LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Aspekte der systemnahen Programmierung bei der Spieleentwicklung

Gruppe 132 – Abgabe zu Aufgabe A214
Wintersemester 2020/21

Mohammed Attia Thomas Torggler Patrick Zimmermann

1 Einleitung

Raumfüllende Kurven bilden eine Brücke zwischen Kunst und mathematischer Geometrie. In der Mathematik werden sie gemeinhin benutzt um ein n-dimensionales Problem in ein Eindimensionales zu konvertieren. Eine solche Kurve beschreibt grundsätzlich einen linearen Pfad durch n-dimensionale Räume. Giuseppe Peano war der Erste, der eine solche Kurve 1890 definierte.

Um einen n-dimensionalen Raum in die Dimension n-1 zu konvertieren, lässt sich eine stetig, surjektive Funktion $f(x)$ erstellen, so dass gilt: $\forall x \in \mathbb{R}^{n-1} \quad \exists y \in \mathbb{R}^n$. Für einen Beweis der Stetigkeit einer Peano-Kurve siehe [3], für einen Beweis der Surjektivität solcher Funktionen und tiefere mathematische Definitionen sei auf [6] verwiesen. Hier wollen wir uns auf die sogenannten Peano-Kurven beschränken. Für eine solche Kurve definieren wir ein Intervall $I = [0; 1]$, sowie $f : I \to I^2$. Dann ist die Peano-Kurve: $\lim_{n\to\infty} f_n(x)$, mit $n \in \mathbb{N}$. Sie entspricht also dem Grenzwert einer Folge von Funktionen $f(x)$ und lässt sich mit der Bedingung, dass sich die Kurve nicht überschneiden darf, folgendermaßen konstruieren:

Man unterteile eine Fläche in 9 Quadrate. Jedes dieser Quadrate soll nun von einer Kurve in Form eines 'S' durchlaufen werden. In einem Iterationsschritt lässt sich eines der 9 Quadrate in weitere 9 Quadrate Unterteilen, die wiederum auf dieselbe Art verbunden werden, wie in Abbildung 1 gezeigt.

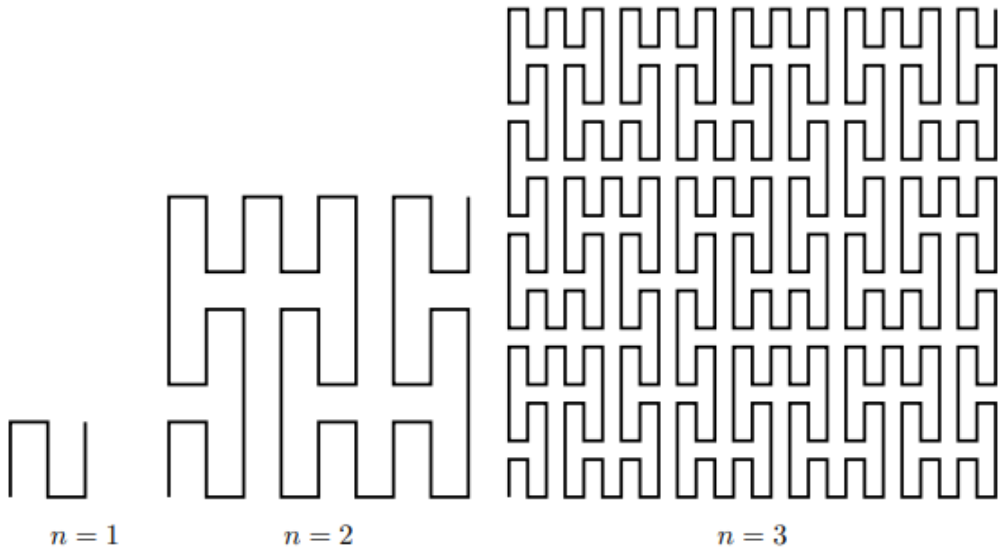

Abbildung 1: Peano-Kurve mit $n = \{1, 2, 3\}$, [2]

In der Realität sind gesammelte Daten meistens nicht linear, nicht stationär und mehrdimensional. Beispiele für solche Daten sind Matrizen, Bilder, Tabellen und Rechengitter, die sich aus der Diskretisierung von partiellen Differentialgleichungen ergeben. Dateioperationen wie Matrixmultiplikationen, Lade- und Speicheroperationen sowie das

Aktualisieren und Partitionieren von Datensätzen können vereinfacht werden, wenn eine effiziente Methode zum Durchsuchen der Daten gewählt wird. Die Verwendung dieser komplexen Daten kann zeitlich sehr teuer sein und erhöhen die Speicherverwendung exponentiell. Deshalb werden Kompressions-Algorithmen gebraucht, um unsere Programme zu optimieren. In solchen Fällen bietet sich die Nutzung der Peano-Kurve oder anderer raumfüllender Kurven an. Eine Transformation eines n-dimensionalen Raumes in die Dimension n-1 führt zu einer wesentlichen Komprimierung der Informationen und behält einen Teil der räumlichen Daten bei. Eine mögliche Anwendung dieses Konzepts ist das Feld des Image Processings bei dem raumfüllende Kurven für Bild- und Farbkompressionen verwendet werden [5].

Der Nachteil der vorgeschlagenen Struktur liegt in ihrer relativen Komplexität. Beispielsweise ist in unserer Implementierung bei Grad $n > 9$ die Speicherallokation nicht mehr möglich, wie später in Kapitel 3.2 erläutert wird.

Im Folgenden definieren wir $n \in \mathbb{N}$ als Grad der Kurve. Wir beschreiben nun unseren Ansatz, einen iterativen Algorithmus mit dem Grad n als Eingabe zu finden, um die eben beschriebene Peano-Kurve darzustellen. Die hierbei generierten Punkte werden in ihrer Reihenfolge ausgegeben. Anders als in der beschriebenen mathematischen Definition werden wir jedoch nicht nur nach $[0; 1]$, sondern nach $\mathbb{N}$ abbilden. Die Korrektheit des hier gewählten Ansatzes wird gezeigt, in C und in Assembler implementiert und hinsichtlich ihrer Performanz analysiert. Dabei ist zu erkennen, dass eine rekursive Implementierung wahrscheinlich bessere Ergebnisse erzielt hätte und eine Optimierung mit SIMD-Instruktionen die Performanz nur bedingt erhöht.

2 Lösungsansatz

2.1 Algorithmus

Der Aufbau der Peano-Kurve ermöglicht es, die Kurve des aktuellen Grades mit einer Variation der originalen Kurve des vorherigen Grades und deren Permutationen zu zeichnen. Wie in Abbildung 2 gezeigt, gibt es ingesamt drei Permutationen der Kurve:

- Die Invertierung, bei der jeder Schritt auf seinen entsprechenden Komplement abgebildet wird.
- Die Spiegelung, bei der nur die vertikalen Schritte auf ihren Komplement abgebildet werden.
- Die gespiegelte Invertierung, bei der die vorherigen Permuationen kombiniert werden.

Diese Observation war der Grundstein des hier beschriebenen iterativen Algorithmus. Anfangs wird die Startkurve hartcodiert in einem Integerarray mit Werten im Interval $[0; 3]$ abgespeichert, wobei jede Zahl eine Richtungen repräsentiert: 0 für oben, 1 für rechts, 2 für unten und 3 für links. Die orignale Kurve des ersten Grades und deren Invertierung sehen wie folgt aus: $\{0, 0, 1, 2, 2, 1, 0, 0\}$ bzw. $\{2, 2, 3, 0, 0, 3, 2, 2\}$. Falls $n > 1$

ist, wird ausgehend von Grad 2 aufsteigend über alle Grade $m \leq n$ iteriert. Bei jeder Iteration wird abwechselnd ein Verbindungschritt entsprechend eines Schritts entlang der Kurve $n = 1$ hartcodiert in den Array geschrieben und danach die entsprechende Permutation der Kurve des vorherigen Grades $m - 1$ berechnet und eingefügt. Dieser Vorgang wird für die acht fehlenden Teilkurven des aktuellen Grades m ausgeführt, bis die vollständige Kurve entsteht.

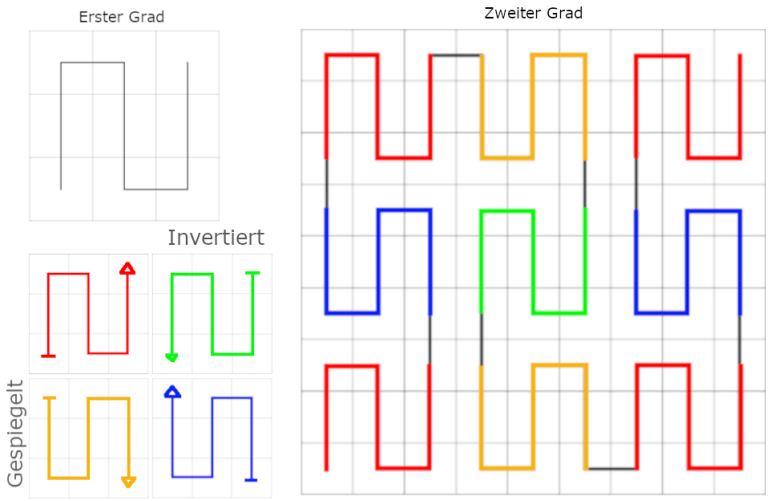

Abbildung 2: Peano-Kurve mit $n = 1$, $n = 2$

Nachdem über alle Grade iteriert wurde, läuft der Algorithmus den vollständigen Richtungsarray ab, verändert dabei bei jedem Schritt je nach Richtungsangabe entweder die X oder Y-Koordinate und speichert diese dann in die Ausgabeliste der Peano-Methode.

2.2 Umgesetzte Optimierungen

Im ursprünglichen Entwurf des Algorithmus wurden die drei Permutationen vor jeder Iteration einmal berechnet und als einzelne Arrays abgespeichert, um sie danach, wenn sie benötigt werden, nur noch an die jeweiligen Stellen des Richtungsarrays zu kopieren. Da dadurch ein erhöhter Speicheraufwand entsteht, haben wir uns dafür entschieden jede Permutation immer neu zu berechnen, wenn sie gebraucht wird und diese direkt in den Array zu speichern. Dies schien ein sinnvoller Optimierungsansatz zu sein, da jede Permutation, wie in Abbildung 2 zu sehen, maximal zweimal vorkommt. Allein die Originalkurve wird dreimal benötigt und auf diese kann sehr schnell zugegriffen werden, indem man auf die Werte an den Adressen von 0 bis $9^{(m-1)} - 2$ im Richtungsarray zugreift.

Bei der Assemblerimplementierung beschränken wir uns auf Instruktionen der x86-64-Architektur bis einschließlich der SSE4.2-Erweiterung [4]. Hier haben wir beide Funktionen, die die invertierte Permutation erstellt sowie die Funktion, die die Orginalkurve

an die benötigte Stelle des Arrays schreibt, mit SIMD-Befehlen optimiert.

2.3 Alternativer Lösungsansatz

Abgesehen von den eben beschriebenen Optimierungen wurde weitgehend bewusst auf die Nutzung von SIMD-Befehlen verzichtet. Der einzige noch gewinnbringende Anwendungspunkt wäre die parallele Umrechnung der Richtungen zu den Koordinaten. Die Richtungsberechnung lässt sich im Rahmen dieser Arbeit auch nicht mehr gewinnbringend mit SIMD optimieren, da die Funktionen, die weitere Permutationen generieren, in jedem Berechnungsschritt eine Fallunterscheidung beinhaltet, weshalb hier SIMD nicht zu erheblichen Performanzgewinnen führen würde.

Da wir 64-Bit große Koordinatenstellen benutzen, könnten wir damit generell maximal zwei Koordinaten gleichzeitig verrechnen. Da die Koordinatenberechnung allerdings aufeinander aufbaut, wurde sie hier nicht mit SIMD optimiert. Hinzu kommt, dass die Kurve nach zwei Schritten die Richtung ändert und man dieselbe Berechnung nicht mehr parallel ausführen kann.

Dennoch würde sich der Algorithmus in einem alternativen Lösungsansatz zusätzlich mit Hilfe von SIMD-Instruktionen weiter optimieren lassen.

Alternativ haben wir gegen Ende der Bearbeitungszeit eine Möglichkeit gefunden, den Richtungsarray auszulassen und die gesamte Koordinatenberechnung direkt durchzuführen.

3 Korrektheit

Die Genauigkeit unseres Algorithmus ist abhängig von dem Grad n, der die Anzahl an Iterationen bestimmt, wie in Abbildung 1 beschrieben. Das geht auch aus der beschriebenen mathematischen Definition hervor, da die Peano-Kurve an sich ein Grenzwert ist. Somit wird der Algorithmus präziser, je größer der Grad der Kurve ist, da die Punkte auf der Kurve sich ebenfalls einem Grenzwert annähern. Ansonsten ist die Kurve wie oben beschrieben, definiert, weshalb die Korrektheit unseres Algorithmus entscheidend ist. Grundsätzlich lässt sich die Korrektheit einer Kurve, besonders bei der von uns behandelten, nachweisen, indem man ihre graphischen Darstellungen vergleicht. Da das theoretisch aber nicht möglich ist, folgt auch eine Implementierung in Assembler und in C, die wir auch hinsichtlich ihrer Performanz analysieren wollen. Dennoch soll der bereits beschriebene Ansatz auf Korrektheit geprüft werden.

Da die eigentliche Koordinatenberechnung abhängig von der zuvor ermittelten Richtung ist, ist der folgende Beweis primär auf die Berechnung der Richtungen ausgerichtet. Die Koordinatenberechnung findet anschließend und basierend auf den bisherigen Ergebnissen statt, indem abängig von der Richtung jeweils die X- und die Y-Koordinate eines Zählers inkrementiert bzw. dekrementiert und abgespeichert werden.

3.1 Beweis der Permutationen durch Induktion

Nachdem die Kurve immer wieder die gleichen, teilweise elementaren Bestandteile verwendet, müssen diese und deren Permutationen richtig berechnet werden. Im Speziellen sind das die Kurven mit $n = 1$ und $n = 2$. Aus diesen lässt sich induktiv die Korrektheit beweisen, da wir, wie im vorherigen Kapitel 2.1 beschrieben, zum Berechnen einer Kurve n-ten Grades nur die Kurve des Grads $n - 1$ sowie die gespiegelte und invertierte Version dieser Kurve benutzen.

3.1.1 Induktionsbasis

Beginnen wir mit $n = 1$. Hier ist die Kurve vordefiniert, dementsprechend gibt es nichts zu zeigen. Da auf diesem Muster alle anderen Kurven mit $n > 1$ basieren, ist sie in unserem Algorithmus ebenfalls als Basis vorgegeben. Des Weiteren ist die Definition der Kurve mit $n = 2$ für die Berechnung aller weiteren Kurven nötig, da man anhand dieser Kurve alle notwendigen Bedingungen der Konstruktion weiterer Kurven ableiten kann. Deshalb nehmen wir sie also auch als definiert an 1. Diese Bedingungen sollen im folgenden ebenfalls geprüft werden.

3.1.2 Induktionsannahme

Da die Funktion wie vorher definiert surjektiv und stetig ist, gilt:

$$\forall n > 1 \in \mathbb{N} \text{ und } \forall y \in \mathbb{R}^n \text{ gilt: } \exists x \in [0, 1] \text{ so, dass } f(x) = y$$

Dadurch lassen sich Kurven mit höherem Grad konstruieren. Es sei nun $n > 1$ und n beliebig gewählt.

3.1.3 Induktionsschritt

Betrachten wir nun Kurven n-ten Grades. Hierzu müssen wir die Permutationen der vorhergehenden Kurve untersuchen, die wir zur Konstruktion der Kurve mit Grad n benötigen. Da diese Permutationen stets auf ähnliche Art konstruiert werden, nämlich durch Addition von 2 zu der gespeicherten Richtung mit anschließendem *modulo* 4, um die Richtung wieder in den gültigen Bereich abzubilden, ist gewehrleistet, dass durch Mehrfachanwendung einer Funktion, die eine solche Permutation generiert, immer nur die Orginale Kurve oder die gewünschte Permutation entstehen kann. Dies wird durch den Wertebereich von $[0; 3]$ und *modulo* 4 erreicht, da so die geraden Werte auf den jeweils anderen geraden Wert abgebildet werden, für die ungerade Zahlen analog. Dieser Umstand wird bei dem Erstellen der Permutationen ausgenutzt. Da die Funktionen, die die Permutationen erzeugen, auf die gesamte Kurve des Grades $n = n-1$ angewand wird, ist sichergestellt, dass die Permutationen keine anderweitigen Fehler in der Richtungsberechnung besitzten. Ergebnis ist in Abbildung 2 dargestellt.

Die Permutationen: Die invertierte Permutation ist im Kern nichts anderes als die normale Peano-Kurve mit anderer Reihenfolge der Koordinatenberechnung, im Speziellen

von $p_1 = (1, 1)$ nach $p_2 = (0, 0)$. Dies hat zur Folge, dass sich außer der invertierten Reihenfolge der Punkte an der grundlegenden Kurve nichts ändert. Somit ist die invertierte Permutation induktiv eben so korrekt wie die zugrundeliegende Kurve niedrigeren Grades, siehe auch Abbildung 2, grüne Kurve.

Bei der Spiegelung wird die Kurve an der X-Achse gespiegelt. Wir verändern also die vertikale Richtung, behalten jedoch weiterhin die horizontale Richtung bei. Dadurch wird die vorausgehende Kurve notwendigerweise abgeändert. Diese zusätzliche Operation ist elementar und greift auf die gespeicherte Richtung einer Koordinate direkt zu. Somit hat diese Operation nur indirekt auf die Koordinatenberechnung Einfluss, ohne die Kurve anderweitig zu verändern. Siehe auch Abbildung 2, gelbe Kurve.

Abschließend bleibt noch die Kombination aus den vorangehenden Permutationen, die ebenfalls für die Konstruktion der Kurve notwendig ist. Diese Permutation ist, ähnlich zu der vorherigen Permutation, eine Spiegelung an der Y-Achse. Dabei wird die horizontale Richtung gewechselt und die Vertikale beibehalten. Ansonsten verhält es sich analog zur Spiegelung. Siehe auch hier Abbildung 2, blaue Kurve.

Verbinden der Permutationen: Die unterschiedlichen Permutationen der Peano-Kurve werden durch eine ebensolche Kurve mit $n = 1$ verbunden. Dadurch bleiben alle Eigenschaften der Kurve erhalten, da wir keine weiteren Punkte in die Kurve einfügen, sondern nur die Art der graphischen Darstellung verändern. Wir fügen also einen weiteren hartcodiertern Schritt zu den bisherigen Richtungen hinzu, um die Koordinaten später korrekt zu berechnen. Dies ist zudem Teil der Definition dieser Kurve.

3.2 Technische Limitationen

Bei dem Testen unseres Programms fallen bei Graden von $n > 6$ bestimmte technische Limitationen unser Rechensysteme auf.

Zum einen haben wir kein Programm gefunden, dass die generierten .svg Dateien von einer Größe 21,8 Megabyte (Peano-Kurve mit $n = 7$) öffnen kann. Mit zunehmendem Grad enthält die .svg Datei logischerweise immer mehr Punkte der Kurve, weshalb die Größe ebenfalls exponentiell zur Basis 9 steigt. Deshalb möchten wir an dieser Stelle auf das Kapitel 3 verweisen, um die Kurve auf selbige zu prüfen.

Zum anderen ist es nicht möglich, beliebig viel Speicher zu allokieren, da die von uns benutzte *malloc*-Funktion maximal 17.179.869.184 Bytes auf dem Heap unser Testumgebungen allokieren kann und der Heap zusätzlich in seiner Größe auf Seite der Hardware begrenzt ist. Deshalb können wir auf unseren Systemen die Funktion mit maximal $n = 9$ ausführen, da sonst nicht genügend Speicher allokiert wird. Für $n = 10$ müssten 27.894.275.208 Bytes allokiert werden, wozu wir nicht im Stande sind.

Um zu ermitteln, wie viel Speicher wir allokieren können, haben wir eine For-Schleife folgenden Typs ausgeführt:

```
u_int64_t *v;
for (u_int64_t i = 1; v = (u_int64_t *) malloc(i); i <<= 1)
{   print(i);   free(v);  }
```

Zusätzlich evaluierte ein Test, der überprüft, ob der von *malloc* zurückgegebenen Pointer bei $n > 9$ NULL ist, auf unseren Testplatformen ausnahmslos zu *true*.

Deshalb ist der Wertebereich von n in den Tests im nachfolgenden Kapitel auf $n \in [1; 9]$ limitiert.

Als ein Beispiel der erstellten .svg Datei für diesen Algortihmus sei auf Abbildung 3 verwiesen, welche die erstellte Kurve mit $n = 4$ zeigt. Für $n < 4$ werden Bilder erzeugt, die den Kurven in Abbildung 1 entsprechen. Für $n > 4$ ist die Kurve aufgrund der hohen Dichte in diesem Dokument schwer lesbar darstellbar.

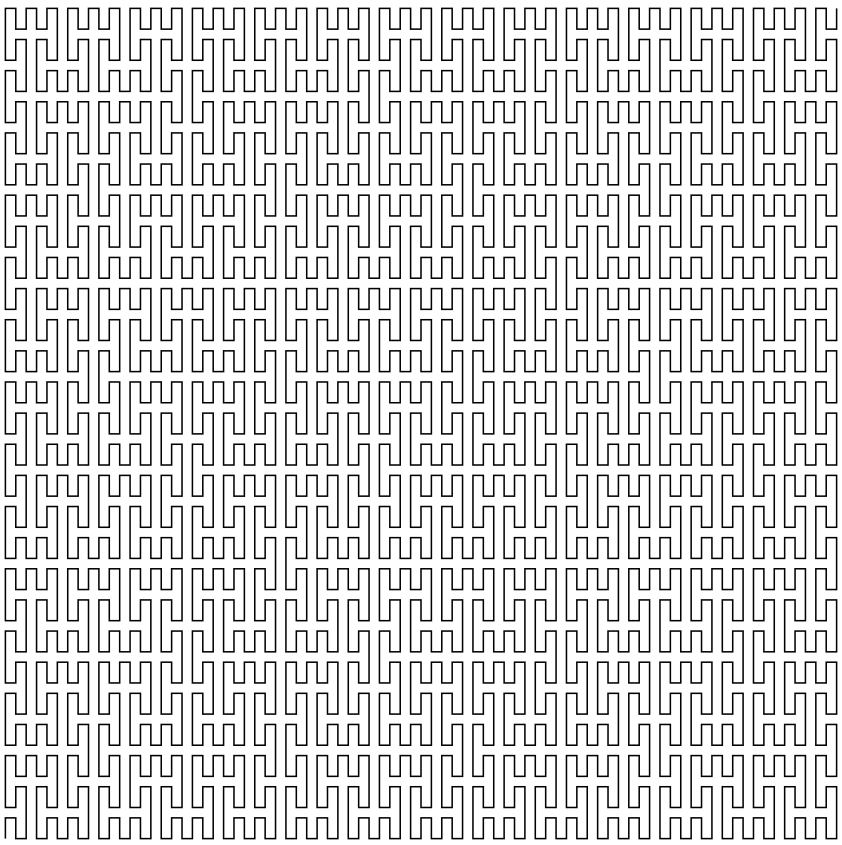

Abbildung 3: Generierte Peano-Kurve mit $n = 4$

4 Performanzanalyse

Bevor der bereits ausgeführten Algorithmus hinsichtlich seiner Performanz analysiert wird, werden kurz die Algorithmen die verglichen werden, beschrieben.

Als erstes wurde der bereits erläuterten Algorithmus aus Kapitel 2 in C implementiert. Um die Performanz-Unterschiede zwischen der direkten Berechnung, im Weiteren als In-Place-Variante, und der einmaligen Berechnung und Speicherung der Permutationen,

im Folgenden als Out-Of-Place-Variante bezeichnet, zu analysieren erweitern wir die Implementierung um einen entsprechenden Algorithmus, welcher eine abgewandelte Form der In-Place-Variante ist. Der Unterschied besteht darin, dass sobald wir eine Kurve mit $n > 2$ berechnen, speichern wir die schon vorher besprochenen Permutationen der Kurve mit Grad $n - 1$.

Da wir uns Eingangs auf einen iterativen Ansatz festgelegt haben, ist es zudem interessant zu sehen, inwiefern sich der Unterschied zwischen einem iterativen und einem rekursiven Algorithmus auswirkt. Deshalb wurde auch ein Algorithmus implementiert, der rekursiv die Richtungen berechnet. Aus Zeitgründen können wir jedoch die Implementierungen nur in C vergleichen.

4.1 Dokumentation

Die Laufzeittests der verschiedenen Algorithmen wurden auf einem System mit einem Intel Core i7-7700K Prozessor mit einer Taktfrequenz von 4.20GHz, 16 GB Arbeitsspeicher und auf einem Linux Mint 20 64-bit Betriebsssystem mit Linux Kernel 5.4.0-26-generic durchgeführt. Kompiliert wurde dabei mit GCC 9.3.0 mit Option -03. Wie bereits im Kapitel 3.2 beschrieben, war es nur möglich bis Grad 9 zu testen. Für jeden Algorithmus wurden für jeden Grad 20 Durchläufe aufgezeichnet und deren Durchschnitt berechnet.

In Abbildung 4 wird der prozentuelle Zeitunterschied zwischen der SIMD-Optimierten Assemblerimplementierung und den Vergleichsalgorithmen dokumentiert. Hier ist zu erkennen, dass SIMD-Befehle bei den Graden eins bis drei zu deutlichen Zeitverbesserungen von bis zu über 28% führen, jedoch bei den Graden fünf bis sieben kleinere Zeiteinbusen mit sich bringen. Bei den letzten zwei Graden acht und neun verbessert die Nutzung von SIMD-Befehlen den Algorithmus wieder, jedoch schafft er es nicht die Zeiten des rekursiven Algorithmus zu schlagen. Damit kann man leider nicht vorhersehen welcher Algorithmus sich für die Grade $n > 9$ am besten eignen würde oder ob die Verwendung von SIMD-Befehlen bei jedem Grad von Vorteil ist. Es kann jedoch behauptet werden, dass die iterativen Algorithmen in C, In-Place sowie Out-Of-Place, nicht die beste Wahl wären.

	Assembly	Assembly SIMD	In-Place	Out-Of-Place	Rekursiv
Grad 1	5,5%	829	132,9%	184,5%	177,3%
Grad 2	28,7%	3.546	68,0%	101,1%	53,7%
Grad 3	12,6%	19.267	14,7%	52,9%	4,3%
Grad 4	1,1%	207.927	6,7%	21,3%	2,3%
Grad 5	-1,2%	1.852.378	12,0%	13,8%	6,1%
Grad 6	-3,2%	5.997.885	5,2%	11,6%	0,7%
Grad 7	-0,6%	32.001.060	12,5%	33,2%	2,4%
Grad 8	3,9%	264.060.050	4,8%	14,9%	-1,4%
Grad 9	2,8%	2.339.423.500	2,0%	11,8%	-1,8%

Abbildung 4: Prozentueller Performanzvergleich (Zeitangabe in Nanosekunden)

Zudem zeigt die folgende Abbildung 5 ein Balkendiagramm. Um das exponentielle Wachstum der Ergebnisse sinnvoll darzustellen werden zuerst die Grade 4, 5 und 6 und anschließend Grad 6 mit 7 und 8 verglichen.

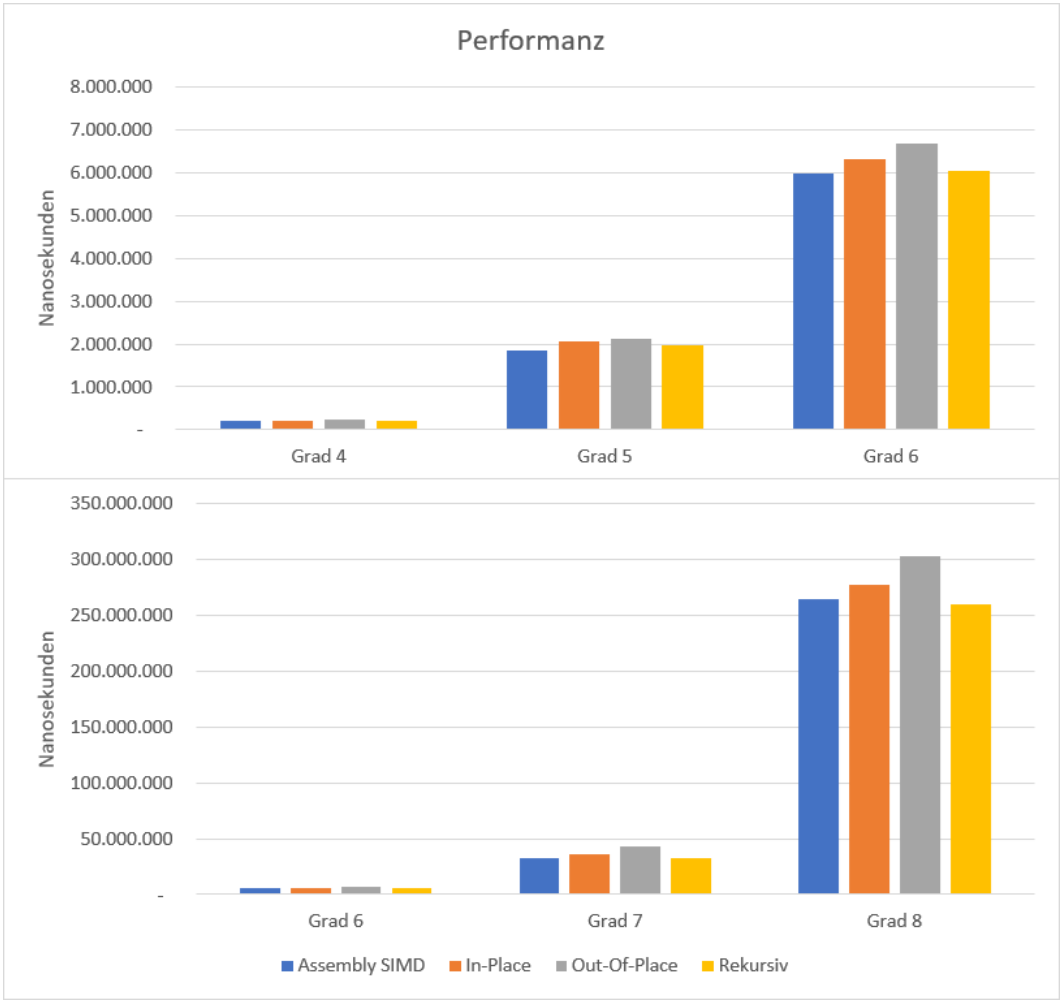

Abbildung 5: Grafischer Vergleich der Grade 4 bis 8

5 Zusammenfassung und Ausblick

Es lässt sich also feststellen, dass eine rekursive Implementierung gegenüber einem iterativer Ansatz vorzuziehen ist, da in unserer Implementierung der rekursive Algorithmus in den Testfällen bis Grad 9 mit Ausnahme $n = 1$ immer besser abgeschnitten hat, vergleiche Abbildung 4. Zu erwarten war, dass eine manuelle Optimierung des Assemblercodes zu besseren Performanzergebnissen führt. Jedoch sind die bisherig implementierten SIMD-Instruktionen bei höhren Graden nicht so performanzsteigernd wie

eingangs erhofft wenngleich verständlich, da nicht alle möglichen SIMD-Optimierungen durchgeführt wurden. Weitere Möglichkeiten die Performanz zu erhöhen wären Speicheroptimierungen, Multithreading, ein 16-Bit Alignment der SIMD-Instruktionen oder die im Kapitel 2.3 beschriebenen Alternativen.

Betrachtet man die Historie der Raumfüllenden Kurven, so fällt schnell auf, dass noch weitere Kurven existieren, die vergleichbare Eigenschaften zu der Peano-Kurve aufweisen. Eine solche Kurve wäre die Hilbert-Kurve, welche den Raum in vier anstatt neun Teile unterteilt. Es bietet sich also an, die hier ausgeführte Kurve mit beispielsweise der Hilbert-Kurve oder anderen Arten der Peano-Kurve [1], welche den Raum auf andere Weise durchläuft, zu vergleichen.

Literatur

[1] Albert Maas. *Hauptseminar: Oktalbäume und hierarchische Basen Thema: Raumfüllende Kurven*. Technische Universität München, Institut für Informatik, June 2003.

[2] Aspekte der systemnahen Programmierung bei der Spieleentwicklung. *Praktikum ASP – Projektaufgabe A214*. Technische Universität München, Lehrstuhl für Rechnerarchitektur und parallele Systeme, January 2021.

[3] Guangjun Yang, Xiaoling Yang, Ping Wang. *Arithmetic-Analytic Representation of Peano Curve*. International Journal of Mathematics and Mathematical Sciences, September 2019. `https://www.hindawi.com/journals/ijmms/2019/6745202/#introduction`, visited 2021-02-11.

[4] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*. Intel Corporation, May 2020.

[5] Baback Moghaddam, Kenneth J. Hintz, and Clayton V. Stewart. Space-filling curves for image compression. In Firooz A. Sadjadi, editor, *Automatic Object Recognition*, volume 1471, pages 414 – 421. International Society for Optics and Photonics, SPIE, 1991.

[6] Moritz Albert Schöbi. *Cantor- und Peanofunktionen*. Technische Universität Wien, December 2020.