SEMINARARBEIT

Rahmenthema des Wissenschaftspropädeutischen Seminars:

Mathematik in Natur und Technik

Leitfach: Mathematik

Thema der Arbeit:

Prozedurale Generation von Labyrinthen

-

Vergleich von zwei Methoden

Verfasser: Theresa Treimer

Kursleiter: [redacted]

Abgabe: 06.11.2018

Bewertung	Note	Notenstufe in Worten	Punkte		Punkte
schriftliche Arbeit				x 3	
Abschlusspräsentation				x 3	
				Summe:	
				Gesamtleistung = Summe: 2 (gerundet)	

Datum und Unterschrift des Kursleiters

0 Inhaltsverzeichnis

1 Notwendigkeit der prozeduralen Generierung für die heutige Videospiel-Industrie

Während viele ältere Spiele von Hand entwickelte Level und Welten beinhalteten, konnte man vor allem in den letzten Jahren einen Anstieg der Spiele mit prozedural generierten, also nicht von Game Designern erstellte, Inhalt beobachten. Eine Liste von Wikipedia, die bei weitem nicht alle Beispiele aufführt, mit solchen Spielen scheint diesen Trend zu bestätigen[1].

Zudem ist noch zu bemerken, dass diese Spiele auch von sehr vielen Nutzern von Steam, einer Videospielplattform mit ca. 9 bis 15 Millionen aktiven Nutzern (je nach Tageszeit)[2], gespielt werden. So hatte zum Beispiel „Terraria“, ein 2D Spiel, in dem man eine Welt erkundet, am 11.07.2018 einen Höchststand von 29.503 aktiven Spielern[3] und „Sid Meier’s Civilization V“ einen Höchststand von 29.284 aktiven Spielern[4]. Dies ist um einiges weniger als die 642.048 aktive Spieler von Valves Dota 2[5], aber immer noch ein beträchtlicher Marktanteil in der Branche.

Während sich mit guter Hardware für den PC weite Spielwelten, wie zum Beispiel in die „Sid Meier’s Civilization“ Spielreihe oder in „Ark: Survival Evolved“ ohne große CPU-Auslastung generieren lassen, beschränkt man sich bei Videospielen für Handys meist auf einzelne Level, wie zum Beispiel in „Hoplite“ oder in „Pixel Dungeon“, da CPUs für mobile Geräte oft nicht sehr leistungsstark sind.

Besondere Chancen ergeben sich natürlich für sogenannte „Indie-Studios“, kleine Videospielentwickler, da sie weniger Angestellte für die Spieleentwicklung benötigen, wenn sie die Welt beziehungsweise einzelne Level prozedural generieren, da mühsame Leveldesign Arbeit wegfällt. Zudem wird ein Spiel, bei dem man jedes Mal etwas Neues sieht wohl öfter gespielt als ein anderes. Diese zwei Faktoren ermöglichen es kleinen

[1] vgl. Wikipedia – List of games using procedural generation

[2] vgl. Steam – Steam: Game and Player Statistics

[3] ebd.

[4] ebd.

[5] ebd.

Spieleentwicklern oder gar Einzelpersonen[6] Videospiele zu entwickeln, die sich trotz der harten Konkurrenz gut verkaufen.

2. Definition von Labyrinth

Der Duden definiert ein Labyrinth unter anderem als „Anlage (als Teil eines Parks oder Gartens), deren verschlungene, zu einem Punkt in der Mitte der Anlage führende Wege von hohen Hecken gesäumt sind, sodass man sich darin verirren kann; Irrgarten"[7]. Diese Definition ist allerdings sehr vage und reicht im Rahmen dieser Arbeit nicht aus, weil die Labyrinthe, die behandelt werden sollen weder zu einem Punkt in deren Mitte führen, noch Teil eines Parks oder Gartens sind. Vielmehr sollen die besprochenen Labyrinthe folgende Eigenschaften besitzen: Sie sollen mehrere Verzweigungen beziehungsweise Weggabelungen, also Punkte, an denen mehrere Optionen zur Auswahl stehen, besitzen. Ein Labyrinth muss im Grundlegenden aber auch nicht aus Gängen mit Wänden bestehen, sondern kann auch eine sehr verzweigte Spielewelt bezeichnen.

Diese Gedanken zeigen sich auch in einem Essay von Clara Fernández-Vara. In dem englischen Text unterscheidet die Autorin zwischen den Begriffen „labyrinth" und „maze", obwohl diese im allgemeinen Sprachgebrauch als Synonyme gelten. Das erste bezeichnet laut ihrer Definition einen einzigen Weg ohne Verzweigungen, der sich um sich selbst schlängelt und somit potentielle Erkunder dessen nur verlangsamen beziehungsweise deren Weg verlängern.[8] Aus diesem Grund sind sie auch für Videospiele irrelevant und dementsprechend rar, da sie für den Spieler keine Herausforderung darstellen und ihn auf einen einzigen Weg leiten.[9]

Der Begriff „maze" hingegen bezeichnet eine spezielle Art Labyrinths, die komplexer ist als das Klassische. Sie bieten mehrere Wege mit Verzweigungen und Sackgassen[10], sodass man eine Wahl zwischen mehreren Möglichkeiten treffen muss. Diese Art wird für Videospiele

[6] Damian Schloter - slasherskeep.info/about.html

[7] Duden - Duden | La-by-rinth | Rechtschreibung, Bedeutung, Definition, Synonyme, Herkunft

[8] vgl. Fernández-Vara 2007, S. 74

[9] ebd.

[10] ebd.

favorisiert, da sie in sich selbst eine Erschwernis darstellen[11]. Da es für „maze“ keinen deutschen Ausdruck gibt, mit dem er sich vom klassischen Labyrinth abgrenzen lässt, stimmt der Begriff Labyrinth für diese Arbeit mit der Definition von „maze“ überein.

3. Prozedurale Generation von Labyrinthen anhand zwei Methoden

3.0 randomWall Methode

3.0.0 Arbeitsweise

Für die Erklärung werden einige Begriffe unabdingbar, diese werden nun erläutert:

Array: Ein Array speichert eine bestimmte Anzahl von Elementen eines bestimmten Typs[12], wie z.B. ganze Zahlen, Dezimalzahlen, Charaktere, oder ähnliches. Diese werden jeweils mit einem nullbasierten Index, d.h., dass die erste Stelle den Index 0 erhält, versehen. Die Anzahl sowie der Typ der gespeicherten Elemente können festgelegt werden.

Mehrdimensionales Array: Mehrdimensionale Arrays lassen sich in zwei Gruppen gliedern: jagged und rechteckig.[13] Für das Verständnis der Arbeit ist allerdings nur das rechteckige Array notwendig. Ein eindimensionales Array lässt sich wie eine Zeile einer Tabelle darstellen, bei zweidimensionalen Arrays wird diese Tabelle um Spalten erweitert. Arrays mit mehr Dimensionen sind auch möglich, so wären diese als Würfel beziehungsweise n-dimensionale Würfel darstellbar. Bei n-dimensionalen Arrays erhält jedes Element n nullbasierte Indexe.

Boolean: Boolean ist der Typ eines Wahrheitswertes. Das heißt, man kann mithilfe eines Booleans die Werte wahr und falsch beziehungsweise true und false.[14]

Ein zweidimensionales, rechteckiges Boolean-Array ist somit eine Repräsentation für mehrere Elemente des Typs Boolean, die je mit zwei nullbasierten Indexen versehen sind.

Liste: Eine Liste ist ähnlich einem Array. In der Programmiersprache C#, in der die Algorithmen für diese Seminararbeit zur Messung verschiedener Gütekriterien geschrieben sind, als Array,

[11] vgl. Fernández-Vara 2007, S. 74

[12] vgl. Albahari 2017, S. 34

[13] vgl. Albahari 2017, S. 36

[14] vgl. Albahari 2017, S. 28

„dessen Größe je nach Bedarf dynamisch erhöht wird.“[15] Dies bedeutet, dass deren Größe, also die Anzahl der gespeicherten Elemente, nicht festgelegt werden muss.

Diese Methode ist selbst entwickelt.

Bei der randomWall-Methode wird zuerst für jede mögliche Position einer Wand zufällig bestimmt, ob dort eine Wand sein sollte. Die Wahrscheinlichkeit für jede Position kann dabei gesetzt werden und wird im weiteren WallSpawnPercent genannt; welche Auswirkung diese Variable hat wird in 3.1.2 evident.

Danach wird überprüft, ob das Labyrinth lösbar ist, also von einer beliebigen Position im Labyrinth jede andere zugänglich ist. Dazu wird ein Startpunkt – für diese Arbeit wird ein Eckpunkt verwendet – festgelegt. In einem zweidimensionalen, rechteckigen Boolean-Array wird dieser Startpunkt, sowie alle Felder, die direkt an das des Startpunkts angrenzen und nicht durch eine Wand davon getrennt sind, mit dem Wahrheitswert true versehen. Zudem werden die für begehbar erklärten Felder mitgezählt. Ebendiese angrenzenden Felder werden in einer Liste gespeichert und danach als neue Startpunkte verwendet. Wenn neben einem Startpunkt keine angrenzenden Felder existieren, die nicht durch eine Wand abgegrenzt sind, und die Anzahl der begehbaren Felder kleiner ist als die der gesamten Felder, so muss eine Wand entfernt werden.

Dazu wird eine zufällige Wand neben dem letzten Startpunkt gewählt und überprüft, ob ein begehbares und ein nicht begehbares Feld an sie grenzt. Falls dies der Fall ist, wird sie entfernt und mit dem letzten Startpunkt erneut, wie oben beschrieben, fortgefahren. Falls nicht, wird ein neues Feld, das begehbar ist, zufällig ausgewählt und überprüft, ob eine Wand an ein begehbares und ein nicht begehbares Feld angrenzt. In diesem Fall wird diese Wand entfernt und das zufällig gewählte Feld wird als Startpunkt für die im oberen Abschnitt beschriebene Vorgehensweise verwendet.

[15] vgl. Microsoft - ArrayList Class (System.Collections) | Microsoft Docs

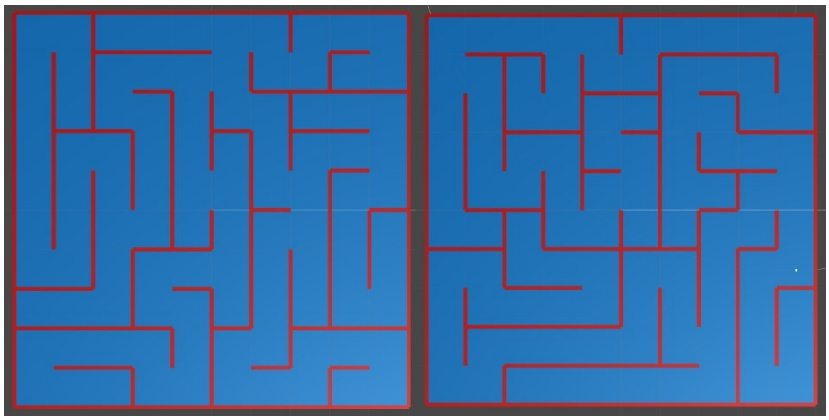

Abb. 1: Zwei mit der RandomWall-Methode generierte Labyrinthe

Wenn die Anzahl der begehbaren Felder gleich der Anzahl der gesamten Felder ist, ist ein lösbares Labyrinth (siehe Abb. 1), ohne ein und Ausgänge generiert worden.

3.0.1 Bewertung anhand verschiedener Gütekriterien

Ein Labyrinth kann anhand mehrerer Kriterien bewertet werden, was allerdings immer von einem Labyrinth, das wie in 2. definiert ist, gefordert ist, ist ein Grad der Verzweigung. Dieser kann anhand der Felder, die eine Verzweigung darstellen, gemessen werde. Ein Feld, das von zwei oder drei Wänden umgeben sind Teil eines Ganges, und stellen somit keine Verzweigung dar. Felder, die an eine Wand grenzen, gelten als eine Verzweigung und Felder, die an keine Wand grenzen als zwei Verzweigungen (siehe Abb.2).

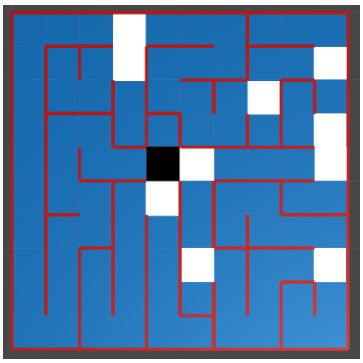

Abb. 2: Labyrinth mit Felder mit 1 Wand als weiß und Felder mit 0 Wänden schwarz

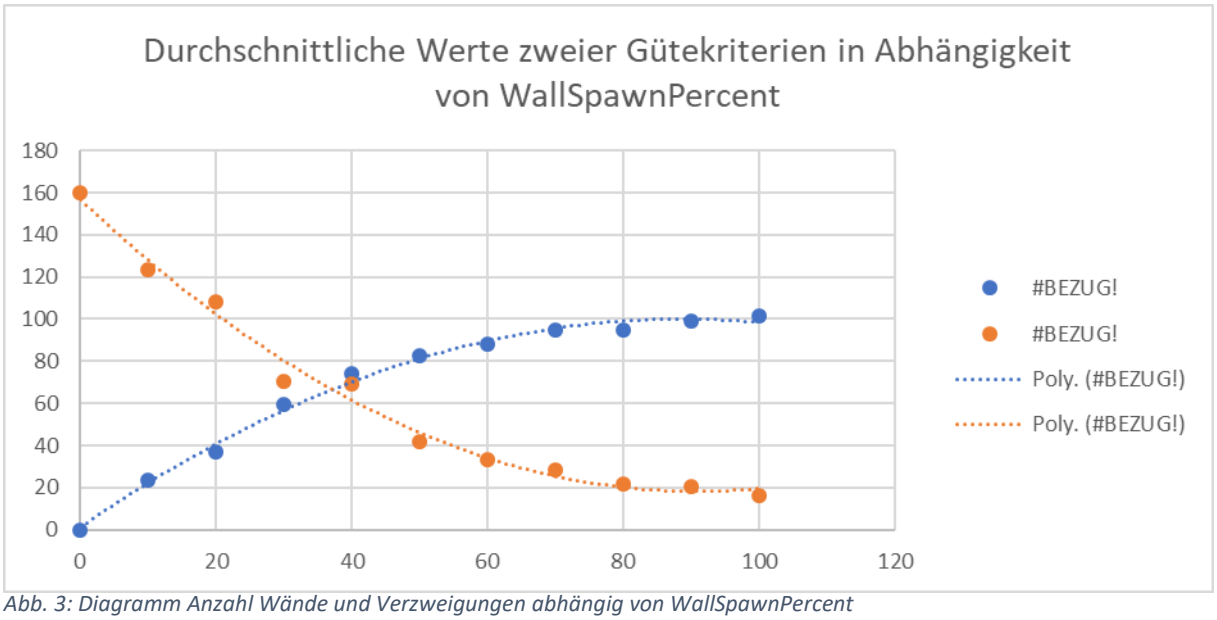

Abb. 3: Diagramm Anzahl Wände und Verzweigungen abhängig von WallSpawnPercent

Nur dieses Kriterium zu verwenden, ist nicht ausreichend, da bei einem niedrigen Wert für WallSpawnPercent die Anzahl solcher Felder steigt, dies aber nicht unbedingt mit schwereren Labyrinthen korreliert, da auch weniger Wände gesetzt werden (siehe Abb.3) und somit auch größere Räume entstehen können.

Aus diesem Grund möchte ich ein neues Kriterium benutzen, das aus den vorherigen hervorgeht, diese aber auch nicht gänzlich ersetzen kann. Es soll die Schwierigkeit von Labyrinthen gleicher Größe gemessen werden. Diese sollte mit der Anzahl der Verzweigungen steigen, aber auch mit der Anzahl der Wände, damit kein Labyrinth ohne Wände als komplex gilt. Es gilt:

$$Schwierigkeit = Anzahl\ Wände + \frac{Anzahl\ Verzweigungen}{Anzahl\ Wände}$$

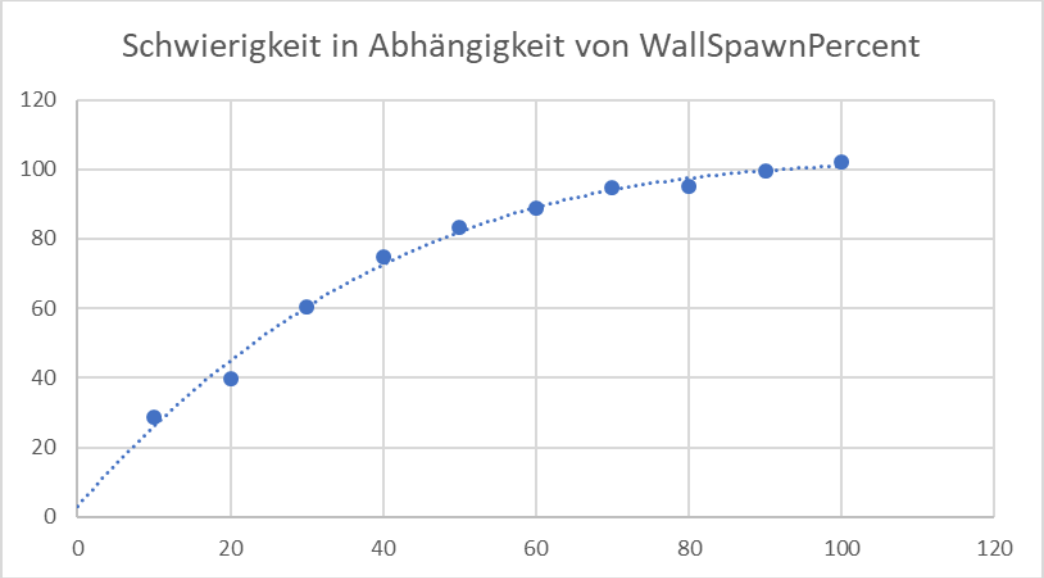

Abb. 4: Schwierigkeit in Abhängigkeit von WallSpawnPercent

Da die Kurve nahe 100 WallSpawnPercent abflacht (siehe Abb. 5), kann man als Alternative auch von vornherein alle Wände als true setzen, und dabei nahezu die gleiche Schwierigkeit erreichen. Dies spart Zeit, lohnt sich aber unter ca. 80 als Wert für WallSpawnPercent nicht mehr (siehe Abb. 6). Die Zeit, die für eine Ausführung des Algorithmus benötigt wird, wird Tick genannt. Diese Einheit entspricht 100 Nanosekunden und 10.000 Ticks sind somit eine Millisekunde.[16]

[16] Microsoft - DateTime.Ticks Property (System) | Microsoft Docs

Hier ist auch zu beobachten, dass die Variable WallSpawnPercent einen Einfluss auf die benötigte Zeit des Algorithmus hat (siehe Abb. 6). Je niedriger WallSpawnPercent, desto niedriger ist auch die benötigte Zeit.

Bis auf weiteres ist das Gütekriterium der Zeit nicht weiter zu bewerten, da kein Vergleich mit anderen Daten möglich ist. Es wird allerdings zum Vergleich mit der zweiten Methode in 3.3.1 wieder aufgegriffen.

WallSpawnPercent	Durchschnittliche Zeit in Ticks
100	5621.4
90	5626.2
80	5302.533333
70	4471
60	3422.2
50	2731.533333
40	2571.666667
30	2037.066667
20	1725.4
10	1676.333333
0	1038.4
ohne WallSpawnPercent	5335.733333

Abb. 5: Tabelle WallSpawnPercent und benötigte Zeit

3.1 Randomisierter Algorithmus von Prim

3.1.0 Arbeitsweise

Der Algorithmus von Prim beschäftigt sich mit der Graphentheorie, dem Teilgebiet der Mathematik, das die Eigenschaften von Graphen und deren Beziehungen zueinander untersucht.[17]

[17] vgl. Mathepedia – Graphentheorie - Mathepedia

Ein Graph „besteht aus Knoten (Ecken) […] und Kanten (Bögen) […], die jeweils zwei Knoten verbinden.“ (Puhl, 2007)[18]

Der Algorithmus von Prim wird angewendet, um alle Knoten eines Graphs mit Kanten so zu verbinden, dass die Summe der „Längen“ der Kanten am kleinstmöglichsten ist.[19]

Auch ein Labyrinth lässt sich als Graph darstellen, was allerdings nicht für das weitere Verständnis notwendig ist.

Der Algorithmus von Prim ist ein „gieriger“ Algorithmus, was bedeutet, dass immer die Option wählt, die mit den geringsten Kosten verbunden ist. Beim randomisierten Algorithmus von Prim wird dieser Teil allerdings nicht beachtet, da nicht die günstigste, sondern eine zufällige Option gewählt wird.

Für die Erläuterung der Methode, die für das Programmbeispiel in C#, welches der Arbeit beigelegt ist, werden zuerst einige Details geklärt:

Es wird ein **zweidimensionales, rechteckiges Boolean-Array** (Definition siehe 3.3.1) verwendet, dessen Elemente initial den Wahrheitswert false haben. Es repräsentiert alle Felder, die Teil des Labyrinths sind, und Elemente mit den Indexen, die gleich den Koordinaten dieser Felder sind, erhalten den Wert true.

Zudem wird eine **Liste** (Definition siehe 3.3.1) verwendet. Diese speichert in Form eines Arrays mit drei Elementen die „Koordinaten“ von Wänden, die Teil des Labyrinths sind. Dies sind Wände, die an ein Feld, welches im oben genannten Array den Wert true hat, angrenzen.

Des Weiteren wird ein **dreidimensionales rechteckiges Boolean-Array** genutzt. Hier korrespondieren die oben genannten „Koordinaten“ von Wänden mit den Indexen. Dieses Array dient dem Zweck der Vermeidung der Verdoppelung von Wänden in der bereits erwähnten Liste.

Beim randomisierten Algorithmus von Prim startet man mit allen möglichen Wänden als existent. Es wird ein zufälliges Feld als Startpunkt gewählt[20] und dessen Wert in dem oben erwähnten zweidimensionalen Boolean-Array auf true gesetzt. Danach werden die

[18] Gesine Puhl „Graphentheorie“ S.8

[19] vgl. Reza Sefidgar – Der Algorithmus von Prim

[20] Wikipedia – Maze generation algorithm - Wikipedia

Koordinaten aller Wände, die an das Startfeld grenzen, der besagten Liste hinzugefügt, sowie die Werte der Elemente, die mit den Wänden korrespondieren, in dem dreidimensionalen, rechteckigen Boolean-Array auf true gesetzt.

Zur Erweiterung des Labyrinths wird der folgende Absatz so lange wiederholt, bis die Anzahl der Felder, die Teil des Labyrinths sind, gleich der Anzahl der gesamten Felder ist.

Es wird eine zufällige Wand aus der Liste ausgewählt und überprüft, ob genau eins der zwei angrenzenden Felder Teil des Labyrinths und genau eins der Felder nicht Teil des Labyrinths ist. Falls dies nicht der Fall ist, wird erneut eine zufällige Wand ausgewählt und ebenso überprüft. Trifft es aber zu, so wird die Wand aus der Liste gelöscht. Das Element des zweidimensionalen Arrays, das mit dem Feld, das nicht Teil des Labyrinths war, korrespondiert, erhält den Wert true, wird also Teil des Labyrinths. Zudem werden die Wände, die an das vorgenannte Feld grenzen, der Liste hinzugefügt.[21]

Danach ist ein Labyrinth entstanden, dessen prozedurale Generation im Vergleich zur RandomWall-Methode (siehe 3.1) nicht durch eine Variable beeinflusst werden kann (siehe Abb. 6).

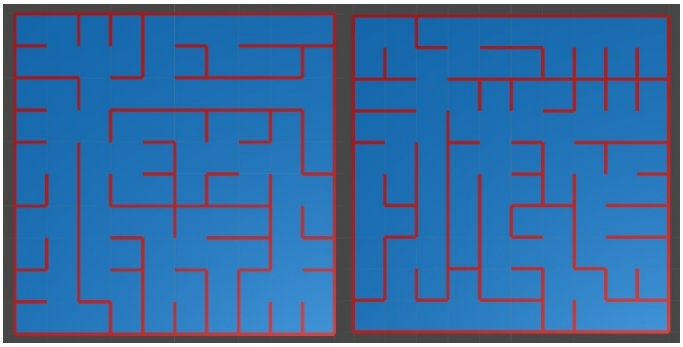

Abb. 6: Zwei mit dem Randomisierten Algorithmus von Prim generierte Labyrinthe

3.1.1 Bewertung anhand verschiedener Gütekriterien

Auch für diese Methode können die gleichen Gütekriterien, die in 3.1.2 verwendet wurden, genutzt werden. Auffällig ist hier, dass die Anzahl der Wände konstant 81 beträgt.

Die Erklärung für alle Begriffe der Tabelle (siehe Abb. 7) befindet sich in 3.1.2.

[21] ebd.

Sonstiges wird im Vergleich in 3.3 behandelt.

Schwierigkeit	Anzahl Verzweigungen	Durchschnitt Wände	Durchschnittliche Zeit in Ticks
81.38888889	31.5	81	2279

Abb. 7: Verschiedene Gütekriterien des Randomisierten Algorithmus von Prim

3.2 Vergleich der Methoden

3.2.0 Daten des Mess-PCs

Zum Vergleich von Daten, insbesondere der Zeit, ist unter anderem die Hardware des PCs zu berücksichtigen. Die relevanten Teile sind in diesem Fall CPU, bzw. der Prozessor, und GPU, bzw. die Grafikkarte, sowie RAM, in der Umgangssprache Arbeitsspeicher.

Diese sind:

CPU: Intel Core i5-7600K CPU 3.80GHz

Grafikkarte/GPU: NVIDIA GeForce GTX 960

RAM: verschiedene Hersteller insgesamt 12GB und eine Frequenz von 802 MHz

Auch das Betriebssystem ist nicht irrelevant. Hier wurde 64-Bit Windows 10 Pro verwendet. Die Version ist 1803.

3.2.1 Vergleich anhand des Gütekriteriums der Zeit

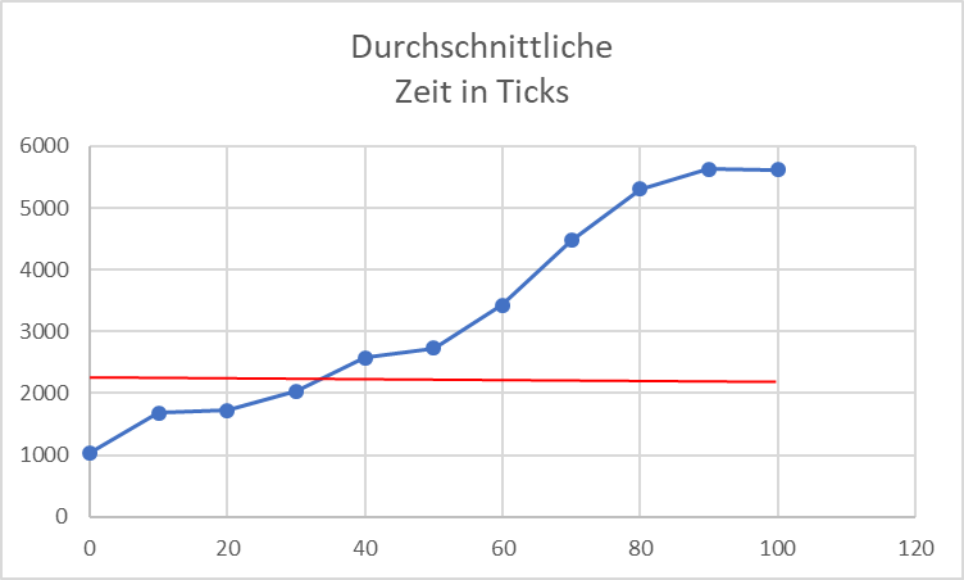

Abb. 8: Diagramm Durchschnittliche Zeit in Ticks RandomWall-Methode: blau; Randomisierter Algorithmus von Prim: rot

Wie in Abb. 7 zu sehen ist, benötigt die RandomWall-Methode bei einem Wert von 0 bis ca. 35 für WallSpawnPercent weniger Zeit. Die maximale Differenz ist 1241 Ticks bei WallSpawnPercent gleich 0, was 0,1241 Millisekunden entspricht. Danach benötigt der Randomisierte Algorithmus von Prim weniger Zeit als die RandomWall-Methode. Die maximale Differenz beträgt hier 3347 Ticks bei WallSpawnPercent gleich 100, was 0,3347 Millisekunden entspricht. Dies ist darauf zurückzuführen, dass der Randomisierte Algorithmus von Prim im Gegensatz zur RandomWall-Methode nicht durch eine Variable beeinflusst werden kann aber damit auch immer ähnliche Ergebnisse liefert. Die Zeiten müssen somit ferner so verglichen werden, dass die Schwierigkeit übereinstimmt. Die Schwierigkeit des Randomisierten Algorithmus von Prim entspricht ungefähr der Schwierigkeit der RandomWall-Methode bei einem WallSpawnPercentWert von ca. 50 (siehe 3.3.2). Wenn man die Zeiten bei diesem Wert vergleicht, so benötigt der Randomisierte Algorithmus von Prim 452,5 Ticks, was 0,04525 Millisekunden entspricht, weniger Zeit als die andere Methode. Dieser Unterschied ist minimal, könnte sich aber mit langsamerer Hardware vergrößern.

Man sollte also, falls man einen Wert von ca. 81 für die Schwierigkeit anpeilt den Randomisierten Algorithmus von Prim verwenden.

3.2.2 Vergleich anhand der Gütekriterien der Anzahl der Wände und der Verzweigungen sowie der Schwierigkeit

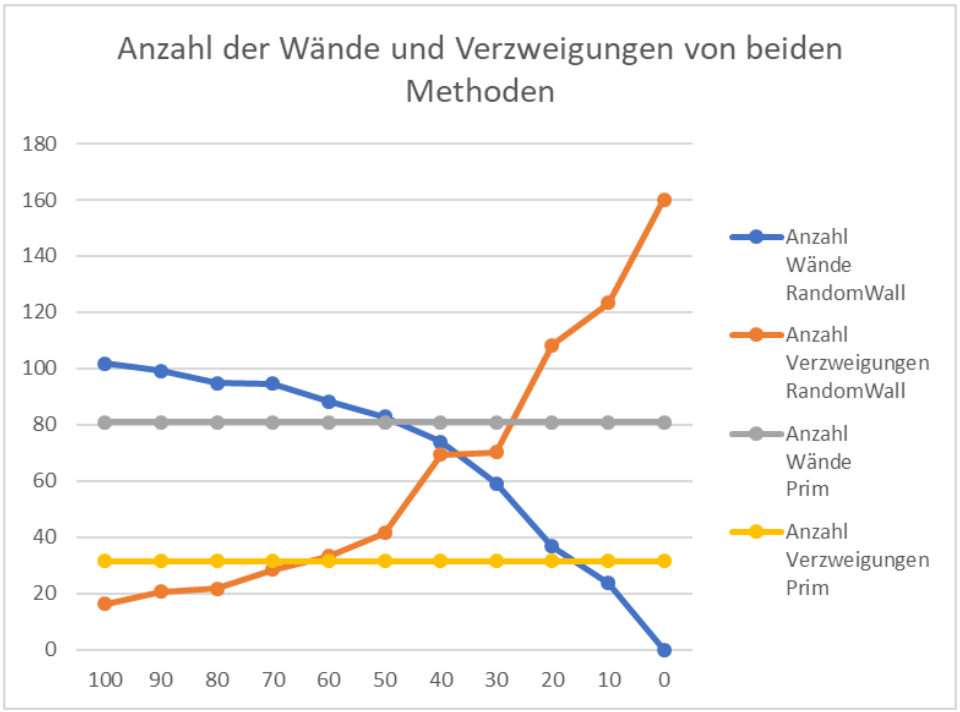

Abb. 9: Diagramm: Anzahl der Wände und Verzweigungen beider Methoden

Auch in dem obigen Diagramm (Abb.8) sind die Werte des Randomisierten Algorithmus von Prim als geraden eingetragen, da dieser nicht durch WallSpawnPercent beeinflusst wird.

Hier ist auch zu beachten, dass die Anzahl der Verzweigungen der RandomWall-Methode bei niedrigen Werten für WallSpawnPercent so hoch ist, weil auch Felder, die an keine Wand grenzen, aber an andere Felder ohne Wand grenzen, also einen Raum bilden, gezählt werden. Dies beeinträchtigt den Vergleich nicht, da die Schnittpunkte bei höheren Werten sind.

Bis zu einem Wert von 65 für WallSpawnPercent hat der Randomisierte Algorithmus von Prim weniger Verzweigungen als die RandomWall-Methode, welche bis zu ca. 50 weniger Wände generiert. Das auffällige daran ist, dass diese Schnittpunkte nicht bei einem Wert für WallSpawnPercent beisammen liegen, was bedeutet, dass man auf alle Fälle andere Labyrinthe generiert, als dies bei der jeweils anderen Methode der Fall ist.

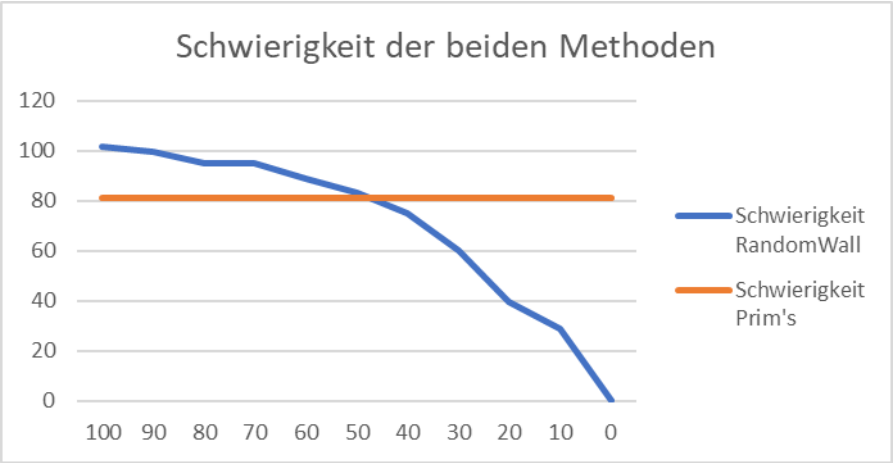

Abb. 10:Diagramm: Schwierigkeit beider Methoden

Bei der Schwierigkeit erhält man wieder einen anderen Schnittpunkt der Werte der beiden Methoden. Die Schwierigkeit des Randomisierten Algorithmus von Prim hat den Wert 81,4 und schneidet den Graphen der Schwierigkeit der RandomWall-Methode bei einem Wert von ca. 50 für WallSpawnPercent. Dies bedeutet, bei diesem Wert erreicht man ähnliche Labyrinthe, aber eben nicht gleiche (siehe Abb.8).

Es ist evident, dass die RandomWall-Methode den Vorteil der Variabilität hat. Das heißt man sollte sie bevorzugen, wenn man die generierten Labyrinthe variieren möchte.

3.2.3 Fazit des Vergleichs

Beide Methoden haben gewisse Vorteile gegenüber der anderen, es muss also basierend auf den Verwendungszweck eine gewählt werden. Sollte Geschwindigkeit im Vordergrund stehen, beziehungsweise Variabilität nicht wichtig sein, so sollte man den Randomisierten Algorithmus von Prim verwenden. Ein Beispiel hierfür wäre ein Videospiel für mobile Geräte, die oft im Vergleich zu Notebooks oder PCs eher langsame Hardware besitzen. Aber auch, wenn man viele Labyrinthe oder sehr große Labyrinthe generieren möchte ist diese Methode zu bevorzugen, da sich bei solchen Anwendungen der Zeitunterschied vergrößern wird und somit einen größeren Einfluss haben wird.

Setzt man hingegen eher auf Variabilität, so ist die RandomWall-Methode zu wählen, da sie diese bietet. Ebenso sollte man diese wählen, wenn man schwerere Labyrinthe generieren möchte, da diese bei der RandomWall-Methode erhöht werden kann. Zudem sollte sie für größere Labyrinthe nur auf Computern verwendet werden, da sie auf Systemen mit weniger starker Hardware höchstwahrscheinlich eine beträchtliche Zeit benötigen würde.

Abbildungsverzeichnis

Quellen der Abbildungen:

Alle Abbildungen: eigene Abbildungen

Literaturverzeichnis

Buchquellen:

Albahari, J. A. (2017). *C# 7.0 Pocket Reference.* O'Reilly.

Fernández-Vara, C. (2007). Labyrinth and Maze. In F. von Borries, S. P. Walz, & M. Böttger, *Space Time Play Computer Games, Architecture and Urbanism: The next Level* (S. 74-77). birkhauser.

Internetquellen:

Duden. (kein Datum). *Duden | La-by-rinth | Rechtschreibung, Bedeutung, Definition, Synonyme, Herkunft*. Abgerufen am 30. 10 2018 von https://www.duden.de/rechtschreibung/Labyrinth

Graphentheorie - Mathepedia.. Abgerufen am 01. 11 2018 von https://mathepedia.de/Graphentheorie.html

List of games using procedural generation - Wikipedia. (11. 07 2018). Von Wikipedia: https://en.wikipedia.org/wiki/List_of_games_using_procedural_generation abgerufen

Microsoft. *ArrayList Class (System.Collections) | Microsoft Docs*. Abgerufen am 02. 11 2018 von https://docs.microsoft.com/de-de/dotnet/api/system.collections.arraylist?view=netframework-4.7.2

Microsoft. *DateTime.Ticks Property (System) | Microsoft Docs*. Abgerufen am 02. 11 2018 von https://docs.microsoft.com/de-de/dotnet/api/system.datetime.ticks?view=netframework-4.7.2#System_DateTime_Ticks

Puhl, G. (14. 10 2007). *Graphentheorie.* Abgerufen am 01. 11 2018 von TU Berlin: Technische Universität Berlin: http://page.math.tu-berlin.de/~felsner/Lehre/GrTh05/GT.pdf

Schloter, D. *slasherskeep.info/about.html*. Abgerufen am 11. 07 2018 von www.slasherskeep.info/about.html

Sefidgar, R. (2014). *Der Algorithmus von Prim*. Abgerufen am 01. 11 2018 von https://www-m9.ma.tum.de/graph-algorithms/mst-prim/index_de.html

Steam. (11. 07 2018). *Steam: Game and Player Statistics*. Von store.steampowered.com/stats/ abgerufen

Wikipedia. (30. 10 2018). *Maze generation algorithm - Wikipedia*. Von https://en.wikipedia.org/wiki/Maze_generation_algorithm abgerufen