PROJEKTARBEIT

Wegfindung in einer zufällig generierten Welt

Inhalt

1 Einleitung 3
2 Perlin-Rauschen 3
2.1 Funktionsweise 4
2.2 Programm 7
3 Wegfindung 7
3.1 A* 7
3.1.1 Heuristik 8
3.2 Dijkstra 9
3.3 Programm 9
4 Fazit 9
Endnoten 10
Anhänge 10
Literaturverzeichnis 11

1 Einleitung

Die Auswahl der Kurse für mein Projekt war eine kleine Hürde, da ich keine Ahnung hatte, was ich tun sollte. Ein paar Wochen nachdem ich darüber nachgedacht hatte, wusste ich, dass es um Informatik gehen muss. Das eigentliche Thema kam kurz vor Ablauf der Frist, als ich in Spielen mit computergesteuerten Gegnern herumspielte. Gedanken über, wie sich den Gegnern leicht auf den Spieler zubewegen und Hindernissen ausweichen, erfüllten meinen Kopf. „Wie aufwendig ist es für den Rechner, diese Pfade zu berechnen?“, „Welche Parameter werden berücksichtigt, um ein Zeil zu erreichen?“. Ich dachte, das wäre ein tolles Thema und habe mich entschieden, ein Programm zu erstellen, das einen optimalen Weg findet. Aufgrund meiner langjährigen Erfahrung habe ich Java[1] als Programmiersprache für das Produkt genommen. Einer der wichtigsten Punkte war, dass die Sprache eine „Library“ („Programmbibliothek“) hatte, um Fenster zu erstellen und darauf zu zeichnen. Java hat eine eingebaute Library namens Swing, die perfekt für den Job geeignet war.

Meiner Meinung nach war die Wegfindung an sich aber ein ziemlich einfaches Thema. Ich wollte das ganze Programm unterhaltsamer bauen, indem ich eine zufällige Welt generierte. Damit wurde das ganze Projekt auch in den Vektorbereich der Mathematik erweitert. Jetzt war die Frage „Wie kann ich eine zufällige Welt erstellen?“. Zum Glück hatte ich das perfekte Werkzeug für diesen Job aufgrund meiner früheren Erfahrungen mit der zufälligen Welterstellung, hauptsächlich aus Spielen wie Minecraft[2]. Es heißt „Perlin Noise“ („Perlin-Rauschen“) und wurde 1982 von Ken Perlin entwickelt, mehr dazu unter Kapitel „Perlin-Rauschen“. Das Einzige, das ich noch durchdenken musste, war, wie ich die Welt und den Pfad darstellen würde. Aus dem Konzept eines Pfadfindungsprogrammes (siehe Abb. 1) von Xueqiao Xu[3] habe ich mich für ein Raster entschieden. Dies funktioniert, weil die Wegfindung mit miteinander verbundenen „Knoten“ arbeitet und ein Raster ist im Grunde eine geordnete Menge von Knoten.

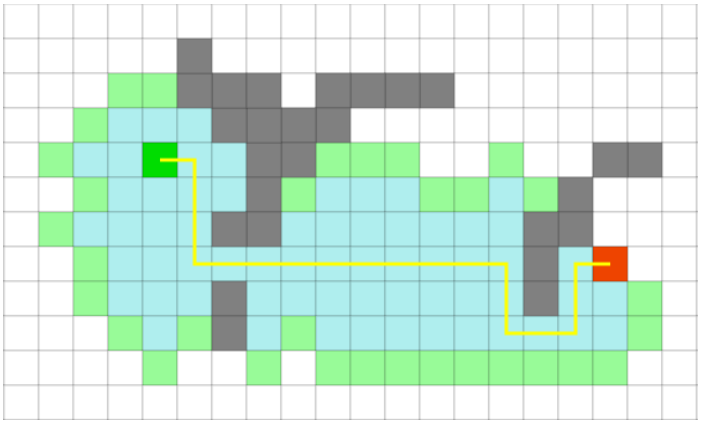

Abbildung 1: Pathfinding.js, Quelle: Xueqiao Xu, et.al., <https://qiao.github.io/PathFinding.js/visual/>, 22. Februar 2022

Nachdem alle wichtigen Punkte zusammengetragen waren und ich ein grundlegendes Verständnis dafür hatte, was ich tun wollte, ging es nur noch darum, die richtige Herangehensweise zu finden, um sie in ein leicht verständliches Programm umzusetzen.

2 Perlin-Rauschen

Das Wichtigste zuerst, ich brauchte eine Karte, auf der die Wegfindung ihre Arbeit erledigen konnte. Wie zuvor beschrieben, habe ich Perlin-Rauschen für diesen Job ausgewählt, da es perfekt ist, um ein zufälliges und vor allem realistisch aussehendes Muster bzw. Textur zu erstellen.

Perlin-Rauschen ist im Grunde eine Funktion, die n (Anzahl der Dimensionen) Koordinaten nimmt und einen scheinbar zufälligen Wert im Bereich von -1 bis 1 zurückgibt. Je nach Frequenz der Funktion würden bei der Angabe von weit auseinander liegenden Koordinaten die Werte zwar zufällig erscheinen, aber bei der Angabe von nahen Punkten würden nur geringfügig unterschiedliche Werte zurückgegeben. Genau diese Eigenschaft von Perlin-Rauschen ermöglicht es, natürlich aussehende Texturen zu erstellen. Zum Beispiel mit den folgenden $\binom{x}{y}$ Koordinaten: $f_{Perlin}\binom{1}{1} = 0{,}51 \quad f_{Perlin}\binom{1}{2} = 0{,}54 \quad f_{Perlin}\binom{20}{5} = -0{,}2$

Ein 2-dimensionales Perlin-Rauschen würde beispielswiese verwendet, um Feuereffekte, Wasser und Wolken zu erzeugen [Biagoli Adrian, Absatz 2], kann aber auch verwendet werden, um eine natürlich aussehende Vogelansicht einer Welt zu erstellen.

Perlin-Rauschen kann in beliebig vielen Dimensionen verwendet werden, für den Zweck dieses Projekts ist es jedoch zweidimensional.

2.1 Funktionsweise

Die zweidimensionale Funktion, die dieses Projekt verwendet, hat zwei Eingabewerte, die x- und y-Koordinate eines Punktes. Zum Anfang stellen wir uns ein Raster vor, bei dem jeder Schnittpunkt eine ganze Zahl ist, und setzen den Punkt auf die angegebenen Koordinaten. Dieser Punkt ist nun von 4 weiteren Punkten eines Quadrats umgeben (siehe Abb. 2).

Nachdem diese 4 Punkte erfasst wurden, wird an jedem Punkt ein pseudozufälliger Vektor, auch „Gradientenvektor“, erstellt (siehe Abb. 3). [Zucker Matt, „How do you generate Perlin noise?“, Absatz 2-3]

$$g(x_{nEcke}, y_{nEcke}) = \begin{pmatrix} x_n \\ y_n \end{pmatrix}$$

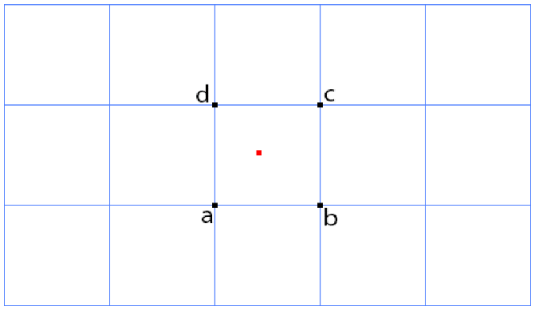

Abbildung 2, Quelle: selbstgemacht

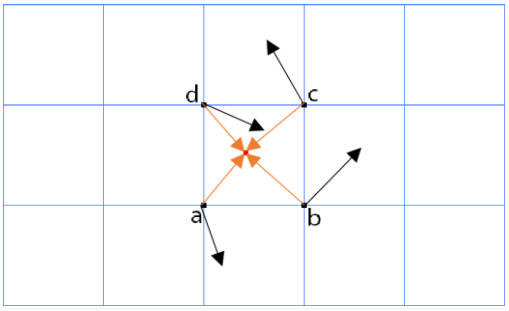

Abbildung 3, Quelle: selbstgemacht

Die Gradientenvektoren müssen bei jedem Zugriff an einem Punkt konstant sein. Daher wird eine konstante Liste mit zufälligen Gradientenvektoren (meistens 256) erstellt oder ein Pseudozufallsgenerator verwendet. Perlin-Rauschen wird sich aus diesem Grund irgendwann wiederholen, aber da Koordinaten Dezimalstellen haben, ist es normalerweise schwierig, diese Grenze zu erreichen.

Es gibt auch einen sogenannten Startwert („Random Seed“ auf Englisch), der im ursprünglichen Perlin-Rauschen nicht vorhanden ist. Es ändert das Aussehen des Rauschens mit einem ganzzahligen Startwert, der vom Benutzer angegeben wird. Eine einfache Methode ist die Koordinaten mit dem Startwert zu multiplizieren. Dies ist aber keine ideale Lösung, da die Werte des Rauschens nur an einer anderen Position verwendet werden, anstatt ein völlig anderes Rauschen zu erzeugen. Es gibt bessere Lösungen, aber sie erfordern die Verwendung von Hashfunktionen, die viel zu kompliziert sind, um sie in diesem Projekt zu erklären.

Wir berechnen auch den Verbindungsvektor von jedem Eckpunkt zum Eingabepunkt.

$$\overrightarrow{v_n} = \begin{pmatrix} x \\ y \end{pmatrix} - \begin{pmatrix} x_{nEcke} \\ y_{nEcke} \end{pmatrix}$$

Nachdem alle Vektoren berechnet sind, muss das Skalarprodukt zwischen jedem Paar von Gradientenvektoren und Verbindungsvektoren erhalten werden.

$$s = g(x_{nEcke}, y_{nEcke}) \cdot \overrightarrow{v_n}$$

Das Skalarprodukt ist positiv, wenn es in Richtung eines Gradienten geht (siehe Abb. 4). Denn wenn 2 Vektoren in dieselbe Richtung zeigen, ist der Kosinus des Winkels zwischen ihnen positiv (unter 90°). Das Gegenteil ist der Fall, wenn die Vektoren in entgegengesetzte Richtungen zeigen (über 90°). [Biagoli Adrian, Überschrift: „Logical Overview“]

$$s = cos(\alpha) \cdot g(x_{nEcke}, y_{nEcke}) \cdot \overrightarrow{v_n}$$

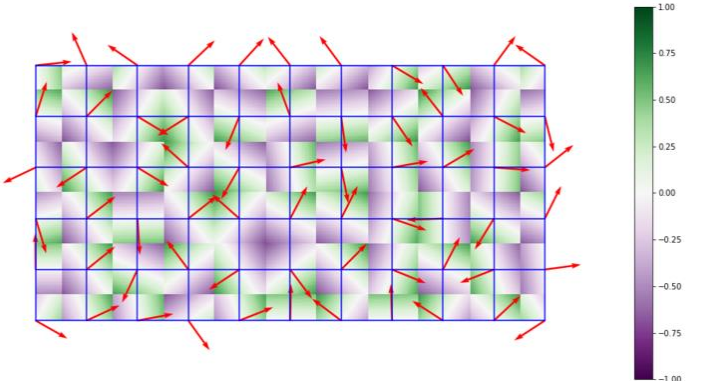

Abbildung 4: „Dot product of the gradient with the distance to the nearest node for generating Perlin noise“, Quelle: Matthewslf, <https://en.wikipedia.org/wiki/File:PerlinNoiseDotProducts.png>, 31. März 2019

Ein Skalarprodukt wird hier auch als Einfluss bezeichnet, weil wir einen Durchschnitt aus den Einflüssen nehmen, aber weil der Einfluss an einem Punkt höher ist als an dem anderen, wird ein normaler Durchschnitt nicht funktionieren, sondern wir müssen einen gewichteten Durchschnitt verwenden. Die Werte müssten linear interpoliert werden, was prozentual einen Wert zwischen zwei Werten ergibt. $x_0 + p \cdot (x_1 - x_0)$

Ein Problem dabei ist, dass bei Verwendung des vorhandenen Eingabepunkts die erzeugten Werte aufgrund der linearen Interpolation nicht glatt wären. Die Lösung dafür ist eine sogenannte „Ease Curve“. Es ist eine S-förmige Kurve, die den eingegebenen Wert näher an 0 ändert, wenn er unter 0,5 liegt, und näher an 1, wenn er über 0,5 liegt. Bei 0,5 bleibt der Wert unverändert. Ken Perlin hat $3t^2 - 2t^3$ verwendet (siehe Abb. 5). [Zucker Matt, „How do you generate Perlin noise?“, Absatz 13]

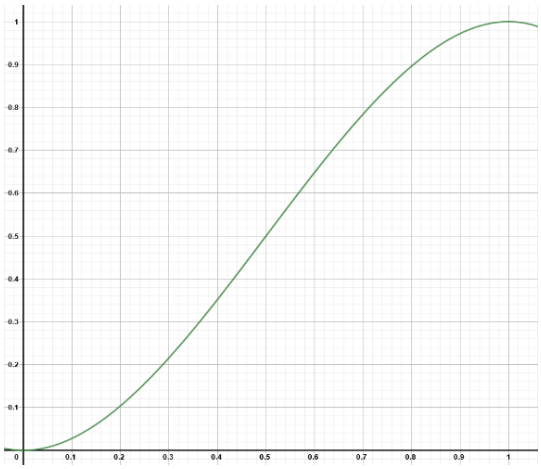

Abbildung 5: „Glattungskurve", Quelle: GeoGebra, <https://www.geogebra.org/>, 22. Februar 2022

Da die Kurve nur zwischen 0 und 1 funktioniert, müssen wir unseren Eingabepunkt nehmen und ihn so gestalten, dass wir nur die Koordinaten innerhalb des Quadrats haben, d.h. der Dezimalteil. Wir nehmen die x- und y-Werte des Punktes und subtrahieren sie von den niedrigsten x- und y-Punkten des Quadrats. Wir nennen die neuen Werte u und v.

Diese Kurve bewirkt, dass sich der Eingabepunkt so verhält, als ob er näher an dem einen oder anderen Punkt des Quadrats wäre, als er tatsächlich ist [Zucker Matt, „How do you generate Perlin noise?“, Absatz 11]. Wenn es auf Perlin-Rauschen verwendet wird, gibt es einen glatteren Übergang zwischen den Punkten des Quadrats. Dies liegt daran, dass der geglättete Punkt als Prozentsatz für den gewichteten Durchschnitt verwendet wird (siehe Abb. 6).

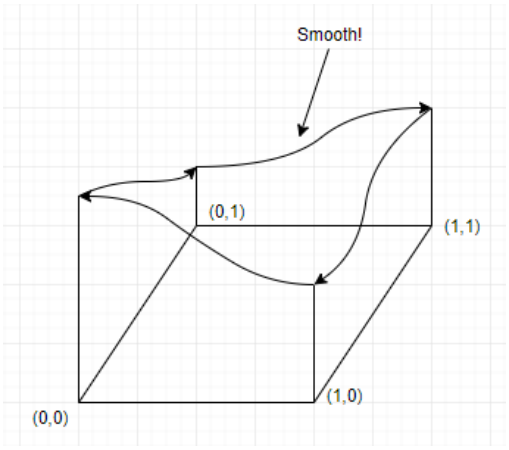

Abbildung 6: „The smooth transition between the corners of a grid square", Quelle: Touti Raouf, <https://rtouti.github.io/graphics/perlin-noise-algorithm>, 26. Februar 2022

Die Skalarprodukte werden in zwei Paare aufgeteilt, eines pro Achse. Wir berechnen dann die lineare Interpolation jedes Paares, wobei das p in der Formel je nach Achse auf u oder v gesetzt wird und x_0 auf das niedrigere Skalarprodukt und x_1 auf das höhere Skalarprodukt gesetzt wird [Zucker Matt, „How do you generate Perlin noise?“, Absatz 15].

Nachdem wir zwei interpolierte Werte haben, verwenden wir erneut die lineare Interpolation an diese Werte, diesmal jedoch mit dem anderen u- oder v-Wert [Zucker Matt, „How do you generate Perlin noise?“, Absatz 16].

Das war es. Der Rauschwert an den Eingabekoordinaten wurde berechnet und wir können jetzt die Welt erstellen (siehe Abb. 7).

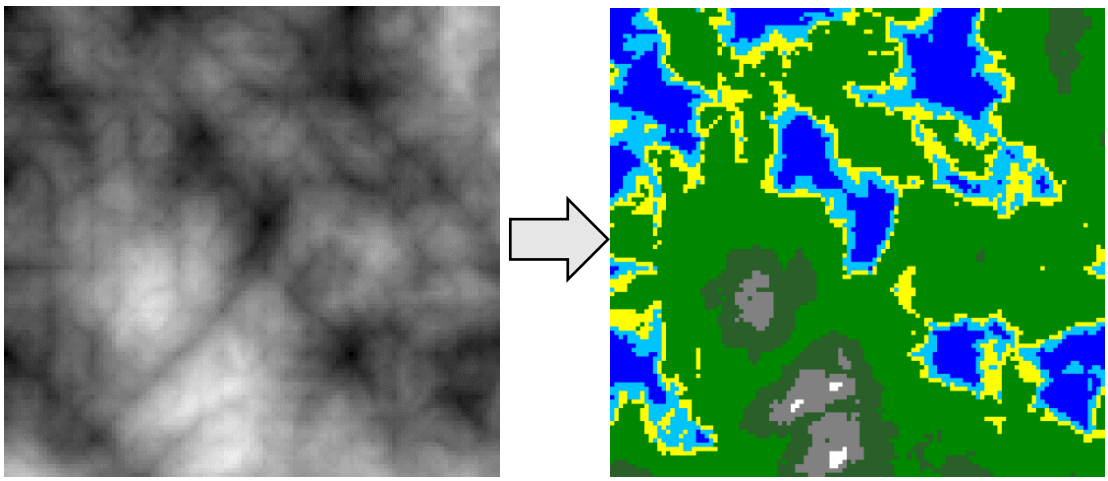

Abbildung 7: Perlin Noise, Quelle: selbstgemacht.
Je heller ein Punkt ist, desto höher ist sein Wert. Die Werte sind hier als Erhebung interpretiert.

2.2 Programm

In dem Programm verwende ich eine leicht modifizierte Version von Perlin-Rauschen aus FastNoiseLite[4] Programmbibliothek.

Einige der Zusatzfunktionen sind den zuvor beschriebener Startwert und Oktaven, d.h. es werden n (Anzahl der Oktaven) Perlin-Rauschen-Funktionen mit unterschiedlichen Frequenzen (multipliziert/dividiert die gegebenen x- und y-Koordinaten) und unterschiedlichen Amplituden (ändert das Ergebnis) erzeugt. Diese Funktionen werden alle miteinander addiert und ergeben eine sehr detaillierte Funktion. [Touti Raouf, Überschrift: „Fractal brownian motion (FBM)“] Eine einzelne Perlin-Rauschen-Funktion hat eine sehr glatte Textur, während eine bergähnliche erwünscht war (siehe Abb. 8).

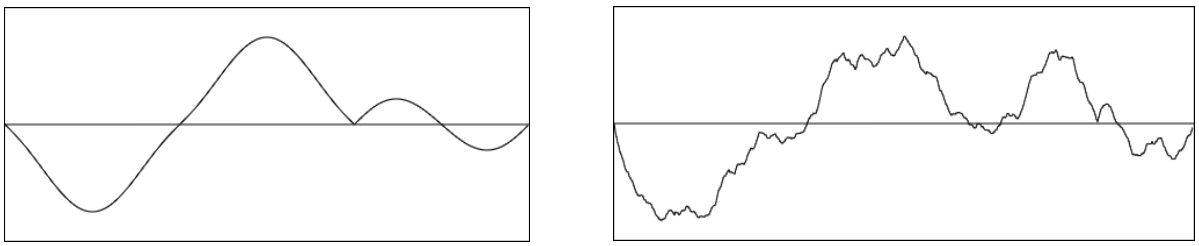

Abbildung 8: "Einzel- und 8-Oktaven Perlin-Rauschen-Funktion",
Quelle: Touti, Raouf, <https://rtouti.github.io/graphics/perlin-noise-algorithm>

3 Wegfindung

Nun zur Wegfindung. Wegfindung ist eine algorithmische Suche nach einem Pfad von einem Start- bis zu einem Endpunkt. Normalerweise wird es verwendet, um den kürzesten bzw. optimalen Pfad zu finden, aber es muss nicht unbedingt sein [Wikipedia „Pathfinding“, Absatz 1]. Dieses Projekt versucht immer den optimalen Pfad zu finden, es kann jedoch so eingestellt werden, dass es der Berechnungszeit den Vorrang vor der Pfadlänge gibt, was meistens zu einem nicht optimalen Pfad führt. Die Priorisierung einer kürzeren Berechnungszeit anstelle der Pfadlänge ist jedoch besonders wichtig in Anwendungen, bei denen die Computerressourcen anderen Teilen des Programms zugewiesen werden könnten.

Es gibt zahlreiche Möglichkeiten bzw. Algorithmen, um einen Pfad zu berechnen, aber dieses Projekt wird sich auf Dijkstra (von Edsger W. Dijkstra erfunden) und A* („A-star“) fokussieren.

3.1 A*

Ich beginne mit A*, da es wirklich einfach ist, den Dijkstra-Algorithmus daraus später zu erstellen. A* ist einer der an den häufigsten verwendeten Algorithmen in Computerspielen, da er ziemlich einfach zu verwenden ist und in den meisten Fälle sehr schnell einen Weg findet. [Patel Amit, Überschrift: „A*’s Use of the Heuristic”].

Der Algorithmus versucht, den besten Weg zu finden, indem er nach den niedrigsten Kosten sucht. Die Kosten (oder auch das Gewicht) $\mathrm{f(n)}$ an jedem n-Knoten sind die Summe aus $\mathrm{g(n)}$, den Kosten des Pfades vom Anfang zum n-Knoten, und der Heuristik $\mathrm{h(n)}$, den geschätzten Kosten des Pfades vom n-Knoten bis zum Ende.

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

Der Algorithmus wird auf folgende Weise implementiert [Wikipedia „A*-Algorithmus“]:

1. Eine Liste erstellen, die alle zu durchsuchenden Knoten enthält.
    - Der Startknoten zur Liste hinzufügen.
2. Eine Möglichkeit für jeden n-Knoten, auf den „Eltern“-Knoten zuzugreifen, d.h. den Knoten, von dem der n-Knoten stammt.
3. Eine Schleife, die läuft, solange die Liste nicht leer ist.
    - Bei jeder Wiederholung wird den Knoten mit den niedrigsten Kosten genommen und dann aus der Liste entfernt.
    - (Wenn der Knoten gleich dem Endknoten ist, wurde der Pfad gefunden und sollte erreicht werden, indem die „Eltern“-Knoten bis zum Start durchlaufen werden.)
    - Alle Nachbarknoten abrufen und mit entsprechenden Kosten zur Liste hinzufügen.
4. Der Algorithmus stoppt, wenn er das Ende erreicht oder wenn kein Pfad gefunden wird.

3.1.1 Heuristik

Die Heuristik von A*-Wegfindung ist, wie zuvor gesagt, die geschätzten Kosten des Pfades von n-Knoten bis zum Ende, also es ist einfach ausgedrückt nur die Entfernung.

Es ist möglich, jede Art von Entfernungsberechnung zu verwenden, aber es gibt einige, die besser funktionieren würden als andere. Was ich damit meine ist, dass einige Arten dazu führen würden, dass A* viel länger läuft als andere. Es ist auch möglich, zu überschätzen und einen längeren als optimalen Pfad zu erhalten, aber die Entfernung allein würde das nicht tun. Die Heuristik, die immer einen optimalen Weg findet, egal wie lange es dauert, heißt zulässige Heuristik [Wikipedia „Admissible Heuristics“, Absatz 1].

Wenn $\mathrm{h(n)}$ kleiner oder gleich den Kosten für die Bewegung von n-Knoten zum Ende ist (zulässige Heuristik), findet der Algorithmus garantiert den optimalen Weg. Wenn es gleich ist, werden unnötige Knoten nicht untersucht, d.h. je niedriger $\mathrm{h(n)}$ ist, desto mehr unnötige Knoten werden untersucht [Patel Amit, Überschrift: „A*’s Use of the Heuristic”].

Wenn $\mathrm{h(n)}$ größer als die Bewegungskosten ist (nicht zulässige Heuristik), ist es nicht garantiert, dass es den optimalen Weg findet, aber es kann schneller laufen [Patel Amit, Überschrift: „A*’s Use of the Heuristic”].

Es ist allgemein eine schlechte Idee, Pythagoras zur Berechnung der Entfernung (euklidische Entfernung) in einem Raster zu verwenden, da es einen viel niedrigeren Wert zurückgeben kann und A* viel mehr unnötige Knoten untersuchen würde [Patel Amit, Überschrift: „Euclidean distance“].

Stattdessen sollte die Manhattan-Entfernung mit 4 Richtungen (links, rechts, oben, unten) verwendet werden [Patel Amit, Überschrift: „Manhattan distance“].

$$|x_2 - x_1| + |y_2 - y_1|$$

Bei der Bewegung in 8 Richtungen sollte die Chebyshev- oder Octile-Entfernung verwendet werden. Die Chebyshev-Entfernung markiert alle 8 Richtungen mit der gleichen Entfernung, während Octile die Diagonale etwas länger macht ($\sqrt{2}$) [Patel Amit, Überschrift: „Diagonal distance“].

Grundformel: $D \cdot (|x_2 - x_1| + |y_2 - y_1|) + (D_2 - 2 \cdot D) \cdot min(|x_2 - x_1|, |y_2 - y_1|)$

Chebyshev: $D = 1, D_2 = 1$

Octile: $D = 1, D_2 = \sqrt{2}$

3.2 Dijkstra

Dijkstra ist im Grunde ein A*-Algorithmus, wobei die Heuristik $\mathrm{h(n)}$ an jedem n-Knoten auf Null gesetzt wird, sodass nur die Kosten jedes Knotens berücksichtigt werden [Patel Amit, Überschrift „A*'s Use of the Heuristic"].

$$f(n) = g(n) + h(n) \rightarrow \mathrm{f(n) = g(n) + 0}$$

In einem Raster mit konstanten Kosten würde sich der Algorithmus in alle Richtungen gleich ausdehnen. Aus diesem Grund wird ein Knoten nur einmal zur Liste hinzugefügt, wobei ein Knoten mehrmals zur Liste in A* hinzugefügt werden könnte. Dies tritt auf, wenn ein „Eltern"-Knoten geringere Kosten als der vorherige hat.

Der Algorithmus wurde aus A* erstellt, aber es gab ihn schon lange vorher, sodass A* tatsächlich als Erweiterung von Dijkstra angesehen werden kann [Wikipedia „A*-Algorithmus", Absatz 1].

3.3 Programm

Im Programm habe ich die Algorithmen genauso implementiert wie beschrieben, aber ich musste noch etwas hinzufügen, um anzuzeigen, wie die Algorithmen die Welt entdecken. Das war einfach, da ich nur eine Liste brauchte, zu der ich alle überprüften Knoten hinzufügen würde. Dann habe ich nur einen Index, der bei 0 beginnt und sich langsam erhöht, bis er die Größe dieser Liste erreicht. Während dieser Index zunimmt, werden alle untersuchten Knoten bis zu ihm gezeichnet.

Ein kleines Problem, das ich mit A* hatte, ist, dass es etwas länger dauerte, weil es mehrere Pfade gleichzeitig durchsuchte, anstatt sich nur auf einen zu fokussieren. Dies ist natürlich, da es auf einem Raster gleichzeitig mehrere Pfade mit der gleichen Entfernung geben kann. Eine Möglichkeit, dies zu beheben, ist den Heuristik leicht zu erhöhen, sodass nur der erste gefundene Pfad erst untersucht wird und nicht alle gleichzeitig [Patel Amit, Überschrift „Breaking ties"]. Ich habe den Wert um 0,05% erhöht. Dies kann natürlich dazu führen, dass die Heuristik nicht zulässig ist, aber es sollte kein großes Problem darstellen.

4 Fazit

Am Ende habe ich etwas bekommen, mit dem ich zufrieden bin. Das Programm verfügt über eine große Karte auf der linken Seite, dass die zufällig generierte Welt anzeigt, und ein rechtes Bedienfeld auf der rechten Seite, das viele verschieden Einstellungen anzeigt. Das Aussehen und Informationen zu den Einstellungen finden Sie im Anhang „Das Programm". Das Programm und der Code können von dem itsLearning Q1d-Mathematikkurs oder von Dropbox unter „https://www.dropbox.com/sh/33qnu6mfagd0tnw/AADoq2-LJQgC6psQexspDOF_a?dl=0" heruntergeladen werden.

Ich hatte keine wirklichen Schwierigkeiten bei der Erstellung des Programms, da ich vorher genug recherchiert hatte, um mich mit dem Thema wohl zu fühlen und bin jetzt wirklich zufrieden mit mir selbst, da ich noch nie so tief in das Thema, was ich täglich sehe, eingetaucht bin.

Die Welterstellung war am einfachsten zu programmieren, da ich eine Programmbibliothek verwendet habe, aber es war am schwierigsten zu erklären, da sie visualisiert werden muss und viel Vektormathematik involviert ist. Dank der verschiedenen Webseiten und deren Autoren habe ich es geschafft, das Thema zu verstehen. Die Wegfindung hingegen war insgesamt ein leicht zu verstehendes Thema. Deshalb habe ich den Code mit Hilfe der Schritte, die der Algorithmus durchläuft, selbst geschrieben.

Wie lange der Rechner braucht, um den Weg zu berechnen, hängt vom Algorithmus ab. Wenn es viele Hindernisse auf dem Weg zum Endpunkt gibt, kann A* länger dauern als Dijkstra. Aber wenn es keine Hindernisse gibt, wird A* fast immer am schnellsten sein.

Dijkstra geht problemlos um Hindernisse herum, da es sich in alle Richtungen ausdehnt, aber A* trifft zuerst auf das Hindernis und beginnt dann, vom Knoten mit niedrigsten Kosten rückwärts zu gehen, bis es das Hindernis umgeht.

Ich bin mit diesen Antworten ziemlich zufrieden, da alles vom Algorithmus abhängt. Es gibt viele andere Wegfindungsalgorithmen, die den Pfad möglicherweise schneller finden könnten, aber ich habe sie in diesem Projekt nicht verwendet.

Endnoten

[1] Oracle Corporation, Java, <https://www.java.com>, 2022. Eine Programmiersprache.

[2] Mojang Studios, Minecraft, <https://www.minecraft.net>, 2022. Ein Spiel.

[3] Xu, Xueqiao et.al., PathFinding.js, <https://qiao.github.io/PathFinding.js/visual/>, 24. April 2017. Wegfindungsprogramm.

[4] Peck, Jordan et.al., FastNoiseLite, <https://github.com/Auburn/FastNoiseLite>, 29. November 2021. Eine Programmbibliothek.

Anhänge

- „Das Programm“

Literaturverzeichnis

Biagioli, Adrian, „Understanding Perlin Noise”, <https://adrianb.io/2014/08/09/perlinnoise.html>, 9. August 2014

Patel, Amit, „Heuristics”, <http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html>, 22. Februar 2022

Touti, Raouf, „Perlin Noise: A Procedural Generation Algorithm”, <https://rtouti.github.io/graphics/perlin-noise-algorithm>,

Wikipedia, „A*-Algorithmus“, <https://de.wikipedia.org/wiki/A*-Algorithmus>, 1. Februar 2022

Wikipedia, „Admissible heuristic”, <https://en.wikipedia.org/wiki/Admissible_heuristic>, 22. Februar 2022

Wikipedia, „Pathfinding“, <https://de.wikipedia.org/wiki/Pathfinding>, 27. Februar 2022

Zucker, Matt, „The Perlin noise math FAQ”, <https://mzucker.github.io/html/perlin-noise-math-faq.html>, Februar 2001

Das Programm