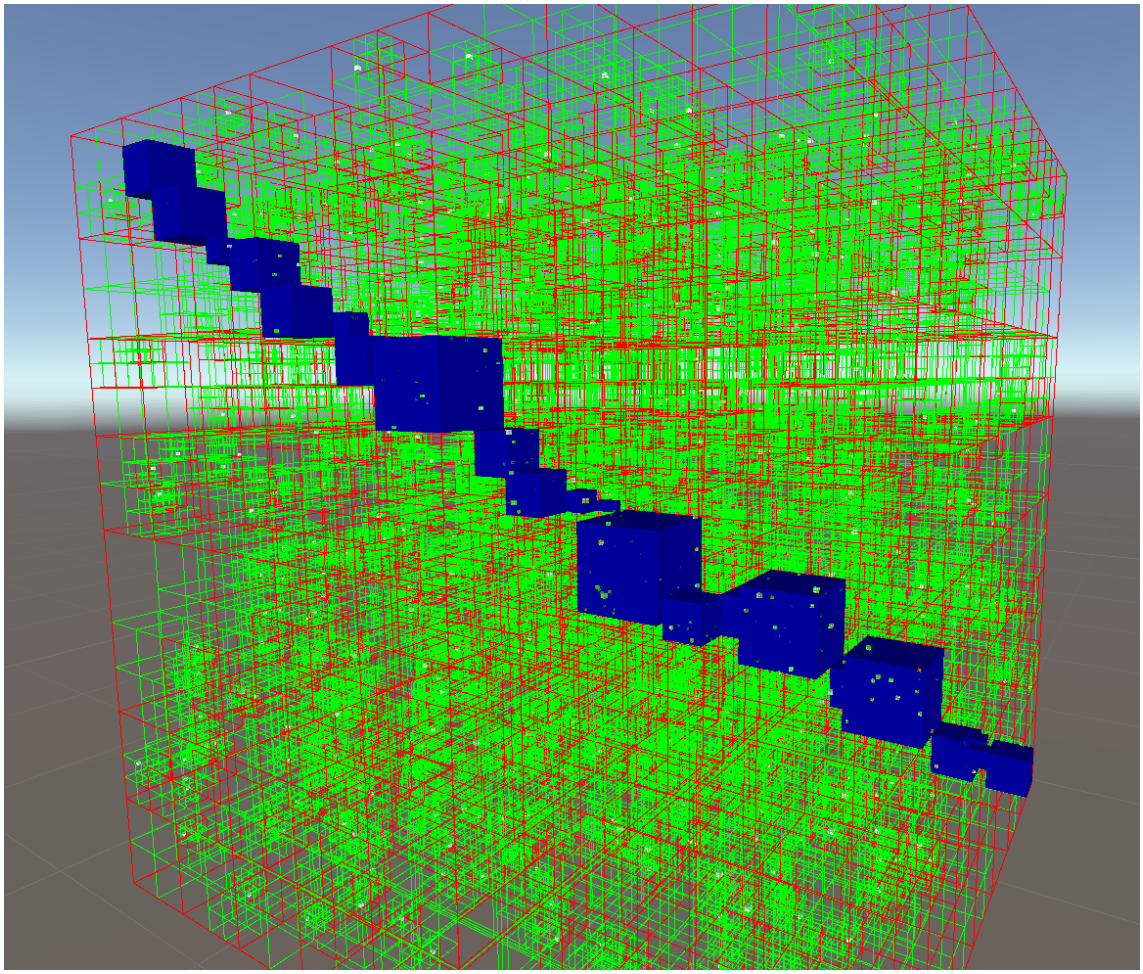

Abbildung 5 - Visualisierung des Gefundenen Pfads

5 Ergebnisse und Zusammenfassung

5.1 Ergebnis

Innerhalb des Projekts wurde ein voll funktionsfähiger 3-Dimensionaler-Algorithmus für eine Sparse Voxel Octree entwickelt. Es ist gelungen den Algorithmus und den Octree zu optimieren. Das Ziel, schnelle und ressourcenschonende 3D-Pfadfindung zu implementieren ist gelungen und alle Ziele wurden erreicht.

5.1.1 Algorithmus

Das Ziel einen 3-Dimensionalen-Pfadfindungs-Algorithmus zu implementieren ist gelungen. Alle notwendigen Aspekte wurden umgesetzt und die Vorgaben wurden erfüllt. Der Vergleich der beiden Algorithmen hat ergeben, dass der optimierte A*-Algorithmus in dem Test Rahmen circa 24-tausend-mal schneller ist als der Dijkstra-Algorithmus und tausend-mal so schnell wie der nicht optimierte A*-Algorithmus.

Ziel	Realisierung
Der Algorithmus muss auf eine geeignete Repräsentation des Raums anwendbar sein.	Die gewählte Repräsentation ist ein Octree. Der gewählte A*-Algorithmus wurde erfolgreich auf einen Sparse Voxel Octree angewendet und ist funktionsfähig.
Der Algorithmus muss 3D-Pfadfindung unterstützen.	Der A*-Algorithmus unterstützt 3D-Pfadfindung einwandfrei.
Die Heuristik des Algorithmus soll auf Geschwindigkeit statt Genauigkeit optimiert werden.	Die Heuristik Funktion des Algorithmus wurde so verändert, dass sie primär die Geschwindigkeit des Algorithmus begünstigt. Die Gewichtung der Heuristik Funktion führt zu einer erhöhten Geschwindigkeit. Die Genauigkeit ist durch die Veränderung niedriger aber immer noch gut genug für das Projekt.

Der Algorithmus muss mit einem weiteren Algorithmus verglichen werden.	Außer dem A*-Algorithmus wurde außerdem der Dijkstra-Algorithmus getestet und beide Algorithmen wurden verglichen.

5.1.2 Search Space Representation

Die Darstellungen des Raums durch einen Sparse Voxel Octree erfüllt alle Anforderungen. Der Implementierte Octree funktioniert einwandfrei und ist Ressourcen schonend.

Ziel	Realisierung
Die Repräsentation muss für 3D-Räume gut geeignet sein.	Zur Darstellung des Raums wurde der Sparse Voxel Octree ausgewählt. Der erstellte Sparse Voxel Octree ist sehr gut geeignet, um 3-Dimensionale-Räume darzustellen.
Der Raum soll möglichst Ressourcen schonend dargestellt werden.	Ein gewöhnlicher Octree bräuchte 19 Millionen Nodes, um den Beispielraum des Projekts darzustellen. Der Sparse Voxel Octree benötigt nur 40 Tausend Nodes. Je nach Menge der Hindernisse variiert die Anzahl an benötigten Nodes für den Sparse Voxel Octree. Mehr Hindernisse bedeutet mehr Nodes und andersherum. Für den Beispielraum mit 1000 Hindernissen benötigt der gewöhnliche Octree jedoch mehr als 450-mal so viele Nodes wie der optimierte Octree.
Die Repräsentation soll Hierarchische Pfadfindung unterstützen.	Durch die Wahl des Sparse Voxel Octree wird die hierarchische Pfadfindung ohne weitere Veränderungen unterstützt.

5.1.3 Visualisierung

Die Visualisierung des Projekts ist sehr grundlegend erfüllt aber seinen Zweck. Alle Ziele wurden erfüllt, es gibt jedoch einige Möglichkeiten das Projekt ästhetisch ansprechender zu gestalten, auch wenn es nicht nötig ist.

Ziel	Realisierung
Der gefundene Weg soll dargestellt werden.	Der gefundene Weg wird durch Unity Gizmos farbig markiert, die gefundenen Nodes werden in Blau ausgemalt. Start und Ziel werden durch leere Unity Gameobjects dargestellt.
Die Aufteilung des Raums soll dargestellt werden.	Die Darstellung des Raums wird ebenfalls durch Unity Gizmos eingezeichnet. Die einzelnen Nodes werden an den Kanten grün oder rot eingefärbt. Nodes mit Kindern werden rot eingefärbt und Nodes ohne Kinder sind grün.
Ein NPC soll dem gefundenen Pfad von Start zu Ziel folgen.	Nachdem ein Pfad gefunden wurde Spawnt der NPC auf dem Start Punkt und verfolgt Node für Node den Weg von Start zu Ziel. Eine Kamera verfolgt den NPC und dreht sich mit der Bewegung mit.

5.2 Zusammenfassung

Der Projektverlauf ist überwiegend positiv verlaufen, die zuvor gesteckten Ziele wurden alle umgesetzt. Regelmäßiges Feedback, Zeitmanagement und gute Planung haben zu einem erfolgreichen Projektergebnis geführt.

Die Zielsetzung und Planung wurden gut durchdacht, sodass die Durchführung größtenteils problemlos möglich war. Die geplanten Algorithmen und die Darstellung des Raumes wurden gut gewählt und haben sich als beste Lösung des Problems herausgestellt. Die gewählten Methoden haben maßgeblich dazu beigetragen, dass das Projekt umgesetzt werden konnte. Die Feedback

Methode hat dazu geführt, dass Fehler schnell erkannt wurden und wenig unnötige Arbeit entstanden ist.

Bei der Implementierung des Algorithmus gab es kleine Probleme, da die Standard Implementierung des A*-Algorithmus auf einen Octree angepasst werden musste. Außerdem gab es ein Problem, welches den Algorithmus einen diagonalen Weg durch eine versperrte Node finden ließ. Dieses Problem hat 2 Tage Zeit gekostet, da zuerst der falsche Ansatz verfolgt wurde. Nachdem festgestellt wurde, dass der Ansatz so nicht funktioniert, wurde ein besser durchdachter Ansatz gewählt, welcher dann funktioniert hat. Außerdem wurde im Zuge dieser Lösung ein Problem festgestellt, welches den Algorithmus verlangsamt hat, obwohl die Funktion nicht aufgerufen wurde. Auch mit Feedback Methode konnte dieses Problem nicht gelöst werden und ist somit auch Teil des Endergebnisses, hier könnte man in der Zukunft versuchen eine Lösung zu finden.

Ein weiteres Problem war die etwas kleine Menge an Informationen zu dem Thema, da es sehr aktuell ist und erst seit kurzem mehr erforscht wird. Die Planung des Projekts hätte etwas besser laufen können, da gegen Ende der Abgabe leichte Zeitprobleme aufgetreten sind. Diese Zeitprobleme konnten jedoch durch Engagement und Überstunden wieder eingeholt werden. Die Quellensuche dauerte länger als gedacht, da es wenig Informationen gab, musste länger nach passenden Quellen gesucht werden. Außerdem wurde zu spät ein konkreter Zeitplan erstellt. Die Zeitprobleme können für zukünftige Projekte einfach gelöst werden, indem frühzeitig ein guter Zeitplan erstellt und eingehalten wird. Trotz weniger praktischer Beispiele und fehlender Expertise für das Thema der Arbeit, konnte das Projekt erfolgreich durchgeführt und fertiggestellt werden. Das systematische Abarbeiten der theoretischen Informationen war der Schlüssel zum Erfolg des Projekts.

Schlussendlich bin ich sehr zufrieden mit dem Ergebnis der Arbeit, die Durchführung hat trotz kleiner Schwierigkeiten sehr viel Spaß gemacht und ist ebenfalls gelungen. Das Ziel der Arbeit wurde erfüllt, es konnte alles umgesetzt werden und der Untersuchungsauftrag wurde erfolgreich behandelt.

5.2.1 Aussicht

Das Projekt im Rahmen der Bachelorarbeit ist ein Erfolg, jedoch können einige Aspekte überarbeitet bzw. erweitert werden. Im Folgenden wird ein Ausblick gegeben, was mit dem Projekt geschehen könnte.

Die Einbußen in der Geschwindigkeit durch die Prüfung auf diagonale Bewegungen könnten mit mehr Zeit und Aufwand behoben werden. Außerdem könnten viele kleine Optimierungen am Code dazu führen, dass der Algorithmus noch schneller und somit performanter ist.
Das Ziel des Projekts in der Zukunft ist es, den gesamten Code von Unity loszulösen. So könnte man die erarbeitete Pfadfindung und den Sparse Voxel Octree in einer beliebigen Engine benutzen und dritten zur Verfügung stellen. Außerdem könnte das Produkt auch außerhalb der Videospiel Industrie verwendet werden.

Literaturverzeichnis

3D Pathfinding | AI Middleware for UE4 and Unity (2019): Mercuna, [online] https://mercuna.com/features/pathfinding/ [abgerufen am 21.08.2020].

3D Pathfinding | Mercuna (o. D.): Mercuna, [online] https://mercuna.com/category/pathfinding/ [abgerufen am 21.08.2020].

Algfoor, Zeyad Abd/Mohd Shahrizal Sunar/Hoshang Kolivand (2015): A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games, hindawi, [online] https://www.hindawi.com/journals/ijcgt/2015/736138/ [abgerufen am 20.08.2020].

Amit, P. (2010): Amit's A* Pages, theory.stanford, [online] http://theory.stanford.edu/%7Eamitp/GameProgramming/index.html [abgerufen am 23.08.2021].

Brewer, Dan (2015): Getting off the NavMesh: Navigating in Fully 3D Environments, GDC Vault, [online] https://www.gdcvault.com/play/1022016/Getting-off-the-NavMesh-Navigating [abgerufen am 23.08.2021].

Brewer, Daniel/Nathan R. Sturtevant (2018): Benchmarks for Pathfinding in 3D Voxel Space, semanticscholar, [online] https://www.semanticscholar.org/paper/Benchmarks-for-Pathfinding-in-3D-Voxel-Space-Brewer-Sturtevant/0b30027ae13a9c2c81e37b63e95fd61749b6bac2 [abgerufen am 21.08.2020].

Cui, Xiao/Hao Shi (2010): A*-based Pathfinding in Modern Computer Games., Researchgate, [online] https://www.researchgate.net/publication/267809499_A-based_Pathfinding_in_Modern_Computer_Games [abgerufen am 21.08.2020].

Denker, Yannick (2019): *Zur Verwendung von Reinforcement Learning nutzenden neuronalen Netzwerken zum Mapping für Pathfinding auf verschiedenen 3D Objekten - Untersuchung unter besonderer Berücksichtigung der Deep Q-Learning-Strategie und des A*-Algorithmus*, Bachelor, Games Programming, Hamburg, Deutschland: SAE Institut.

He, Zhang/Minyong Shi/Chunfang Li (2016): Research and application of path-finding algorithm based on unity 3D, IEEE Conference Publication, [online] https://ieeexplore.ieee.org/document/7550934 [abgerufen am 09.02.2021].

IronEqual (2018a): Pathfinding Like A King — Part 1 - IronEqual, Medium, [online] https://medium.com/ironequal/pathfinding-like-a-king-part-1-3013ea2c099 [abgerufen am 19.08.2020].

IronEqual (2018b): Pathfinding Like A King — Part 2 - IronEqual, Medium, [online] https://medium.com/ironequal/pathfinding-like-a-king-part-2-4b74588262af [abgerufen am 19.08.2020].

Krafft, Carina (2019): *Implementation and comparison of pathfinding algorithms in a dynamic 3D space*, Bachelor Thesis, Design, Media and Information, Hamburg, Deutschland: Hochschule für Angewandte Wissenschaften Hamburg.

Laine, Samuli/Tero Karras (2010a): Efficient Sparse Voxel Octrees | Research, research.nvidia, [online] https://research.nvidia.com/publication/efficient-sparse-voxel-octrees [abgerufen am 23.02.2021].

Laine, Samuli/Tero Karras (2010b): Efficient Sparse Voxel Octrees - Analysis, Extensions, and Implementation | Research, research.nvidia, [online] https://research.nvidia.com/publication/efficient-sparse-voxel-octrees-analysis-extensions-and-implementation [abgerufen am 23.02.2021].

Millington, Ian (2019): *AI for Games, Third Edition*, 3. Aufl., Boca Raton, Florida, USA: CRC Press.

Mueller, John Paul/Luca Massaron (2017): *Algorithmen für Dummies (German Edition)*, 1. Aufl., Hoboken, New Jersey, USA: Wiley-VCH.

Niu, Lei/Guobin Zhou (2008): AN IMPROVED REAL 3 D A * ALGORITHM FOR DIFFICULT PATH FINDING SITUATION, semanticscholar, [online] https://www.semanticscholar.org/paper/AN-IMPROVED-REAL-3-D-A-*-ALGORITHM-FOR-DIFFICULT-Niu-State/7e13c913a7f4a6377088df0ca6f940da876698af [abgerufen am 21.08.2020].

Rabin, Steve (2019): *Game AI Pro 360: Guide to Movement and Pathfinding*, 1. Aufl., Abingdon, Vereinigtes Königreich: Routledge.

Schwarz, Michael/Hans-Peter Seidel (2010): Fast parallel surface and solid voxelization on GPUs, research.michael-schwarz.com, [online] http://research.michael-schwarz.com/publ/2010/vox/ [abgerufen am 23.08.2021].

Smołka, Jakub/Kamil Miszta/Maria Skublewska-Paszkowska/Edyta ŁUkasik (2019): A* pathfinding algorithm modification for a 3D engine | MATEC Web of Conferences, matec-conferences, [online] https://www.matec-conferences.org/articles/matecconf/abs/2019/01/matecconf_cmes2018_03007/matecconf_cmes2018_03007.html [abgerufen am 21.08.2020].

Stamford, John/Arjab Singh Khuman/Samad Ahmadi/Jenny Carter (2014): Pathfinding in partially explored games environments: The application of the A∗ Algorithm with occupancy grids in

Unity3D, University of Huddersfield Research Portal, [online] https://pure.hud.ac.uk/en/publications/pathfinding-in-partially-explored-games-environments-the-applicat [abgerufen am 21.08.2020].

Stein, Daniel (2018): *Zur Programmierung eines Autorennspiels unter Unity: Die Planung und Implementation einer künstlichen Intelligenz des Fahrens und der Wegfindung*, Bachelor, Games Programming, Hamburg, Deutschland: SAE Institut.

Unity (o. D.): Unity, [online] https://unity.com/de [abgerufen am 23.08.2021].