LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Aspekte der systemnahen Programmierung bei der Spieleentwicklung

Gruppe 132 – Abgabe zu Aufgabe A214

Wintersemester 2020/21

Mohammed Attia Patrick Zimmermann Thomas Torggler

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 essentiell 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 surrjektive Funktion $f(x)$ erstellen, so das gilt: $\forall x \in \mathbb{R}^{n-1} \quad y \in \mathbb{R}^n$. Für einen Beweis siehe (QUELLE?). Hier wollen wir uns auf die sogenannten Peano-Kurven beschränken. Für eine solche Kurve definieren wir ein Intervall $I = [0; 1]$, sowie $f : I \rightarrow I^2$. Dann ist die Peano-Kurve: $\lim_{x \to \infty} f(x)$, mit $x \in I$. Sie { entspricht 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 durch eine Kurve besucht werden. Dadurch durchläuft die Kurve die Quadrate in Form eines „S“. In einem Iterationsschritt lässt sich eines der 9 Quadrate in weitere 9 Quadrate Unterteilen, die wiederum auf selbe Art verbunden werden, wie in Abb.1 gezeigt.

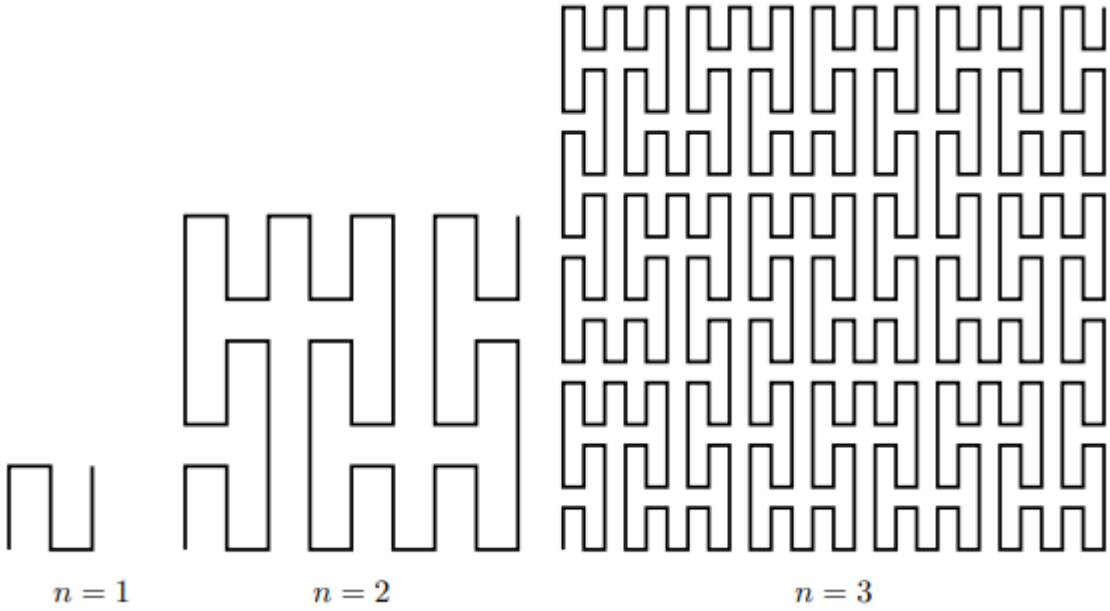

Abbildung 1: Peano-Kurve mit n = {1, 2, 3}, **[?]**

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 Definition werden wir jedoch

nicht nur nach $[0; 1]$ sonder nach $\mathbb{N}$ abbilden.

2 Lösungsansatz

Der Aufbau der Peano-Kurve ermöglicht es die Kurve des aktuellen Grades mit einer Variation der gespiegelten und originalen Kurve des vorherigen Grades zu zeichnen. Diese Observation war der Grundstein unseres iterativen Algorithmus. Anfangs wird dabei die Startkurve hardcodiert in einem Integer Array abgespeichert wobei die Zahlen von 0 bis 3 jeweils einer der vier Richtungen entsprechen. Falls der eingegebene Grad $n > 1$ ist, wird aufsteigend über alle Grade bis inklusive n iteriert wobei während jedem Schritt die Kurve des vorherigen Grades in der originalen Reihenfolge des ersten Grades entweder im Originalzustand oder gespiegelt acht mal in den Array eingefügt wird, sodass die vollständige Kurve des aktuellen Grades entsteht.

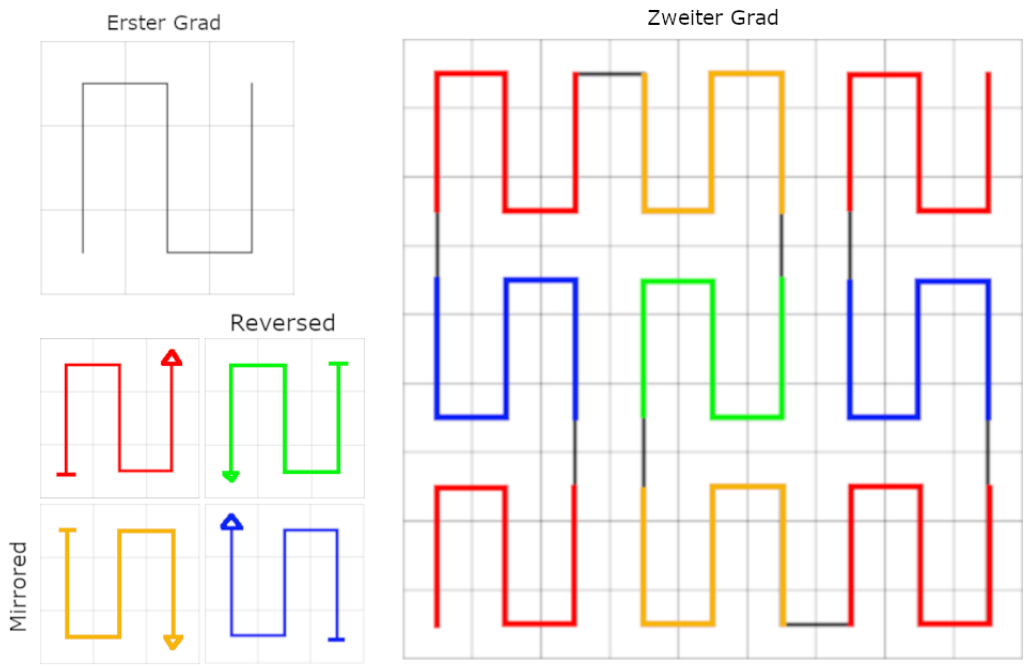

Abbildung 2: Peano-Kurve mit n = 1, n = 2

Zu beachten ist dabei jedoch, dass durch das Spiegeln und Einfügen von Kurven deren Richtung noch inkorrekt sein könnte, dafür gibt es aber ähnlich zur Spiegelungsmethode eine Methode zum Umkehren der Richtung. Nach diesen Schritten sind die neun erstellten Kurven jedoch noch nicht miteinander verbunden. Da sich das Muster der ursprünglichen Kurve durch alle Grade wiederholt, kann nach jedem Schritt ein hardcodierter Pfad zur nächsten Kurve eingefügt werden. Nachdem über alle Grade iteriert wurde, läuft der Algorithmus den vollständigen Richtung Array durch, verändert dabei bei jedem Schritt je nach Richtungsangabe entweder die x oder y-Koordinate und speichert diese dann in die Ausgabeliste der Peano-Methode.

3 Korrektheit

Die Genauigkeit unseres Algorithmus ist abhängig von dem Grad n, dass die Anzahl an Iterationen bestimmt, wie in den Abb.1 beschrieben. Das geht auch aus der beschriebenen mathematischen Definition hervor, da die Peano-Kurve an sich ein Grenzwert ist. Somit wird der Algorithmus genauer, je größer der Grad der Kurve ist, da die Punkte auf der Kurve sich ebenfalls einem Grenzwert annähern. Ansonsten ist die Kurve wie ebenfalls schon oben beschrieben definiert, weshalb nun 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 Assambler und in C, die wir auch hinsichtlich ihrer Performanz analysieren wollen. Wir wollen dennoch unseren Ansatz auf Korrektheit prüfen.

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 beschrieben, zum berechnen eine Kurve n-ten Grades nur die Kurve des Grads $n-1$ sowie die gespiegelte und invertierte Version dieser benutzen. Bei der als invertierten Version betitelten Permutation handelt es sich in diesem Kontext um eine Punktspiegelung im Mittelpunkt der vorhergehenden Kurve, siehe Abb.2, weitere Erläuterungen folgen.

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 fest gegeben. Des Weiteren ist die Definition der Kurve mit $n = 2$ für die Berechnung aller weiteren Kurven von Nöten, da man anhand dieser Kurve alle notwendigen Bedingungen der Konstruktion weiterer Kurven ableiten kann. Deshalb nehmen wir sie also auch als definiert an [**?**]. Diese Bedingungen sollen im folgenden ebenfalls geprüft werden.

3.1.2 Induktionsannahme

Da die Funktion wie vorher definiert surrjektiv ist, gilt:

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

Dadurch lassen sich Kurven mit höherem Grad konstruieren. Sei nun $n > 1$ und 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.

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 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 Abb.2, grüne Kurve.

Bei der Spiegelung wird die Kurve an der Y-Achse gespiegelt. Wir verändern also ebenfalls die vertikale Richtung, behalten jedoch weiterhin die horizontale Richtung bei. Dadurch wird die vorausgehende Kurve notwendiger weise verändert. Diese zusätzliche Operation ist ebenfalls elementar und somit direkt auf die Koordinatenberechnung Einfluss haben ohne die Kurve anderweitig zu verändern, ändert sich der charakteristische Verlauf der Kurve nicht. Siehe auch Abb.2, gelbe Kurve.

Abschließend bleibt noch die Kombination aus der vorangehenden Permutationen, die ebenfalls für die Konstruktion der Kurve von Nöten ist.

Die unterschiedlichen Permutationen der Peano-Kurve werden durch eine eben solche Kurve mit $n = 1$ verbunden. Dadurch bleiben alle Eigenschaften der Kurve erhalten, da wir keine weiteren Punkte in die Kurve einfügen, sonder nur die Art der graphischen Darstellung. Zudem ist dies auch ebenso Teil der Definition dieser Kurve.

4 Performanzanalyse

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 43.0 Megabyte (Peano-Kurve mit $n = 7$) öffnen kann. Mit zunehmendem Grad enthält die .svg Dateien immer mehr Punkte der Kurve, weshalb sie ebenfalls exponentiell zur Basis 9 steigt. Deshalb möchten wir an dieser Stelle auf das Kapitel *Korrektheit* 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 allokieren kann und der Heap zusätzlich in seiner Größe 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 allokieren, wozu wir eben nicht im Stande sind, weshalb wir den Wertebereich von n auf $[1; 9]$ limitiert haben.

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);
}
```

5 Zusammenfassung und Ausblick