Institut für Visualisierung und Interaktive Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 72

GPU-basierte Graphenvisualisierung

Tilo Pfannkuch

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Thomas Ertl
Betreuer/in:	Dipl.-Inf. Alexandros Panagiotidis, Dr. Guido Reina
Beginn am:	13. Mai 2013
Beendet am:	12. November 2013
CR-Nummer:	E.1, I.3.1

Kurzfassung

Visuellen Repräsentationen von Graphen begegnet man an vielen Stellen des Alltags, aber auch in der Wissenschaft und der Industrie. Diese basieren auf abstrakten Daten, die entweder implizit bei der Erstellung der Grafik entstanden sind oder die explizit aus einer Datenbank entnommen wurden, um sie durch eine schematische Zeichnung zugänglicher zu machen. Das Themengebiet des Graph-Drawings beschäftigt mit der Problematik, aus diesen abstrakten Daten eine visuelle Repräsentation zu schaffen. Eine Art der Darstellung ist die des Node-Link-Diagramms, in dem Knoten als positionierte Knotenpunkte und Kanten als Verbindungen zwischen diesen visualisiert werden.

Ein wichtiger Aspekt dabei ist die Bestimmung eines Knotenlayouts. Je nachdem, wo die Knoten gesetzt werden, lassen sich Eigenschaften der darunterliegenden Daten erkennen. Dies ruft die Notwendigkeit von Layout-Verfahren auf den Plan. Dabei gibt es verschiedene Ansätze, die ein Optimierungsproblem definieren und anschließend numerisch anwenden, da für nicht-triviale Problemgrößen eine direkte Lösung problematisch ist. So werden zum Beispiel Knoten elektrisch geladen und Kanten als Federn zwischen den Knoten modelliert oder es wird eine Übersetzung in ein algebraisches Problem vorgenommen, für das es bereits Lösungen gibt.

Im Zusammenhang der Problemgröße bietet es sich durch neue Entwicklungen in der Grafikhardware an, aufwändige Berechnungen durch die gebotene Parallelität auf der GPU zu beschleunigen. Es gibt dafür bereits einige Konzepte, die durch die damals aktuellen Schnittstellen nicht direkt, sondern nur durch umständliche Umwege umgesetzt werden konnten.

Neben dem Knotenlayout stellen sich weitere Fragen im Zusammenhang mit der Visualisierung von Graphen als Node-Link-Diagramme. Diese fließen zusammen in das Konzept eines Frameworks ein, das ein generelles Vorgehen präsentiert und dabei auf die Verwendung der Grafikhardware eingeht. Dazu gehört neben der Einbettung der Layout-Verfahren auch die Wahl passender Visualisierungen für Knoten und Kanten, aber auch eine Auseinandersetzung mit Interaktionskonzepten. Eine Implementierung dieses Frameworks bietet weitere Aufschlüsse über die konzipierten Strukturen und Zusammenhänge und die Möglichkeit der Evaluation bezüglich Speicher- und Laufzeitverhalten.

Inhaltsverzeichnis

1 Einleitung **9**

2 Grundlagen **13**
2.1 Graphen . . . 13
2.1.1 Vernetzungsgrad . . . 14
2.1.2 Positiv-semidefinite Graphen . . . 14
2.2 Computergrafik . . . 15
2.2.1 Bounding Volumes . . . 16
2.2.2 *k*-d-Baum . . . 17

3 Layout-Verfahren **19**
3.1 Hierarchisierung . . . 20
3.2 Force-Directed Layout . . . 20
3.2.1 Kraftmodell . . . 20
3.2.2 Temperatur . . . 21
3.3 Ausgewählte Layout-Verfahren . . . 22
3.3.1 Fruchterman-Reigold . . . 23
3.3.2 Edge-Edge Repulsion . . . 24
3.3.3 Algebraic Multigrid Computation of Eigenvectors . . . 25
3.3.4 Kamada-Kawai . . . 28

4 Verwandte Arbeiten **29**
4.1 Mehrstufiges Graph-Layouting auf der GPU . . . 29
4.2 Schnelles Zeichnen von Graphen auf der GPU mit Hilfe von Multipolentwicklung . . . 31
4.3 Software . . . 33

5 Konzept eines Frameworks zum Zeichnen von Graphen **35**
5.1 Datenstrukturen . . . 36
5.1.1 Adjazenzliste . . . 36
5.1.2 Knoten und Kantenattribute . . . 37
5.1.3 Layout und Hilfsdaten . . . 39
5.2 Programmkomponenten . . . 39
5.3 Preprocessing . . . 40
5.4 Visualisierung . . . 41
5.4.1 Knoten . . . 42
5.4.2 Kanten . . . 43

5.5 Einbettung von Layout-Verfahren 47
5.5.1 Initialisierung 48
5.5.2 Auflösung von Positionssingularitäten 49
5.5.3 Berücksichtigung von Bounding Volumes 49
5.6 Interaktion 54
5.6.1 Speichern und Laden 54
5.6.2 Veränderung von Layout-Parametern 54
5.6.3 Selektion 55
5.6.4 Kameratransformation 56
5.6.5 Berechnung der Bounding Volumes 56

6 Implementierung 57
6.1 Speicherlayout und Datenstrukturen 58
6.2 Linearisierung und der Einsatz von Compute-Shadern 60
6.3 Ein Beispiel für ein Reduktions-Programm 61
6.4 Einbettung von Fruchterman-Reigold 62

7 Evaluation und Ausblick 63
7.1 Visualisierung 63
7.2 Speicherkomplexität 63
7.3 Laufzeitanalyse 68
7.4 Zusammenfassung 71
7.5 Ausblick 71

Abbildungsverzeichnis

2.1 Rendering-Pipeline 15
2.2 Axis-Aligned Bounding Box 17
2.3 *k*-d-Baum 17

3.1 Hierarchisierung 20
3.2 Kraftmodell 24

5.1 Graph-Drawing-Pipeline 35
5.2 Speicherlayout und Indizierung 37
5.3 Fixierung von Knoten 38
5.4 Tessellierung von Knoten und Kanten 41
5.5 Tesselierung eines Quads zu einem kreisförmigen Polygon 43
5.6 Tesselierung eines Quads zu einer gebogenen Kante 44
5.7 Modifizierte Verbindungsvektorberechnung 51
5.8 Resultierender Verbindungsvektor 53

7.1 Visualisierung des Amazon-Graph 64
7.2 Speicherkomplexität 65
7.3 Messergebnisse (1) 68
7.4 Messergebnisse (2) 69
7.5 Messergebnisse (3) 70
7.6 Messergebnisse (4) 70

Tabellenverzeichnis

6.1 Funktionsüberblick der Implementierung 57

7.1 Beispieleingaben 65

7.2 Speicherverbrauch der Adjazenzliste 67

Code-Verzeichnis

3.a Potenzmethode [KCH03] 26

6.a Datenstruktur für zweidimensionale Vektoren 58
6.b Knotendatenstruktur 58
6.c Kantendatenstruktur 59
6.d Datenstruktur für Bounding Boxes 59
6.e Server-seitiger Speicherzugriff 60

1 Einleitung

In der Realität wird man häufig mit schematischen Darstellungen, wie dem Streckennetz alternativer Verkehrsmittel, Wissen vermittelnder Schaubilder oder sogar dem Spielbrett eines Gesellschaftsspiels konfrontiert, die uns das Verständnis der zugrundeliegenden Daten erleichtern. In der Mathematik wird ein Graph als eine abstrakte Struktur bestehend aus einer Menge von Knoten und deren Verbindungen formalisiert. Viele Datenbanken enthalten Informationen, die sich mit dieser formalen Definition decken. Auf der Website des *Stanford Network Analysis Project* [Les] finden sich einige Beispiele, wie die Verbindungen innerhalb sozialer Netzwerke, Analysedaten der Verkaufsplattform Amazon oder die Struktur von Straßennetzen. Die Knotenmengen bestehen dabei jeweils aus den Accounts der Benutzer, angebotenen Produkten beziehungsweise Kreuzungen, die mit sozialen Verbindungen, Verkaufsrelationen respektive Straßen miteinander verbunden sind.

Der Mensch tut sich schwer daran, Daten in ihrer Rohform, insbesondere bis ins kleinste Detail, aufzunehmen und zu verarbeiten. Eine grafische Darstellung eines Graphen bietet viele Vorteile in Bezug auf die Perzeption des Menschen. So kann durch einen ersten Blick die grobe Struktur der Daten erfasst werden und nach einer Fokussierung auf interessante Bereiche erleichtert die Darstellung das Verständnis der vorliegenden Information, da Bilder leichter verarbeitet werden können als textuelle Daten. Die Disziplin des Graph-Drawing beschäftigt sich mit der Problematik, die abstrakte Repräsentation eines Graphen in eine grafische Darstellung zu überführen.

Diese Visualisierung kann in einem sogenannten Visual-Analytics-Prozess zum Einsatz kommen. Von Landesberger u. a. [Von+11] beschreiben ihn als einen iterativen Prozess. Jede Iteration beginnt mit einer Datenbank, die mit Hilfe des bisherigen Wissens vorbereitet wird, bevor deren Daten in zwei verschiedene voneinander abhängige Repräsentationen überführt werden: Die Visualisierung der Daten und die Konstruktion eines Datenmodells. Durch Interaktion mit dem System können Interessenschwerpunkte ausgemacht und Parameter verfeinert werden. Mit diesem neuen Wissen beginnt ein weiterer Iterationsschritt des Gesamtprozesses.

Die erste Entscheidung, die bei der Darstellung eines Graphen getroffen werden muss, liegt bei der grundlegenden Struktur. Diese Arbeit beschränkt sich auf Node-Link-Diagramme. Für einen Überblick über weitere Möglichkeiten wird auf Abschnitt 3 aus dem Artikel „Visual Analysis of Large Graphs: State-of-the-Art and Future Research Challenges“ [Von+11] verwiesen. In dieser Art der Darstellung werden Knoten als positionierbare Objekte betrachtet, die miteinander verbunden sind. Grundsätzlich besteht die Aufgabe also aus der Konzeption

1. eines Layouts (einer Platzierung) der Knoten,
2. deren Visualisierung und

3. der Darstellung der Verbindungen.

Nur in Ausnahmefällen, wie dem oben genannten Straßennetz, sind die Knotenpositionen einer Visualisierung des Graphen inhärent. Deshalb besteht ein großer Teil des Graph-Drawings darin, diese zu bestimmen.

Der Qualitätsbegriff einer solchen Visualisierung lässt sich nur teilweise objektiv formulieren. Außer Frage ist, das die Qualität fest mit der Laufzeit gekoppelt ist, insbesondere wenn Interaktion mit im Spiel ist. Jedoch lassen sich ästhetische Qualitätsmerkmale nur subjektiv formulieren, da diese vom persönlichen Empfinden des Betrachters abhängen. Einige Beispiele wären die Minimierung von Kantenüberschneidungen und des Darstellungsvolumens und die Maximierung von Symmetrien [Von+11]. Lin und Yen [LY12] stellen den Fokus auf das Auflösungsvermögen der Winkel ausgehender Kanten eines Knotens. Sollten überlappende Knoten deren Visualisierung verfremden oder unkenntlich machen oder sogar den Eindruck eines einzigen Knotens vermitteln, gehört zu dieser Liste sicherlich auch das Bestreben Knoten überlappungsfrei darzustellen.

Insbesondere mit Blick auf die Laufzeit der Visualisierung ergibt sich die Frage mit welchen Datenmengen man es unter Umständen zu tun hat. Die größten Datenbanken der oben genannten drei Beispiele sind der Reihenfolge nach 4 847 571 Knoten mit 68 993 773 Kanten, 403 394 Knoten mit 3 387 388 Kanten und 1 965 206 Knoten mit 5 533 214 Kanten. Es ist offensichtlich, dass diese Größenordnungen bei quadratischer Komplexität untragbar werden. Die vollständige Diskussion darüber, inwieweit die Visualisierung solcher Daten noch sinnvoll ist, wird an dieser Stelle übersprungen. Insofern eine Realisierung möglich ist, kann die komplette Darstellung des unveränderten Graphen, also insbesondere ohne aktives Preprocessing, zumindest für die Bestimmung der Grundstruktur verwendet werden. Mit geringem Aufwand kann auch die Vergrößerung der Darstellung für die Fokussierung interessanter Bereiche umgesetzt werden. Die jüngste Entwicklung, die die Grafikhardware erfahren hat, macht das Wagnis vorstellbar, direkt mit solchen und größeren Datenmengen zu arbeiten. Denn die Grafikhardware ist für massive Parallelität ausgelegt. So hat die aktuelle GeForce GTX TITAN von NVIDIA 2688 Recheneinheiten und 6144 MB Speicher mit einer Speichergeschwindigkeit von bis zu 6 GB/s [NVI].

Die neuen Entwicklungen in der Grafikhardware und deren Schnittstellen motivieren eine erneute Auseinandersetzung mit der Implementierung von Layout-Verfahren für Node-Link-Diagramme. Es gibt jedoch ein sehr breites Spektrum an Themengebieten in diesem Kontext, die über das Layouting hinaus gehen. Anstatt sich in eine bestimmte Richtung zu vertiefen, bietet diese Arbeit einen weitreichenden Blick von Layout-Algorithmen, über eine Visualisierung abstrakter Daten, wie Graphen es sind, zu Überlegungen hinsichtlich der Interaktion mit der Visualisierung. Das entwickelte konzeptionelle Framework zielt nicht darauf eine Lösung für ein Problem zu sein, sondern eine Basis für Lösungen vieler damit verbundener Probleme bereitzustellen. Eines dieser Probleme beschäftigt sich mit der Visualisierung von dynamischen Graphen. Das sind Graphen, die sich über die Zeit verändern. Dies ruft neue Probleme bei der Speicherverwaltung und der Formulierung des Qualitätsbegriffs hervor. Es werden hier ausschließlich statische Graphen betrachtet. Für eine Betrachtung der Problematik und einen Ansatz für eine Implementierung auf der GPU wird auf den Artikel „Online dynamic graph drawing“ [FT08] verwiesen.

Diese Arbeit besteht aus zwei großen Teilen. Sie stellt einerseits die generelle Methodik zur Bestimmung des Layouts von Node-Link-Diagrammen vor. Der zweite Teil besteht aus der Konzeption eines GPU-beschleunigten Frameworks, in das diese Methoden eingebettet werden können. Dazu gehört die Entwicklung einer GPU-zentrischen Datenstruktur und der zugehörigen Graph-Drawing-Pipeline unter Berücksichtigung von Interaktion, die jeweils evaluiert werden.

In Kapitel 2 werden nötige Grundlagen erklärt und definiert, auf denen nachfolgende Kapitel aufbauen. Einige Herangehensweisen an das Problem der Bestimmung eines Knotenlayouts werden in den Kapiteln 3 und 4 vorgestellt, wobei erstere die theoretischen Grundlagen der anderen Verfahren legen, bei denen die Layout-Berechnung in die Architektur der GPU eingebettet ist und deren Autoren sehr viel Anstrengung in die Konzeption eines effizienten Algorithmus investieren. Kapitel 5 beschreibt ausführlich die Struktur eines Frameworks zum Zeichnen von Graphen, von dem ein Teil implementiert wurde. Die Ergebnisse befinden sich teilweise ebenfalls in Kapitel 5 in Konzepten oder Verfahren wieder. In den letzten beiden Kapiteln 6 und 7 wird genauer auf Probleme der Implementierung und auf die Komplexität einer einfachen Umsetzung eines Layout-Verfahrens eingegangen.

2 Grundlagen

Die Visualisierung von Graphen beruht auf einigen theoretischen und methodischen Grundlagen. Die kurze formale Beschreibung eines Graphen aus der Einleitung muss detaillierter formuliert werden, sodass die damit verbundene Begrifflichkeit und Symbolik verstanden werden kann. Außerdem erfolgt eine kurze Einführung in die Computergrafik anhand der Grafikprogrammierschnittstelle OpenGL.

2.1 Graphen

Ein endlicher gewichteter Graph $\mathcal{G}$ ohne Schlingen wird nachfolgend durch $\mathcal{G} := (\mathcal{V}, \mathcal{E}, \omega)$ charakterisiert. $\mathcal{V} := \{0, \ldots, n-1\}$ mit $n \in \mathbb{N}$ beschreibt die Menge aller Knoten, die Menge der Kanten wird mittels

$$\mathcal{E} :\subseteq \begin{cases} \{\, (i,j) \mid i,j \in \mathcal{V},\, i \neq j\}^2 \times \mathbb{N} & \text{gerichtet,} \\ \{\, \{i,j\} \mid i,j \in \mathcal{V}\}^2 \times \mathbb{N} & \text{ungerichtet} \end{cases}$$

definiert, wobei $m := |\mathcal{E}| < \infty$. Die Kantengewichtsfunktion $\omega : \mathcal{E} \to \mathbb{R}^{\geq 0}$ ordnet jeder Kante ein Gewicht zu.

Im Speziellen stellt $e^k_{i,j} := (i,j)_k$ eine Kante von Knoten $i \in \mathcal{V}$ nach Knoten $j \in \mathcal{V}$ dar. Da die Richtung bei einem ungerichteten Graphen von keiner Bedeutung ist, gilt für eine ungerichtete Kante $e^k_{i,j} = e^k_{j,i} := \{i,j\}_k$. Diese Definition erlaubt die Existenz von Mehrfachkanten, die sich nur um den Index $k \in \mathbb{N}$ unterscheiden.

Zur Vereinfachung können die Komponenten eines Graphen $\mathcal{G} = (\mathcal{V}', \mathcal{E}', \omega)$ mit den folgenden Abbildungen identifiziert werden: $\mathcal{V}(\mathcal{G}) = \mathcal{V}'$ beziehungsweise $\mathcal{E}(\mathcal{G}) = \mathcal{E}'$. Zudem wird eine Relation $\mathcal{G}' \subset \mathcal{G}$ definiert, die alle Teilgraphen $\mathcal{G}'$ von $\mathcal{G}$ enthält. Deren Qualifikation erfolgt durch die Bedingungen $\mathcal{V}(\mathcal{G}') \subset \mathcal{V}(\mathcal{G})$ und

$$\mathcal{E}(\mathcal{G}') = \left\{ e^k_{i,j} \;\middle|\; i,j \in \mathcal{V}(\mathcal{G}),\, e^k_{i,j} \in \mathcal{E}(\mathcal{G}) \right\}.$$

Soll $\mathcal{G}$ keine Mehrfachkanten aufweisen, wird nur der Index $k = 0$ zugelassen. Insbesondere sind in diesem Fall $e_{i,j} = (i,j)$ beziehungsweise $e_{i,j} = e_{j,i} = \{i,j\}$ Abkürzungen für $e^0_{i,j} = (i,j)_0$ beziehungsweise $e^0_{i,j} = e^0_{j,i} = \{i,j\}_0$.

Für den Umgang mit Mehrfachkanten ist die Definition zusätzlicher Strukturen hilfreich. Für $i,j \in \mathcal{V}$ und $k \in \mathbb{N}$ kategorisieren

$$\mathcal{A}^k_{i,j} = \mathcal{A}^k_{j,i} := \left\{ e^k_{p,q} \in \mathcal{E} \;\middle|\; p,q \in \{i,j\} \right\}$$

alle Kanten gleichen Indexes k zwischen i und j. Diese Adjazenzmengen enthalten maximal zwei gerichtete beziehungsweise eine ungerichtete Kante und sind paarweise disjunkt, sofern sie nicht leer sind. Weiterhin umfasst die Vereinigung

$$\mathcal{A}^*_{i,j} := \bigcup_{k \in \mathbb{N}} \mathcal{A}^k_{i,j}$$

alle Kanten beliebigen Indexes k.

2.1.1 Vernetzungsgrad

Bei dem Versuch das World Wide Web abzubilden fiel Barabási und Bonabeau [BB03] eine Eigenschaft natürlicher Netzwerke auf. Bei zufälligen Netzwerken ist der Grad $\deg i$ eines Knotens $i \in \mathcal{V}$ glockenförmig Poisson-verteilt, das heißt nur wenige Knoten weichen stark von dem Durchschnittsgrad des Netzwerks ab. Jedoch weisen natürliche Netzwerke eine Potenzverteilung $P(\deg i = k) \sim {}^1/_{k^n}$ für ein $n > 0$ auf. Es handelt sich um sogenannte skalenfreie Netzwerke und deren Repräsentationen – skalenfreie Graphen. In einem ungerichteten Graphen definiert

$$\deg i = \sum_{\{i,j\} \in \mathcal{E}} \omega(i,j)$$

den Grad eines Knoten $i \in \mathcal{V}$.

Generell wird Komplexität der Kantenanzahl m durch drei besondere Zusammenhänge charakterisiert.

1. $m = m_{max} := c \cdot n(n-1)$ stellt die theoretische Grenze möglicher Verbindungen dar. Für ungerichtete Graphen ist $c = {}^1/_2$, da bei gerichteten die Rückrichtung hinzu kommt, gilt dort $c = 1$ und im Falle der Existenz von Mehrfachkanten verschwimmt die Grenze. Da $m < \infty$ lässt sich jedoch auch hier eine Konstante $c \in \mathbb{R}$ bestimmen. Insbesondere bei den ersten beiden Fällen spricht man von einem vollständigen Graphen.
2. $\mathcal{G}$ wird als dünn besetzt bezeichnet, falls $n \leq m \leq m_{sp} \ll m_{max}$ [Von+11]. Die obere Grenze m_{sp} ist hier ebenfalls unscharf.
3. Für $m < n$ existiert mindestens ein Knoten, der zu keinem anderen verbunden ist, $\mathcal{G}$ ist also nicht zusammenhängend.

2.1.2 Positiv-semidefinite Graphen

Für spezielle Verfahren reicht die obige Definition nicht aus [KCH03]. Durch eine Erweiterung um eine Knotengewichtsfunktion $\gamma : \mathcal{V} \to \mathbb{R}^{>0}$ wird ein sogenannter positiv-semidefiniter (PSD-)Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \gamma, \omega)$ definiert, wobei mit $\omega : \mathcal{E} \to \mathbb{R}$ auch negative Kantengewichte erlaubt sind, insofern die zugehörige Laplace-Matrix L positiv-semidefinit ist. Diese ist definiert als

$$L_{i,j} := \begin{cases} \deg i & i = j, \\ -\omega(i,j) & \{i,j\} \in \mathcal{E}, \\ 0 & \text{sonst} \end{cases}$$

für $i, j = 0, \ldots, n-1$. Notiert man die Werte von γ ebenfalls in einer Matrix M mit

$$M_{i,j} := \begin{cases} 0 & i \neq j, \\ \gamma(i) & i = j, \end{cases}$$

ergibt sich zusammen mit L die Alternativnotation $\mathcal{G} := (L, M)$. Diese Definition beschränkt sich auf ungerichtete Graphen ohne Mehrfachkanten. Für herkömmliche Graphen gilt in dieser Notation $\forall i \in \mathcal{V}: \gamma(i) = 1$.

2.2 Computergrafik

Das Handwerkszeug, bestehend aus theoretische Grundlagen und darauf aufbauenden Verfahren, mit dessen Hilfe eine Visualisierung umgesetzt werden kann, findet sich unter dem Begriff der Computergrafik. Damit eine Anwendung, der Client, mit der Grafikhardware, dem Server, kommunizieren kann, bedarf es einem Application Programming Interface (API), das eine Umsetzung dieser Methodik ermöglicht.

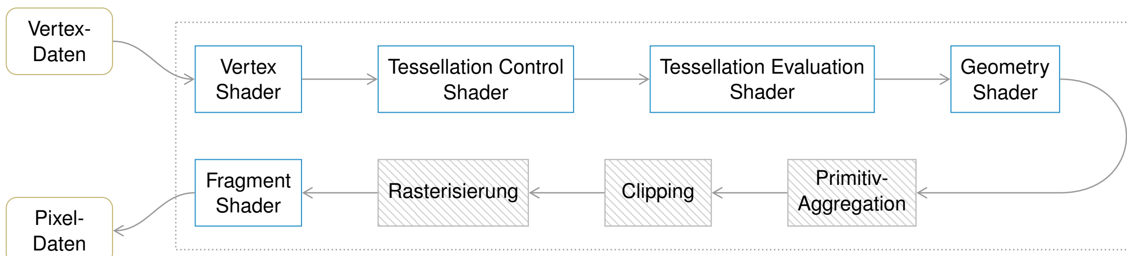


Abbildung 2.1: Die Rendering-Pipeline von OpenGL[Shr+13, Abbildung 1.2]

Shreiner u. a. [Shr+13] geben eine gute Einführung in die Computergrafik und insbesondere in modernes OpenGL, die nachfolgend zusammengefasst wird.

Der erste wichtige Begriff ist das Rendering, der das Ziel der Computergrafik auf den Punkt bringt. Es bezeichnet den Prozess, in dem ein Computer aus einer vorliegende Menge von abstrakten Objekten ein Bild erzeugt. Die Objekte bestehen dabei aus Vertices, von denen jeweils die passende Anzahl eines der geometrischen Primitive Punkt, Linie oder Dreieck beschreiben. Das erzeugte Bild wird im sogenannten Framebuffer gespeichert, einer Speichermatrix, die jedem Pixel der Anzeige einen Farbwert zuordnet. In OpenGL wird dieser Prozess durch eine Pipeline umgesetzt, die auf dem Prinzip der Rasterisierung aufbaut. Die Struktur dieser Pipeline ist in Abbildung 2.1 zu sehen. Die schraffierten Teile sind fest vorgegeben, der Rest kann durch die Spezifikation von Shader-Programmen vom

Programmierer beeinflusst werden. Für einen Pipeline-Durchlauf kann jeweils nur ein Shader pro Abschnitt aktiv sein.

Zunächst wird für jeden Vertex eine dedizierte Instanz des Vertex Shaders gestartet. Seine Aufgabe kann darin bestehen alle nötigen Attribute, wie die Position oder die Farbe pro Vertex zu berechnen, aber auch einfach nur die vorhandenen Daten weiterzureichen (Pass-Through). Optional können diese Daten mit dem Shader-Paar Tessellation Control Shader (TCS) und Tessellation Evaluation Shader (TES) verfeinert werden. Zunächst werden im TCS aus vorhandenen Primitiven weitere Primitive erzeugt, die anschließend im TES, der eine ähnliche Rolle wie der Vertex Shader einnimmt, attribuiert werden können. Der gleichfalls optionale Geometry Shader erlaubt die zusätzliche Verarbeitung einzelner Primitive und kann ebenso neue Geometrie erzeugen. Die finalen Vertex-Daten und deren Attribute werden erst in der Primitiv-Aggregation zu tatsächlicher Geometrie zusammengesetzt. Befindet sich ein Teil dieser Geometrie außerhalb des sichtbaren Bereichs, müssen betroffene Primitive durch Clipping abgeschnitten werden, bevor durch die Rasterisierung die Geometrie in Pixelkandidaten, sogenannte Fragmente, überführt wird. Für jedes dieser Fragmente wird eine Instanz des Fragment Shaders aufgerufen, um dessen endgültige Farbe zu bestimmen, sodass abschließend durch Regeln der Sichtbarkeit die Farbe des zugehörigen Pixels festgelegt wird.

Die Anzeige ist einerseits begrenzt durch horizontale und vertikale Anzahl an Pixeln und andererseits auf zwei Dimensionen beschränkt. Dreidimensionale Objekte und verschiedene Blickwinkel können mittels einer Projektionstransformation umgesetzt werden. Dazu kommen zwei weitere Transformationen, darunter die Modelltransformation, die für jedes Objekt unterschiedlich sein kann. Deren Aufgabe ist unter anderem die Positionierung der Objekte. Desweiteren sorgt eine View-Transformation dafür, dass die Position einer virtuellen Kamera relativ zu den transformierten Objekten, der Szene, ausgerichtet werden. Die Verwendung homogener Koordinaten ermöglicht OpenGL nach Anwendung der Projektionstransformation diese automatisch auf die Anzeige zu projizieren (vergleiche dazu Abbildung 5.3 aus *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3, Eighth Edition* [Shr+13]).

Zusätzlich zu dieser teilweise programmierbaren Rendering-Funktionalität können abseits der Pipeline Compute-Shader spezifiziert werden, die parallelisierbare Berechnungen auf der Grafikhardware ausführen. Diese haben Zugriff auf den gleichen Speicher, wie die Shader aus der Rendering-Pipeline, und können auf diese Weise gegebenenfalls über gemeinsame Daten mit diesen verbunden werden.

2.2.1 Bounding Volumes

Objekte einer Szene können eine komplexe Oberflächenstruktur ausweisen. Um davon abhängige Berechnungen, wie Schnitt oder Kollisionserkennung bei physikalischen Simulationen zu simplifizieren, werden einfachere Objekte, sogenannte Bounding Volumes konstruiert, die das Objekt vollständig einschließen.

Eine Bounding Box ist eine Spezialisierung eines solchen Bounding Volumes, die einen Quader definiert. Sind die Seiten der Bounding Box an den Achsen des Koordinatensystems orientiert, handelt es sich um eine Axis-Aligned Bounding Box (AABB).

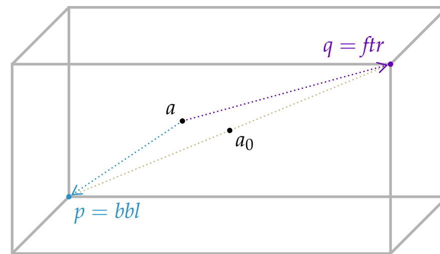
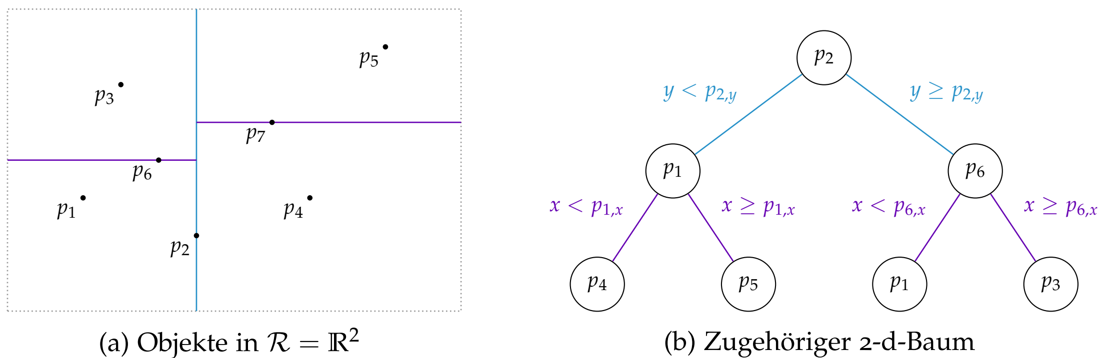


Abbildung 2.2: Axis-Aligned Bounding Box in $\mathcal{R} = \mathbb{R}^3$

Diese wird hier durch Angabe zweier Ortsvektoren $p, q \in \mathcal{R}$ definiert mit $\forall d = 0, \ldots, \dim \mathcal{R} - 1 : \; p_d \leq 0 \wedge q_d \geq 0$. Diese beschreiben die Lage zweier diagonal gelegener Eckpunkte der Bounding Box relativ zu einem Aufhängepunkt $a \in \mathcal{R}$ (für $\mathcal{R} = \mathbb{R}^2$ unten links beziehungsweise oben rechts). Falls $p = q = (0, \ldots 0)^T$, stellt die Bounding Box lediglich einen Punkt dar. Das Zentrum der Bounding Box ergibt sich aus $a_0 := a + 1/2\,(p + q)$, wobei für $p + q = (0, \ldots, 0)^T$ die beiden Punkte aufeinander fallen. Siehe Abbildung 2.2 für den Fall $\mathcal{R} = \mathbb{R}^3$.

Folgende zwei Operationen vereinfachen den mathematischen Umgang mit AABBs. Gegeben $B := (p, q)$ und $\Delta a \in \mathcal{R}$ sei $\Delta a \oplus B = B \oplus \Delta a := (p + \Delta a, q + \Delta a)$ als Addition der beiden Strukturen definiert. Desweiteren sei das Maximum von B und einer zweiten Bounding Box B' komponentenweise definiert als $\max_d \{B, B'\} := \left(\min\{p_d, q'_d\}, \max\{p_d, q'_d\}\right)$ für $d = 0, \ldots, \dim \mathcal{R} - 1$.

2.2.2 *k*-d-Baum



(a) Objekte in $\mathcal{R} = \mathbb{R}^2$ (b) Zugehöriger 2-d-Baum

Abbildung 2.3: Beispiel eines *k*-d-Baums

Für die räumliche Partitionierung einer Menge von Objekten, kann ein sogenannter *k*-d-Baum eingesetzt werden. Dabei steht *k* für die Dimensionalität der vorliegenden Szene. Im Prinzip handelt es sich um einen Binärbaum, das heißt für jeden Knoten gilt, dass dessen zwei Kinder jeweils kleiner (links) beziehungsweise größer gleich (rechts) sind. Dabei entscheidet für jeden Level h der Diskriminator $d = h \bmod k$ welche Komponente der *k*-dimensionalen Koordinaten zum Vergleich herangezogen wird. Die Definition und weitere Ausführungen

finden sich im Artikel „Multidimensional binary search trees used for associative searching“ [Ben75].

Für den Aufbau eines balancierten k-d-Baumes aus einer gegeben Menge von Objekten wird jeweils der Median bezüglich Komponente d gewählt, um die vorhandenen Partitionen der Objekte weiter zu unterteilen [FT07]. Abbildung 2.3 zeigt ein Szenario und den daraus resultierenden k-d-Baum.

3 Layout-Verfahren

Wie bereits in der Einleitung angesprochen, gehört die Platzierung der Knoten zu einer der Hauptaufgaben der Visualisierung eines Graphen. Formal wird also eine Abbildung $\mathcal{L} : \mathcal{V} \to \mathcal{R}$ gesucht, die jedem Knoten $i \in \mathcal{V}$ eine Position aus einem Darstellungsraum $\mathcal{R}$ zuordnet.

Das Kernprinzip aller hier vorgestellten Verfahren ist die Formulierung eines diskreten Energiefunktionals, dessen Minimierung angestrebt wird. Dies kann durch explizite Angabe geschehen oder implizit aus dem Verfahren folgen. Eine direkte Lösung der expliziten Optimierungsprobleme stellt sich für nicht-triviale Problemgrößen als besonders schwierig heraus. So muss beispielsweise ein n-dimensionaler Vektor berechnet werden, der ein diskretes Funktional minimiert [KK89, Abschnitt 3.3.4], oder der niedrigste Eigenwert einer $n \times n$-Matrix [KCH03, Abschnitt 3.3.3], wobei $n = |\mathcal{V}|$. Es muss also auf numerische Iterationsverfahren zurückgegriffen werden. Der Iterationsgedanke findet sich auch in den impliziten Verfahren, so wird zum Beispiel in jedem Iterationsschritt die Wirkung eines Kraftmodells auf die aktuellen Knotenpositionen angewendet [FR91; LY12, Abschnitt 3.2].

Eine wichtige Informationsquelle für die Bestimmung eines Layouts ist die Adjazenz der Knoten, also die Kantenmenge $\mathcal{E}$. Gerichtete Graphen mit oder ohne Mehrfachkanten müssen jedoch für die Berechnungen in ungerichtete Graphen ohne Mehrfachkanten überführt werden, da Richtung oder Grad einer Verbindung nicht relevant sind. Auf diese Eigenschaften wird erst bei der Visualisierung der Kanten eingegangen.

Der Darstellungsraum des Layouts wird häufig auf $\mathcal{R} = \mathbb{R}^2$ beschränkt, um einerseits Laufzeit und Speicherplatz zu sparen. Anderseits kommt es bei der unvermeidbaren Projektion eines dreidimensionalen Layouts, um es auf einer Anzeige darstellen zu können, zu Überdeckungseffekten, die nur innerhalb einer Anwendung durch Veränderung der Perspektive aufgelöst werden können. Zwischen diesen beiden Extremen befinden sich sogenannte 2.5D-Layouts, von denen sich einige Beispiele im Artikel „Navigation techniques for 2.5 D graph layout“ [AH07] finden.

Nachfolgend wird kurz auf das Konzept der Hierarchisierung eingegangen, das sich in effizienten Implementierungen wiederfindet. Da viele Layout-Verfahren und deren Implementierungen auf der Idee des Force-Directed-Layout(FDL) basieren, wird dieses anschließend allgemein genauer betrachtet. Neben den zwei Repräsentanten dieser Verfahren, Fruchterman-Reigold und Edge-Edge-Repulsion (EER), werden zwei weitere, vorgestellt, die auf dem Standpunkt der Energieminimierung stehen: Algebraic Multigrid Computation of Eigenvectors (ACE) und Kamada-Kawai.

3.1 Hierarchisierung

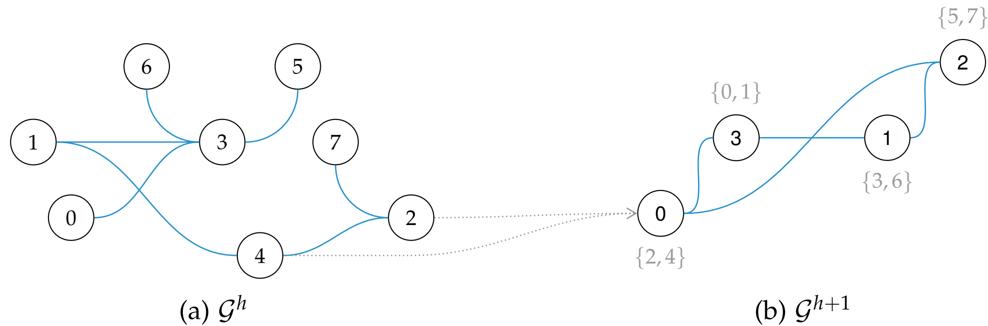


Abbildung 3.1: Beispiel für eine Bestimmung von $\mathcal{G}^h + 1$ zur Hierarchisierung eines ungerichteten Graphen $\mathcal{G}^h$

Mit einer sinnvollen Abbildung kann aus einem Graph $\mathcal{G}^h = (\mathcal{V}^h, \mathcal{E}^h, \omega^h)$ ein kleinerer Graph $\mathcal{G}^{h+1} = (\mathcal{V}^{h+1}, \mathcal{E}^{h+1}, \omega^{h+1})$ gewonnen werden, dessen Knoten $i \in \mathcal{V}^{h+1}$ jeweils eine Gruppe von Knoten $j \in \mathcal{V}^h$ zusammenfassen (vergleiche Abbildung 3.1). Es gilt also $|\mathcal{V}^{h+1}| \leq c \cdot |\mathcal{V}^h|$ mit üblicherweise $c \leq 1/2$. Wiederholt man diese Abbildung auf den jeweils entstehenden Graphen beginnend mit $\mathcal{G}$ erhält man eine Folge $\mathcal{G}^0 = \mathcal{G}, \mathcal{G}^1, \ldots, \mathcal{G}^c$, die eine Hierarchie von $\mathcal{G}$ darstellt. Dabei befindet sich $\mathcal{G}^c$ auf der höchsten Abstraktionsebene, die so gewählt ist, dass die Berechnung mit den dort vorliegenden Daten schnell genug durchgeführt werden kann. Hinter dem Sinn der Hierarchisierung steht eine wichtige Eigenschaft, die $\mathcal{G}^0, \mathcal{G}^1, \ldots, \mathcal{G}^c$ aufweisen müssen: Für einen Anwendungsfall relevante Eigenschaften von $\mathcal{G}^{h+1}$ müssen sich mit genügend geringer Abweichung mittels Interpolation auf $\mathcal{G}^{h+1}$ übertragen lassen. Dadurch ergeben sich gute Startwerte für die iterative Berechnung des Layouts von $\mathcal{G}^{h+1}$ [KCH03; FT07; God+09].

3.2 Force-Directed Layout

Der Kern von FDL-Verfahren liegt darin, aus gegebenen Positionsdaten der Knoten, dem aktuellen Layout, ein neues Layout zu berechnen, indem anhand eines Kraftmodells festgestellt wird, welche Kräfte auf die Knoten ausgeübt werden. Anhand dieser wird dann eine Verschiebung durchgeführt, in der Hoffnung dadurch näher an das optimale Layout heranzurücken. Je nach dem wie feingranular diese Verschiebung durchgeführt wird, kann es passieren, dass ein lokales Minimum den iterativen Prozess stilllegt. Umso wichtiger ist ein gutes initiales Layout, damit das erreichte Minimum das globale und nicht eines der lokalen Minima ist.

3.2.1 Kraftmodell

Eine zeitabhängige, eindimensionale und kontinuierliche Krafteinwirkung $\widetilde{F} : \mathbb{R} \to \mathbb{R}$ auf ein ruhendes Objekt der Masse γ induziert eine Beschleunigung $\widetilde{a}_{ph}(\tau) := 1/\gamma \cdot \widetilde{F}(\tau)$, die

eine Geschwindigkeitsänderung beschreibt. Die momentane Geschwindigkeit des Objekts zu Zeitpunkt $\tau \in \mathbb{R}$ beträgt dann

$$\widetilde{v}_{ph}(\tau) := \int_0^\tau \widetilde{a}_{ph}(t)\, dt.$$

Eine Simulation dieses Geschwindigkeitsverlaufs erfordert die Berechnung der Stammfunktionen beliebiger Funktionen $\widetilde{a}_{ph}$, was im Allgemeinen nicht möglich und nur in wenigen Ausnahmefällen effizient ist. Es muss also auf numerische Integration zurückgegriffen werden. Mit einem diskretem Kraftverlauf $F : \mathbb{N} \to \mathbb{R}$ ergibt sich mit Hilfe numerischer Integration die Approximation

$$\widetilde{v}_{ph}(\tau) \approx v_{ph}(\tau) := \frac{\tau}{\gamma} \sum_{t=0}^{\tau} c_t F(t).$$

Diese Vorgehensweise erfordert erstens die Berechnung der Gewichte c_t und zweitens die Speicherung aller Werte $F(t)$ für $t \leq \tau$. Eine Abweichung zum physikalischen Modell vereinfacht diese Berechnungen und erreicht hinreichend gute Ergebnisse [FR91]. Wird die Geschwindigkeit gleich der Beschleunigung gesetzt, fällt der Integrationsschritt weg: $v(\tau) := 1/\gamma \cdot F(\tau)$. Betrachtet man den in einem Zeitschritt zurückgelegten Weg Δs ergibt sich dieser direkt aus $v(\tau) = \Delta s/\Delta\tau$ mit $\Delta\tau = (\tau + 1) - \tau = 1$, also $\Delta s(\tau) := 1/\gamma \cdot F(\tau)$.

Ein spezifisches FDL-Verfahren stellt ein Kraftmodell bereit, das heißt eine Sammlung an Gesetzen, deren Wirkung für jeden Zeitschritt $\tau \in \mathbb{N}$ eine Funktion $F^\tau : \mathcal{V} \to \mathbb{R}$ ergeben, mit der ein Knoten $i \in \mathcal{V}$ mittels $\Delta s^\tau(i) = 1/\gamma(i) \cdot F^\tau(i)$ verschoben wird. Für das zu berechnende Layout ergibt sich dann die Beziehung $\mathcal{L}^{\tau+1}(i) = \mathcal{L}^\tau(i) + \Delta s^\tau(i)$. Damit der Algorithmus konvergiert, muss gelten:

$$\forall i \in \mathcal{V} : \lim_{t\to\infty} F^\tau(i) = 0. \tag{3.1}$$

Für $\dim \mathcal{R} > 1$ werden die Berechnung komponentenweise durchgeführt.

3.2.2 Temperatur

Die Diskretisierung der Kraft $\widetilde{F}$ hat zur Folge, dass während eines kontinuierlichen Zeitintervalls $(\tau, \tau + 1]$ die Kraft $F(\tau)$ ungeachtet des virtuellen Systemzustands, der während der Verschiebung durch Wirken des Kraftmodells angenommen würde, die Verschiebung Δs festlegt. Mit einer Skalierung von F um ein $c \in (0, 1)$ und damit einer Verkürzung des Zeitintervalls werden bessere Ergebnisse erzielt. Dies folgt insbesondere aus der Tatsache, dass für $c \to 0$ eine kontinuierliche Krafteinwirkung vorliegt. Dadurch verschiebt sich offensichtlich die Problematik auf die Seite der Laufzeit.

Durch Simulated Annealing wird die Wahrscheinlichkeit einer großen Verschiebung Δs, das heißt einem hohen Kraftbetrag $\|F^\tau\|$, mit dem Fortschreiten der Zeit τ verringert. Dies entspricht dem molekularen Verhalten der Abkühlung eines Metalls. Zunächst stellt eine hohe Temperatur sicher, dass lokale Minima überwunden werden können und am Ende

der Simulation wird das Einstellen eines Kräftegleichgewichts ermöglicht. Eine einfache Umsetzung dieses Konzepts ist die Begrenzung der tatsächliche Kraft durch eine Temperaturfunktion $T : \mathbb{N} \to \mathbb{R}^{\geq 0}$. Den endgültigen Kraftbetrag zum Zeitpunkt $\tau \in \mathbb{N}$ erhält man durch aus dem Minimum von $\|F^\tau\|$ und $T(\tau)$:

$$F_{sa}^{\tau}(i) := \begin{cases} F^{\tau}(i) \cdot \dfrac{T(\tau)}{\|F^{\tau}(i)\|} & \text{für } \|F^{\tau}(i)\| > T(\tau), \\ F^{\tau}(i) & \text{sonst.} \end{cases}$$

Insbesondere ist für $\lim_{t\to\infty} T(\tau) = 0$ Bedingung 3.1 für beliebige F^τ erfüllt. Naiv fällt die Wahl von T auf eine exponentielle Abnahme $T(\tau+1) = c_0 \cdot T(\tau)$ mit $c_0 \in (0,1)$. Ein adaptiver Ansatz wird von Hu [Hu05] vorgeschlagen. Anstatt T stetig zu vermindern, wird diese nur herabgesetzt, falls die Energie E des Systems gesunken ist, das heißt $E(\tau+1) < E(\tau)$. Die Energie bezeichnet den Wert des bereits erwähnten impliziten Funktionals

$$E(\tau) := \sum_{i \in \mathcal{V}} \langle F^{\tau}(i) \rangle ,$$

wobei $\langle p \rangle := \langle p, p \rangle$ für ein $p \in \mathcal{R}$ mit dem Skalarprodukt $\langle ., . \rangle$. Dabei werden die Zeitschritte gezählt, in denen $E(\tau+1) \geq E(\tau)$ gilt. Wird eine Grenze $c_1 \in \mathbb{N}$ erreicht, erfolgt eine Erhöhung $T(\tau+1) = {}^{1}/_{c_0} \cdot T$.

Der iterative Charakter der Verfahren macht die Angabe einer Endbedingung notwendig. Temperaturwert und Energiedifferenz stellen mögliche Kandidaten für die Feststellung des Terminierungszeitpunkts dar. Falls $T < \varepsilon$ für eine kleine Konstante $\varepsilon \in \mathbb{R}^{>0}$, ist gewährleistet, dass nur noch wenig Bewegung stattfindet. Auch wenn die Methode von Hu [Hu05] T gegebenenfalls wieder steigen lässt, falls oft genug ein ungünstigerer Zustand erreicht wird, handelt es sich um ein gutes Maß, da insbesondere in diesem Fall die Energiedifferenz miteinbezogen wird. Bei der naiven Methode ergibt sich jedoch ein Problem, wenn eine gute Energiedifferenz vor Erreichen der Abbruchbedingung oder sogar danach erst angenommen wird, wodurch überflüssige Iterationen durchgeführt werden beziehungsweise das Layout nicht gut genug ist im Sinne der Konstruktion des Kraftmodells. In diesem Fall ist die Energiedifferenz, also $E(\tau+1) - E(\tau) < \varepsilon$ das bessere Abbruchkriterium.

3.3 Ausgewählte Layout-Verfahren

Eine überschaubare Auswahl an Verfahren zeigt bereits, wie unterschiedlich die Herangehensweisen an das Problem des Graph-Layouting sind. Da dieses Problem nur informell beschrieben werden kann, insbesondere auf Grund der subjektiven Natur der Kriterien, können sich die Ergebnisse ebenfalls unterscheiden, von Ähnlichkeit bis zur Unvergleichbarkeit. Die hier vorgestellten Verfahren befinden sich auf der linken Seite dieser Skala, da deren Grundgedanken auf dem Abstand der Knoten untereinander und deren Adjazenz basiert.

Das FDL-Verfahren Fruchterman-Reigold sticht hervor durch seine Einfachheit, die besonders attraktiv für eine Implementierung ist. Dadurch konnten in der Vergangenheit viele Erfahrungen gesammelt werden, die in Optimierungsversuche und -erfolge der Qualität und

umso mehr der Laufzeit eingehen konnten. EER verlässt den Anspruch der Vollständigkeit und reiht sich als FDL-Postprocessing hinter ein beliebiges anderes Verfahren, das die Grundstruktur vorbereitet, um das Layout weiter nach Kriterien zu verbessern, die diese nicht berücksichtigen. Durch eine Abbildung der Problemstellung in die lineare Algebra zeigt sich, dass die Bestimmung eines Layouts auf ein anderes bekanntes Problem hinausläuft: Die Berechnung von Eigenvektoren. ACE setzt diese Idee um und liefert dazu einen Ansatzpunkt für die Umsetzung einer Hierarchisierung. Abschließend führt Kamada-Kawai in ein weiteres Themengebiet der Mathematik. Mittels Analysis wird eine Vorgehensweise abgeleitet, die die Knoten einzeln nacheinander in die richtige Position bringt.

3.3.1 Fruchterman-Reigold

Das Kraftmodell des Verfahrens von Fruchterman und Reingold [FR91] besteht aus zwei Kraftkomponenten. Adjazente Knoten werden jeweils mit einer Feder verbunden, die deren Nähe im Layout gewährleisten soll. Damit sich die Knoten gleichmäßig im Raum verteilen, stoßen sie sich elektrisch ab.

Die anziehende Kraftkomponente, die auf einen Knoten $i \in \mathcal{V}$ auf Grund seiner Adjazenz $\{i, j\} \in \mathcal{E}$ zu $j \in \mathcal{V}$ wirkt, wird abhängig zu deren Verbindungsvektor $\Delta l = \Delta l^{\tau}(i, j) := \mathcal{L}^{\tau}(j) - \mathcal{L}^{\tau}(i)$ im aktuellen Layout $\mathcal{L}^{\tau}$ durch die Funktion

$$F_a(\Delta l) := \frac{\|\Delta l\|^2}{L} \cdot \frac{\Delta l}{\|\Delta l\|}$$

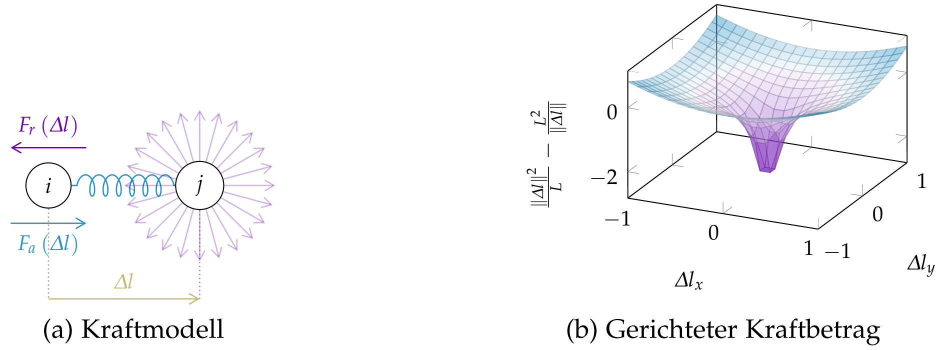
modelliert, wobei die Ruhelänge der Verbindungsfeder als $L \in \mathbb{R}^{>0}$ in der Formel untergebracht wird. Von jedem anderen Knoten $j \in \mathcal{V} \setminus \{i\}$ erfährt i eine abstoßende Kraft ebenfalls abhängig von $\Delta l = \Delta l^{\tau}(i, j)$ und mit Einbezug von L, damit für $\|\Delta l\| = L$ das Gleichgewicht $F_a(L) = F_r(L)$ gilt:

$$F_r(\Delta l) := -\frac{L^2}{\|\Delta l\|} \cdot \frac{\Delta l}{\|\Delta l\|}.$$

Zum Zeitpunkt $\tau \in \mathbb{N}$ wirkt auf i also die Gesamtkraft

$$F^{\tau}(i) := \sum_{\{i,j\} \in \mathcal{E}} F_a(\Delta l^{\tau}(i, j)) + \sum_{j \in \mathcal{V}} F_r(\Delta l^{\tau}(i, j)).$$

Abbildung 3.2a zeigt ein einfaches zweidimensionales Szenario mit $L = 2$. Die wirkenden Kräfte und deren Ursprung sind symbolisch angedeutet, 3.2b visualisiert den Betrag der Gesamtkraft (mit Rücksicht der Parallelrichtung) abhängig der Komponenten von Δl. Neben der höheren Ordnung von $-\|F_r\|$ gegenüber $\|F_a\|$ erkennt man, dass im Zentrum, also an der Position des betrachteten Knotens i, eine Singularität auftritt. Diese ist hier durch eine Begrenzung mit $\varepsilon = 0.1$ aufgelöst.



(a) Kraftmodell (b) Gerichteter Kraftbetrag

Abbildung 3.2: Veranschaulichung des Kraftmodells anhand eines Szenarios

3.3.2 Edge-Edge Repulsion

Lin und Yen [LY12] haben es sich zur Aufgabe gemacht, den Grad der Symmetrie eines bereits durch ein anderes Verfahren berechneten Layouts weiter zu erhöhen, dabei gleichmäßige Kantenlängen zu erhalten und überlappende Kanten zu verhindern. Wie auch bei Fruchterman-Reigold erfolgt durch die Adjazenz eines Knotens $i \in \mathcal{V}$ eine Anziehungskraft abhängig von $\Delta l = \Delta l^{\tau}(i,j)$. Allerdings kommt eine ältere Version wieder zum Einsatz, die Fruchterman und Reingold [FR91] zu ihrer Zeit wegen der Zeitkomplexität der Berechnung des Logarithmus vereinfacht hatten:

$$F_a(\Delta l) := c_1 \log\left(\frac{\|\Delta l\|}{c_2}\right) \frac{\Delta l}{\|\Delta l\|}$$

mit den Konstanten $c_i \in \mathbb{R}$. Die abstoßende Kraft unterscheidet sich grundsätzlich von den herkömmlichen Methoden, da nicht die Position der Knoten, sondern die Länge und der eingeschlossene Winkel zwischen Kanten gleichen Ursprungs eine Abstoßungskraft induzieren. Für jeden Knoten $i \in \mathcal{V}$ mit $\deg i > 1$ werden alle inzidenten Kanten $\{i,j\} \in \mathcal{E}$ jeweils zusammen mit ihrem nächsten Nachbar $\{i,k\} \in \mathcal{E}$ betrachtet, genauer deren Vektorrepräsentationen $a = \Delta l^{\tau}(i,j)$ und $b = \Delta l^{\tau}(i,k)$. Jedes Paar trägt mit

$$F_p(a,b) := (F_e(a,b) + F_\theta(a,b)) \cdot F_d(a,b)$$

zur abstoßenden Kraft bei, deren Richtung mittels

$$F_d(a,b) = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \frac{a+b}{\|a+b\|}$$

bestimmt wird. Der Kraftbetrag richtet sich nach der Länge der Kante (F_e) und dem eingeschlossenen Winkel θ (F_θ):

$$\begin{aligned} F_e(a,b) &= c_3\left(\arctan\left(\frac{\|a\|}{c_4}\right) + \arctan\left(\frac{\|b\|}{c_4}\right)\right), \\ F_\theta(a,b) &= c_5 \cot\underbrace{\left(\arccos\left(\frac{a\,(a+b)}{\|a\|\,\|a+b\|}\right)\right)}_{\theta/2}. \end{aligned}$$

Zum Zeitpunkt $\tau \in \mathbb{N}$ wirkt auf i also die Gesamtkraft

$$F^{\tau}(i) := \sum_{\{i,j\} \in \mathcal{E}} F_a\left(\Delta l^{\tau}(i,j)\right) + \sum_{\{a,b\} \in \mathcal{E}^{\tau}(i)} F_p(a,b),$$

wobei $\mathcal{E}^{\tau}(i)$ die Vektorrepräsentationen aller benachbarten Kantenpaare enthält. Dieses Kraftmodell kann zur Folge haben, dass wiederum die Knotenpositionen aufeinander fallen. Ein hybrider Ansatz aus Kanten- und Knotenabstoßung verschafft Abhilfe.

3.3.3 Algebraic Multigrid Computation of Eigenvectors

Das Verfahren von Koren, Carmel und Harel [KCH03] basiert auf folgender Energie

$$E := \frac{1}{2} \sum_{\{i,j\} \in \mathcal{E}_u} \omega(i,j) \langle \Delta l(i,j) \rangle .$$

Sie kann dazu verwendet werden, die Qualität eines Layouts $\mathcal{L}$ zu messen. Eine große Distanz zwischen adjazenten Knoten $i, j \in \mathcal{V}$ führt zu einer hohen Teilenergie, die durch das Gewicht $\omega(i,j)$ skaliert in die Gesamtenergie einfließt. Die Überführung in einen Energievektor η, sodass $E = \sum_{d=0}^{k} \eta_d$ mit $k = \dim \mathcal{R} - 1$ und

$$\eta_d := \frac{1}{2} \sum_{\{i,j\} \in \mathcal{E}_u} \omega(i,j) \left(\Delta l_d(i,j)\right)^2 ,$$

ermöglicht es, jede Teilenergie η_d mit der quadratische Form der Laplace-Matrix L von $\mathcal{G}$ auszudrücken. Es gilt also $\eta_d = x^T L x$ mit $x_i = \mathcal{L}_d(i)$ für $i = 0, \ldots, n-1$. Neben der trivialen Lösung $x_i = c$ für ein $c \in \mathbb{R}$ minimiert v_1 die Teilenergie mit $\eta_d = \lambda_1$, wobei $\lambda_0 = 0 < \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_{n-1}$ die Eigenwerte von L mit den zugehörigen Eigenvektoren $v_0 = c \cdot (1, \ldots, 1)^T, v_1 \ldots, v_{n-1}$. Da für $\mathcal{L}(i) = (v_{1,i}, \ldots, v_{1,i})^T$ alle Knoten auf einer Linie verteilt liegen, werden für $\dim \mathcal{R} > 1$ weitere Eigenvektoren $v_2, \ldots, v_k$ benötigt, sodass $\mathcal{L}(i) = (v_{1,i}, v_{2,i}, \ldots, v_{k,i})^T$ mit $k = \dim \mathcal{R}$. Ziel dieses Verfahrens ist die Lösung dieses Eigenprojektionsproblems (EPP), also die Berechnung der Eigenvektoren der $\dim \mathcal{R}$ kleinsten Eigenwerte.

Das EPP bleibt jedoch im Allgemeinen in einer beliebigen Hierarchiefolge $\mathcal{G}^0 = \mathcal{G}, \mathcal{G}^1, \ldots, \mathcal{G}^c$ nicht erhalten. Eine Generalisierung des Problems auf PSD-Graphen, das generalisierte EPP (GEPP), erfüllt die gewünschte Eigenschaft.

Iterative Berechnung von Eigenvektoren

Für die Berechnung der Eigenvektoren kommt ein beliebiges numerisches Iterationsverfahren in Frage, dessen Konvergenzverhalten bei guten Startwerten schnell zu einer Approximation mit geringem Fehler führt.

Code 3.a: Potenzmethode [KCH03]

```
1  function PowerIteration({s_1,...,s_k}, L, M, ε)
2  {
3    u_1 = M^{1/2} · (1,...,1)^T; u_1 = u_1 / ‖u_1‖; // Erster Eigenvektor
4    B = ρ(M^{-1/2} L M^{-1/2});
5
6    for (i = 1; i ≤ k; i++)
7    {
8      p_i = M^{1/2} s_i; p_i = p_i / ‖p_i‖;
9
10     do
11     {
12       u_i = p_i;
13
14       for (j = 0; j < i; j++) // Orthogonalisierung zum vorherigen Vektor
15       {
16         u_i = u_i − (u_i^T u_j) u_j;
17       }
18
19       p_i = B · u_i; p_i = p_i / ‖p_i‖; // Iteration
20     }
21     while (p_i · u_i ≤ 1 − ε); // Bis Richtungsänderung vernachlässigbar
22
23     u_i = p_i; v_i = M^{-1/2} u_i;
24   }
25
26   return v_1,...,v_k;
27 }
```

Ein Beispiel für einen solches Verfahren ist die Potenzmethode, die die Eigenvektoren der zugehörigen größten Eigenwerte von symmetrischen Matrizen berechnet. Zu lösen ist das Eigenwertproblem $Lv = \mu Mv$, das sich in die übliche Form $Bu = \lambda u$ überführen lässt mit $\lambda = \mu$, $B = M^{-1/2} L M^{-1/2}$ und $u = M^{1/2} v$. Für eine Diagonalmatrix D und ein $c \in \mathbb{R}$ ist D^c definiert als die Potenzierung der einzelnen Einträge $D^c_{i,i}$. Die Potenzmethode liefert Eigenvektoren einer Matrix in absteigender Reihenfolge der Eigenwerte. Da die Lösung des GEPP die umgekehrte Reihenfolge verlangt, wird die Matrix B entsprechend modifiziert. Die Abbildung

$$\rho(A) := \max_{i \in \mathcal{V}} \left(A_{i,i} + \sum_{j \neq i} |A_{i,j}| \right) \cdot I - A$$

bildet eine Matrix $A \in \mathbb{R}^{n \times n}$ derart ab, dass ihr Bild durch dieselben Eigenvektoren charakterisiert wird, jedoch in umgekehrter Reihenfolge der Eigenwerte. Code 3.a zeigt den Pseudocode für die modifizierte Potenzmethode, wobei $k = \dim \mathcal{R}$.

Interpolationsmatrix

Neben der Vorstellung des Hierarchisierungskonzepts von ACE dienen folgende Ausführungen einer Vertiefung zum Konzept der Hierarchisierung, das in Abschnitt 3.1 bereits

eingeführt wurde. Der angesprochene Ansatzpunkt für ebendiese wird mit Hilfe einer Interpolationsmatrix $P \in \mathbb{R}^{n \times n'}$ mit $n > n'$ realisiert. Dabei müssen für P folgende Bedingungen gelten:

1. Die Einträge von P sind positiv: $\forall i \leq n, j \leq n' : P_{i,j} \geq 0$,
2. Die Zeilensumme von P ergibt Eins: $\forall i \leq n : \sum_{j=0}^{n'-1} P_{i,j} = 1$,
3. P hat vollen Rang: $\text{rk}\,(P)$.

Bedingungen 1 und 2 beschreiben, wie aus den n' Positionen von $\mathcal{L}^{h+1}$ die n Positionen von $\mathcal{L}^h$ interpoliert werden. Dabei entspricht die Multiplikation mit P der gewichteten Summe

$$\mathcal{L}_d^h(i) = \sum_{j=0}^{n'-1} P_{i,j} \mathcal{L}_d^{h+1}(j)$$

für $d = 0, \ldots, \dim \mathcal{R} - 1$. Bedingung 3 stellt sicher, dass unterschiedliche Layouts durch Interpolation nie das selbe Layout ergeben. Die folgenden zwei Gleichungen beschreiben den Zusammenhang zweier Abstraktionsebenen $h+1$ und h:

$$\begin{aligned} L^{h+1} &= P^T L^h P, \\ M^{h+1} \cdot (1, \ldots, 1)^T &= P^T M^h P \cdot (1, \ldots, 1)^T, \end{aligned} \tag{3.2}$$

wobei Gleichung 3.2 als komponentenweise Berechnung

$$M_{j,j}^{h+1} = \sum_{i=0}^{n-1} P_{i,j} M_{i,i}^h$$

verstanden werden kann.

Zu den Bedingungen, die P erfüllen muss, kommen auch widersprüchliche Eigenschaften, die P besitzen sollte, da sie den Kern einer effektiven Hierarchisierung ausmacht:

1. Die Interpolation des Layouts $\mathcal{L}^{h+1}$ sollte die Minimaleigenschaft der zugehörigen Energie E so gut wie möglich weitergeben, damit die Potenzmethode mit einer geringen Anzahl an Iterationen zu einer akzeptablen Lösung kommt.
2. Die Berechnung von P sollte offensichtlich effizienter als die direkte Berechnung der Lösung sein, da sonst kein Gewinn davon ausgeht.
3. P sollte eine dünn besetzte Matrix sein (vergleiche dünn besetzter Graph), da für die Berechnung von L^{h+1} zwei Matrixmultiplikationen mit P durchgeführt werden müssen.

Mit Hilfe von einer Berechnungsvorschrift für P wird wie folgt vorgegangen, um das Layout eines Graphen zu berechnen. Gegeben ist ein Graph der Form $\mathcal{G} = (L, M)$. Beginnend mit $h = 0$ wird die Hierarchiefolge $\mathcal{G}^0 = \mathcal{G}, \mathcal{G}^1, \ldots, \mathcal{G}^c$ aufgestellt, indem jeweils P^{h+1} berechnet wird und deren Anwendung auf L^h, M^h zu kleineren Matrizen L^{h+1}, M^{h+1} führt. Dies wird so oft wiederholt, bis $\text{rk}\, L^c$ klein genug ist für eine direkte Berechnung der Eigenvektoren $v_1^c, \ldots, v_k^c$ mit $k = \dim \mathcal{R}$. Diese werden mittels der Potenzmethode mit zufälligen Startwerten $s_1^c, \ldots, s_k^c$ bestimmt. Rückwärts, von $h = c$ an, werden die berechneten $v_1^h, \ldots, v_k^h$ mit P^h interpoliert und auf der nächsten Ebene $h-1$ als Startwerte eingesetzt, also $s_1^h, \ldots, s_k^h = P^h v_1^h, \ldots, P^h v_k^h$. Nach der letzten Iteration auf Ebene $h = 0$ ist das Layout von $\mathcal{G}$ aus $v_1^0, \ldots, v_k^0$ bestimmbar.

3.3.4 Kamada-Kawai

Der Algorithmus, den Kamada und Kawai [KK89] vorstellen, vereint die Idee die Eigenschaften von Federn auszunutzen (Fruchterman-Reigold) und die Minimierung einer daraus entstehenden Energiefunktion (ACE). In diesem Modell sind alle Knoten $i \in \mathcal{V}$ eines ungerichteten Graphen $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \omega)$ gegenseitig mit einer Feder verbunden. Für dieses System ergibt sich die Energie

$$E(\mathcal{L}) := \frac{c}{2} \sum_{i,j \in \mathcal{V}, i \neq j} \frac{\|\Delta l(i,j)\| - L \cdot \Delta p(i,j)}{\Delta p^2(i,j)},$$

wobei $c \in \mathbb{R}^{>0}$ eine Konstante, $L \in \mathbb{R}^{>0}$ die Ruhelänge der Federn und $\Delta p(i,j) \in \mathbb{R}$ die Länge des kürzesten Pfades zwischen i und j. Die Kanten spielen also nur eine Rolle bei der einmaligen Bestimmung von Δp. Die notwendige Bedingungen, um E zu minimieren, sind in $\mathcal{R} = \mathbb{R}^2$

$$\frac{\partial E}{\partial x_i} = \frac{\partial E}{\partial y_i} = 0$$

für $i = 0, \ldots, n-1$, wobei $n = |\mathcal{V}|$ und $(x_i, y_i) = \mathcal{L}(i)$. Eine Lösung dieser $2n$ nichtlinearen Gleichungen mit einem iterativen Lösungsverfahren ist erst möglich, wenn die Gleichungen voneinander entkoppelt werden. Die Positionen der Knoten $j \in \mathcal{V} \setminus \{i\}$ werden also fixiert, sodass die nur noch von $\mathcal{L}(i)$ abhängende Funktion $\eta_i(x_i, y_i) = E(\mathcal{L}(0), \ldots, (x_i, y_i), \ldots, \mathcal{L}(n-1))$ minimiert werden kann.

Den Knoten $i \in \mathcal{V}$ wird zu Beginn eine Position $\mathcal{L}^0(i)$ zugeordnet, die einem Eckpunkt eines regelmäßigen Polygons entspricht. In jedem Iterationsabschnitt fällt die Wahl auf den Knoten $k = \operatorname{argmax}_{i \in \mathcal{V}} \|\nabla \eta_i(\mathcal{L}^\tau(i))\|$, wobei $\nabla \eta_i = (\partial\eta_i/\partial x, \partial\eta_i/\partial y)^T$. Mit der Übergangsvorschrift $\mathcal{L}^{\tau+1}(k) = \mathcal{L}^\tau(k) + \Delta s^\tau$ wird die Position von k immer weiter verbessert, bis $\nabla \eta_k(\mathcal{L}^\tau(k))$ klein genug wird. Die Bestimmung von Δs^τ erfolgt durch Lösen folgendes linearen Gleichungssystems: $J^k(\mathcal{L}^\tau(k)) \cdot \Delta s^\tau = -\nabla \eta_k(\mathcal{L}^\tau(k))$ mit der Jacobi-Matrix

$$J^k_{i,j} := \begin{pmatrix} \frac{\partial^2 \eta_k}{\partial x_k^2} & \frac{\partial^2 \eta_k}{\partial x_k \partial y_k} \\ \frac{\partial^2 \eta_k}{\partial y_k \partial x_k} & \frac{\partial^2 \eta_k}{\partial y_k^2} \end{pmatrix},$$

deren Einträge sich durch vorberechnete partielle Ableitungen bestimmen lassen.

4 Verwandte Arbeiten

Das Potential der massiv parallelen Grafikhardware, besonders aufwändige Berechnungen beschleunigen zu können, hat bereits einige wissenschaftliche Arbeiten im Zusammenhang des Graph-Layouting hervorgebracht. Zwei dieser bis ins Detail ausgeklügelten Konzepte verdeutlichen, dass es bereits ein großes Repertoire an Verfahren gibt, die isoliert voneinander Probleme des Graph-Layouting lösen. Einige davon kommen aus übergreifenden Themengebieten, wie der Graphen-Theorie oder der Computergrafik. Der Fokus der beiden Arbeiten liegt dabei auf der effizienten Bestimmung eines Layouts von hoher Qualität. Damit rückt in den Hintergrund, wie das gefundene Layout letztendlich visualisiert werden kann. Diese Frage muss jedoch spätestens beantwortet werden, wenn Graph-Drawing als Teil einer Software umgesetzt wird. Der dritte Teil dieses Kapitels befasst sich mit praktischen Umsetzungen in Software, bei denen die theoretischen Aspekte und Konzepte im Quellcode verborgen sind. Die Verlagerung von Berechnungen auf die Grafikhardware hat dort, wie es scheint, noch keinen Einzug gefunden.

4.1 Mehrstufiges Graph-Layouting auf der GPU

Frishman und Tal [FT07] präsentieren einen besonders komplexen Ansatz, der eine mehrstufige Hierarchisierung des Graphen vornimmt, um einerseits das Konvergenzverhalten der numerischen Verfahren auszunutzen und andererseits effiziente Speicherzugriffe zu ermöglichen.

Im Wesentlichen besteht der entwickelte Algorithmus aus drei Phasen. Zu Beginn wird mit einigen wenigen Abstraktionsschritten eine strukturelle Hierarchiefolge des Graphen angelegt. Der letzte Graph dieser Folge wird weiter rekursiv in spektrale Partitionen unterteilt, die die zweite Hierarchie aufstellen. Auf jeder dieser Ebenen wird von unten nach oben ein Layout bestimmt, das durch Interpolation nach oben weitergereicht wird. Dabei erfolgt die dritte Hierarchisierung durch eine räumliche Partitionierung, die das Speicherlayout definiert und für approximative Zwecke eingesetzt wird.

Die erste Abstraktionsstufe wird wie folgt bestimmt. Aus gegebenem Graph $\mathcal{G} = (\mathcal{E}, \mathcal{V}, \gamma, \omega)$ mit $\forall i \in \mathcal{V} : m(i) = 1$ wird eine Hierarchiefolge $\mathcal{G}^0 = \mathcal{G}, \mathcal{G}^1, \ldots, \mathcal{G}^c$ angelegt, indem für jeden Knoten $i \in \mathcal{V}^h$ in der Reihenfolge des Grads $\deg i$ die Kante

$$\{i, j\} = \underset{\{p,q\} \in \mathcal{E}^k}{\operatorname{argmax}} \left(\frac{\omega(p,q)}{\gamma(p)} + \frac{\omega(p,q)}{\gamma(q)} \right)$$

entfernt wird und somit die zwei Knoten $i, j \in \mathcal{V}^h$ zu einem neuen Knoten $\{i, j\} = i' \in \mathcal{V}^{h+1}$ mit $\gamma^{h+1}(i') = \gamma^h(i) + \gamma^h(j)$ und

$$\forall j' \in \mathcal{V}^{h+1} : \omega^{h+1}(i', j') = \sum_{p \in i', q \in j'} \omega^h(p, q)$$

verschmelzen.

Der Arbeitsgraph $\mathcal{G}^c$ wird weiter zu einer Folge von spektralen Partitionsgraphen $S^0, \ldots, S^f$ unterteilt. Begonnen wird bei S^0 mit $\mathcal{V}(S^0) = \{P_0^0 = \mathcal{G}^c\}$ und $\mathcal{E}(S^0) = \emptyset$. Durch die Anwendung von ACE wird der Eigenvektor $v_1 \in \mathbb{R}^n$ jedes Subgraphen P_k^l mit $n = |\mathcal{V}(P_k^l)|$ bestimmt. Die Einträge $v_{1,i}$ werden nach $\deg i$ sortiert, sodass jeder Knoten $i \in \mathcal{V}(P_k^l)$ entsprechend der Position seines Eintrags einer von m gleichgroßen Partitionen $P_0^{l+1}, \ldots, P_{m-1}^{l+1}$ zugeordnet werden kann. Für $m > 2$ können isolierte Partitionen vorliegen, die anschließend nach und nach jeweils mit der größeren Nachbarpartition verschmolzen werden. Die gefundenen Partitionen ergeben die Knotenmenge $\mathcal{V}(S^{l+1})$, die durch akkumulierte Kanten $\mathcal{E}(S^{l+1})$ verbunden werden, die sich aus den Kanten zwischen den Graphen ergeben.

Beginnend bei S^0, dessen triviales Layout ohne Rechnung folgt, werden die Knotenpositionen von $\mathcal{L}_S^{l+1}$ aus dem bereits berechneten Layout $\mathcal{L}_S^l$ interpoliert. Nach der Initialisierung $\mathcal{L}_S^{l+1}(i) = \mathcal{L}_S^l(i')$ mit $i \in \mathcal{V}(S^{l+1})$ und $i \subset i' \in \mathcal{V}(S^l)$ wird das Layout skaliert mittels

$$\mathcal{L}_S^{l+1}(i) = \sqrt{\frac{|\mathcal{V}(S^{l+1})|}{|\mathcal{V}(S^l)|}} \cdot \mathcal{L}_S^{l+1}(i).$$

Anschließend werden die Knoten von S^{l+1} zwischen ihrer aktuellen Position und der durchschnittlichen Position der adjazenten Knoten iterativ nach folgender Vorschrift positioniert:

$$\mathcal{L}_S^{l+1}(i) = \frac{1}{2}\left(\mathcal{L}_S^{l+1}(i) + \frac{1}{\deg i} \sum_{\{i,j\} \in \mathcal{E}(S^{l+1})} \mathcal{L}_S^{l+1}(j)\right).$$

Der Partitionsgraph S^{l+1} wird räumlich, ähnlich der Konstruktion eines k-d-Baumes, erneut in Partitionen P_k unterteilt, deren Zugehörigkeit für jeden Knoten $i \in \mathcal{V}$ mittels $P(i)$ abgefragt werden kann. Außerdem wird für jede Partition ein Gravitationszentrum $C(P_k) \in \mathcal{R}$ berechnet, bevor der k-d-Baum zusammen mit $\mathcal{L}_S^{l+1}$ als Initial-Layout $\mathcal{L}_F^0$ an eine Modifikation von Fruchterman-Reigold übergeben wird. Dabei wird die Gesamtkraft durch

$$F^\tau(i) := \sum_{\{i,j\} \in \mathcal{E}_u} F_a(\Delta l^\tau(i,j)) + \sum_{j \in P(i)} F_r(\Delta l^\tau(i,j)) + \sum_{P_k \neq P(i)} F_r(\Delta c^\tau(i, P_k))$$

berechnet mit $\Delta c^\tau(i, P) := C(P) - \mathcal{L}_F^\tau(i)$. Die neue Berechnungsvorschrift bezieht nur noch Knoten innerhalb der gleichen Partition $P(i)$ direkt ein und approximiert Knoten anderer Partitionen $P_k \neq P(i)$ durch deren Gravitationszentrum $C(P_k)$. Der k-d-Baum wird für jede Iteration erneut berechnet. Für genügend kleine Graphen kommt alternativ ohne die Partitionierung der zeitaufwändigere aber robustere FDL-Algorithmus Kamada-Kawai zum Einsatz. Mit demselben Verfahren wird von $\mathcal{L}_S^0$ als $\mathcal{L}^c$ fortgeschritten, sodass letztendlich ein

Layout für $\mathcal{G}^1$ vorliegt. Der letzte Schritt wird auf die Interpolation reduziert, um wertvolle Zeit bei der Berechnung des finalen Layouts $\mathcal{L}^0$ von $\mathcal{G}$ zu sparen.

Die Layout-Iterationen werden durch Auslagern parallelisierbarer Berechnungen auf die GPU beschleunigt. Dies sind im Einzelnen die Bestimmung des Gravitationszentrums $C(P_k)$, der wirkenden Kräfte $F^\tau(i)$ und der tatsächlichen Verschiebung $\Delta s^\tau(i)$. Dabei wird für jede abgeschlossene Berechnung, die jeweils für eine räumliche Partition zuständig ist, eine Rendering-Pipeline gestartet, die mit Texturen als Ein- und Ausgabe arbeitet.

Wie die Speicherverwaltung mittels Texturen realisiert ist, wird hier kurz beschrieben. In der `locations`-Textur wird die aktuelle Position und die Partitionszugehörigkeit eines jeden Knotens gespeichert. Dabei liegen Daten der gleichen Partition in rechteckigen, zusammenhängenden Speicherbereichen, um die Lokalität der Daten zu gewährleisten. Die Adjazenz wird in zwei Texturen `adjacency` und `neighbors` gehalten. Letztere enthält für jeden Knoten dessen Grad und den Zeiger auf den ersten Adjazenzeintrag in `adjacency`, in der nach Quellknoten sortierte Zeiger auf die `locations`-Textur abgelegt sind. Das Ergebnis der Partitionierung durch einen k-d-Baum wird in ebenfalls zwei Texturen verwaltet. Die `partition information`-Textur listet für jede Partition einen Zeiger auf den ersten enthaltenen Knoten, Breite und Höhe des Speicherbereichs, Anzahl der Knoten in der letzten Zeile und insgesamt. Daneben wird das jeweilige Gravitationszentrum in einer weiteren Textur `partition center of gravity` gespeichert.

Die Kraftberechnung findet getrennt statt, sodass jeweils Programm und Daten in abstoßend und anziehend unterteilt sind, bevor sie zusammen in die Berechnung der tatsächlichen Verschiebung eingehen. Da Texturen nicht gleichzeitig geschrieben und gelesen werden können, ist für das Endergebnis eine separate Textur für die neuen Positionen nötig, bevor diese als `locations`-Textur erneut verarbeitet werden können. Dies erklärt auch die Trennung der beiden Partititonstexturen, die bei der Berechnung der Gravitationszentren in die Rollen Ein- beziehungsweise Ausgabetextur schlüpfen.

4.2 Schnelles Zeichnen von Graphen auf der GPU mit Hilfe von Multipolentwicklung

Der Algorithmus von Godiyal u. a. [God+09] ist einfacher gestrickt. Im Gegensatz zu Frishman und Tal [FT07] beschränkt sich das Hierarchisierungskonzept auf die strukturelle und die räumliche Abstraktion. Für die Approximation entfernter Knoten wird der physikalische Aspekt von Fruchterman-Reigold ausgenutzt. Mit Hilfe der sogenannten Multipolentwicklung werden hier nicht Partitionen gleicher Größe betrachtet, sondern stets der gesamte gegenüberliegende Teilbaum.

Zu Beginn wird eine Folge von Graphen $\mathcal{G}^0 = \mathcal{G}, \mathcal{G}^1 \ldots, \mathcal{G}^c$ durch die Bestimmung der Knotenmengen $\mathcal{V}^0 = \mathcal{V} \supset \mathcal{V}^1 \supset \ldots \supset \mathcal{V}^c$ aufgestellt, sodass jede Teilmenge $\mathcal{V}^{h+1} \subset \mathcal{V}^h$ eine unabhängige Menge von $\mathcal{G}^h$ ist, das heißt $\forall i \in \mathcal{V}^{h+1} : \{i, j\} \notin \mathcal{E}^h$. Das NP-vollständige Problem kann durch die folgende 2-Appoximation gelöst werden: Um eine unabhängige Menge $\mathcal{V}^{h+1} \subset \mathcal{V}^h$ zu finden, werden solange Knoten $i \in \mathcal{V}^h$ und deren Nachbarschaft $\{i, j\} \in \mathcal{E}^h$ entfernt, bis $\mathcal{V}^h = \emptyset$. Erstere ergeben zusammen die unabhängige Menge $\mathcal{V}^{h+1}$.

Von der untersten Ebene $h = c$ an wird das Layout $\mathcal{L}^{h+1}$ des Graphen $\mathcal{G}^{h+1}$ mit Hilfe derselben Interpolation wie in Abschnitt 4.1 zunächst aus $\mathcal{L}^h$ bestimmt, um dann mit Fruchterman-Reigold verfeinert zu werden. Dabei wird die anziehende Kraft F_a modifiziert zu

$$F_a(\Delta l) = \|\Delta l\|^2 \log\left(\frac{\|\Delta l\|}{L}\right) \cdot \frac{\Delta l}{\|\Delta l\|}.$$

Auch hier kommt eine Approximation weit entfernter Knoten zum Einsatz, die mit Hilfe eines modifizierten k-d-Baumes realisiert wird. Allerdings wird dieser nur die ersten vier und dann jede zwanzigste Iteration rekonstruiert. Die Ebenen der zwei Diskriminanten x, y werden zusammengefasst, sodass jeder Baumknoten vier Kinder erhält. Außerdem wird die Hierarchisierung abgebrochen, falls nur noch vier Knoten in einer Partition übrig sind. Es wird folgende endliche Multipolentwicklung verwendet, um die kumulative Kraft der Knoten, die außerhalb der jeweiligen Partition liegen, zu berechnen:

$$\Phi(z) \approx \sum_{i=0}^{m-1} q_i \cdot \log(z - z_0) - \sum_{k=0}^{3} \frac{a_k}{(z - z_0)^k},$$

mit $a_k = \frac{1}{k} \sum q_i (z_i - z_0)^k$, wobei $z_i \in \mathbb{C}$ Positionen von m Ladungen $q_i \in \mathbb{R}$ in der gaußschen Zahlenebene darstellen. Dieses Vorgehen beschränkt sich also auf $\mathcal{R} = \mathbb{R}^2$. Die Positionen spannen einen Kreis $K = (z_0, r)$ mit Zentrum z_0 und Radius $r = \max_{i=0,\dots,m-1} |z_i - z_0|$ auf. Für ein $z \in \mathbb{C}$, das außerhalb dieses Kreises liegt, also $|z - z_0| > r$, wirken die Ladungen q_i gemeinsam als Potential $\Phi(z)$, das sich direkt als Kraftwirkung

$$F_\Phi(z) = \begin{pmatrix} -\mathrm{Re}(\Phi'(z)) \\ \mathrm{Im}(\Phi'(z)) \end{pmatrix}$$

ausdrücken lässt. Da jedem Baumknoten vier Kinder zugeordnet sind, die tatsächliche Knoten $i \in \mathcal{V}$ repräsentieren, können für jeden dieser Baumknoten die Koeffizienten a_k für $k = 0, \dots, 3$ während der Konstruktion des k-d-Baums aus den Ladungen q_k berechnet werden, sodass anschließend für ein beliebiges $z \in \mathbb{C}$ mit Hilfe dieser Hilfskonstruktion wie folgt Kräfte akkumuliert werden. Begonnen wird bei der Wurzel. Für alle vier Geschwister $k = 0, \dots, 3$ wird jeweils überprüft, ob z außerhalb des Kreises $K^k = (z_0^k, r^k)$ liegt. Ist dies der Fall, also $|z - z_0^k| > r^k$, wird die abstoßende Kraft $F_r^k(z)$ mit der Multipolentwicklung F_Φ^k approximiert. Für einen der vier Knoten wird dies nicht der Fall sein und die Prozedur wird erneut für seine Kinder durchgeführt, bis der Blattknoten mit $j \in \mathcal{V}$ erreicht wird, an dem die tatsächliche Kraft $F_r(i, j)$ berechnet werden kann.

Bei der Konstruktion des k-d-Baumes wird für große $n = |\mathcal{V}|$ die GPU eingesetzt, um die Berechnungen der Mediane zu beschleunigen. Die Bestimmung der Kräfte erfolgt ebenfalls parallelisiert, indem für jeden Knoten ein Thread auf der GPU gestartet wird, um die jeweilige Kraft zu berechnen.

Die notwendigen Daten werden hier ebenfalls in Texturen gehalten. Dazu gehören zwei Arrays, wovon eines die Knoten listet, die über entsprechende Indizes mit Speicherbereichen des zweiten assoziiert werden, das die Kanten sortiert nach Quellknoten enthält. Außerdem wird der k-d-Baum für die GPU so verfügbar gemacht, dass eine Traversierung ohne Stapelspeicher möglich ist.

4.3 Software

Es gibt eine Reihe von Möglichkeiten, wenn es darum geht, Graphen mittels vorhandener Software zu visualisieren. Unter diesen befinden sich Programmbibliotheken wie iGraph [lib], Tulip [AM] oder Microsoft Automatic Graph Layout (MSAGL) [Mic]. Desweiteren gibt es Plattformen wie Prefuse [Pre], Graphviz [Res] oder Gephi [Gep]. Es gibt sogar eine Erweiterung von Microsoft Excel, NodeXL [Tea], und Wolfram Mathematica bietet ebenfalls eine Layout-Funktionalität an [Wol]. Mit einer Spezialisierung auf dynamische Graphen, fällt die Software Ubigraph [Ubi] ein wenig aus der Reihe.

Interessant im Kontext dieser Arbeit ist das Gephi-Plugin OpenOrd Layout, das einen parallelisierten Fruchterman-Reigold mit einem sehr speziellen Simulated-Annealing-Schema verwendet. Die Parallelisierung erfolgt durch die Aufteilung der Knoten, sodass ein Thread jeweils für eine Untermenge der Knotenmenge zuständig ist.

Eine Untersuchung der verschiedenen Software für die Visualisierung von Graphen mit einer anschließenden Evaluation könnte von Interesse sein. Dies liegt allerdings außerhalb des zeitlichen Rahmens dieser Arbeit.

5 Konzept eines Frameworks zum Zeichnen von Graphen

Wie können Layout-Verfahren auf die GPU verlagert werden? Welche Möglichkeiten gibt es, Knoten und Kanten darzustellen und wie kann diese Visualisierung mittels einer Rendering-API, nämlich OpenGL, umgesetzt werden? Dieses Kapitel gibt Antworten auf diese und weitere Fragen, die im Zusammenhang des Graph-Drawings gestellt werden können.

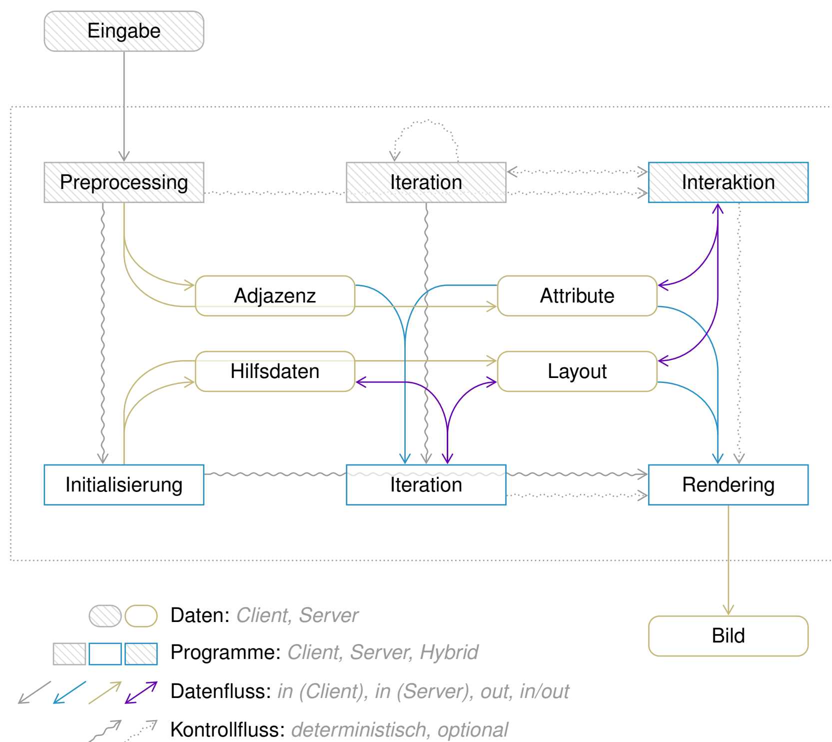


Abbildung 5.1: Schematischer Aufbau der Graph-Drawing-Pipeline

Abbildung 5.1 zeigt den schematischen Aufbau der entwickelten Graph-Drawing-Pipeline, die als Einstiegspunkt für die nachfolgenden Abschnitte dient. Am Anfang der Pipeline steht die Eingabe eines Graphen in Form einer standardisierten Datenstruktur, die aus einer Datenbank (unter anderem auch formatierte Textdateien) oder einer künstlichen Generierung gewonnen werden kann. Das Ergebnis besteht aus einem Bild, das die Visualisierung des Graphen darstellt.

Im Zentrum der Abbildung befindet sich das Herzstück des Frameworks. Die Daten, die im

Hauptspeicher der GPU gehalten werden, sind in vier Kategorien unterteilt. Eine statische Adjazenzliste ermöglicht eine Traversierung des Graphen, Knoten- und Kantenattribute erlauben eine gezielte Steuerung der Visualisierung und Hilfsdaten unterstützen unter anderem die Bestimmung des Knotenlayouts, das sich ebenfalls auf der GPU befindet. Wie bereits zu Beginn von Kapitel 3 erläutert, müssen die Eingabedaten vorverarbeitet werden. Dazu kommt eine Optimierung hinsichtlich des Server-seitigen Speicherzugriffs. Anschließend erfolgt eine Initialisierung der GPU-zentrischen Daten und die erste Darstellung der aktuellen Szene. Ein wichtiger Aspekt des Frameworks ist der Einbezug von Interaktion. So entscheidet der Benutzer über den weiteren Kontrollfluss. Darunter auch über die wiederholte Anwendung von Layout-Iterationen, die, falls gewünscht, einzeln auf der Anzeige erscheinen, was bei genügend hoher Framerate den Eindruck einer Videosequenz erweckt.

Damit verbunden ist die Idee der Entkoppelung des Iterations- und Rendering-Prozesses, sodass unabhängig der unter Umständen aufwändigen Berechnungen jederzeit eine Szene, nämlich die letzte konsistente, dargestellt werden kann. Es wird also mindestens Double-Buffering des Layouts benötigt. Der Front-Buffer enthält das konsistente Layout, wohingegen das Layout-Verfahren die neuen Positionen im Back-Buffer ablegt. Bei FDL-Algorithmen bietet es sich an, die berechneten Kräfte abzuspeichern, sodass die "Vertauschung" der Buffer durch simples Verschieben der Positionen anhand der berechneten Kräfte durchgeführt wird. Eine wahre Entkopplung ist nur möglich, wenn auf Client-Seite eine Parallelisierung stattfindet und zwei verschiedene OpenGL-Umgebungen Zugriff auf den gleichen Speicher haben. Für ineffiziente Layout-Verfahren oder besonders hohe Eingabegrößen kann die oben eingeführte Framerate nicht eingehalten werden. Die Entkopplung bietet zusammen mit einem weiteren Positionspuffer die Möglichkeit zwischen zwei Layouts zu interpolieren, während das nächste Layout berechnet wird.

5.1 Datenstrukturen

Im Gegensatz zu den zwei GPU-Implementierungen, die in Kapitel 4 vorgestellt wurden, ermöglichen neuste Funktionen von OpenGL 4.3 die Spezifikation von linear adressierbarem Speicher, dessen Server-seitige Nutzung der eines gewohnten Arrays entspricht. Außerdem kann auf den "Missbrauch" der Rendering-Pipeline für reine Berechnungen verzichtet werden, da Compute-Shader unverfälschten Zugriff auf die Rechenleistung der GPU freigeben. Dies eröffnet neue Herangehensweisen an die Grafikprogrammierung und insbesondere an die Speicherverwaltung. Wie bereits erwähnt, befinden sich die Daten in der Mitte von Abbildung 5.1.

5.1.1 Adjazenzliste

Da es sich bei einem Graphen um eine zweidimensionale Struktur handelt, ist die Speicherrepräsentation nicht ohne Weiteres klar. Anforderungen sind unter anderem eine effiziente Matrix-Multiplikation (ACE) oder die Aufzählung adjazenter Knoten (Fruchterman-Reigold). Da viele Graphen natürlichen Ursprungs dünn besetzt sind, folgt eine Anforderung an die

Effizienz der Speicherausnutzung. Dieses Kriterium streicht die Adjazenzmatrix mit $\mathcal{O}(|\mathcal{V}|^2)$ von der Liste möglicher Kandidaten. Auch eine reine Adjazenzliste kommt nicht in Frage wegen des Zusatzaufwands der vielen Zeiger und dem Knotenzugriff in $\mathcal{O}(|\mathcal{V}|)$.

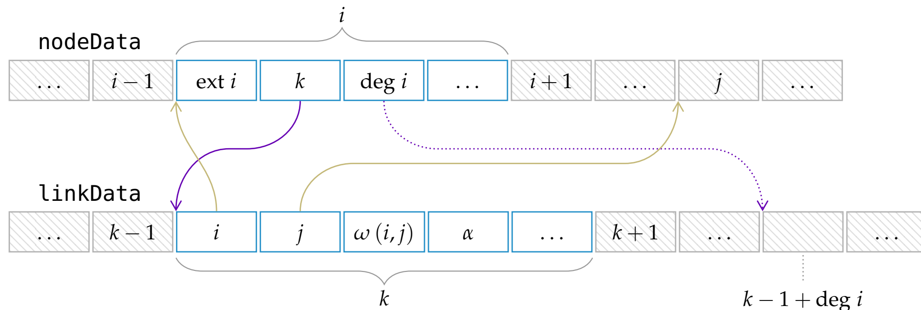


Abbildung 5.2: Speicherlayout und Indizierung

Durch die Normalisierung der Knotenmenge auf $\mathcal{V} = \{0, \ldots, n-1\}$ kann eine Array-basierte Adjazenzliste eingesetzt werden. Da die Daten dabei lückenlos im Speicher liegen, ergibt sich neben einem $\mathcal{O}(1)$ Knotenzugriff ein Speicherverbrauch von $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}|)$. Abbildung 5.2 zeigt einen Ausschnitt aus dieser Datenstruktur, die aus zwei Arrays `nodeData` und `linkData` besteht. Ersteres enthält n Elemente, wobei `nodeData[i]` den Knoten $i \in \mathcal{V}$ repräsentiert. In `linkData` sind alle Kanten nach Quellknoten sortiert abgelegt, sodass mit Hilfe von Kantenindex k und Grad $\deg i$ alle zu i inzidenten Kanten $k, \ldots, k + \deg i$ und damit alle adjazenten Knoten $j \in \mathcal{V}$ mit $\{i, j\} \in \mathcal{E}$ mittels reiner Indexrechnung durchlaufen werden können. Damit eine getrennte Betrachtung der Kanten möglich ist, enthält deren Datenstruktur neben dem obligatorischen Ziel- auch den Quellknoten.

In Abbildung 5.1 erkennt man zwei Pfeile, die den Datenfluss rund um die Adjazenzliste andeuten. Der eingehende Datenfluss stellt den den Upload der vorverarbeiteten Daten der Eingabe dar, der ausgehende führt zu der Server-seitigen Iterationskomponente, da die Vernetzung der Knoten beim Layouting eine wichtige Rolle spielt.

5.1.2 Knoten und Kantenattribute

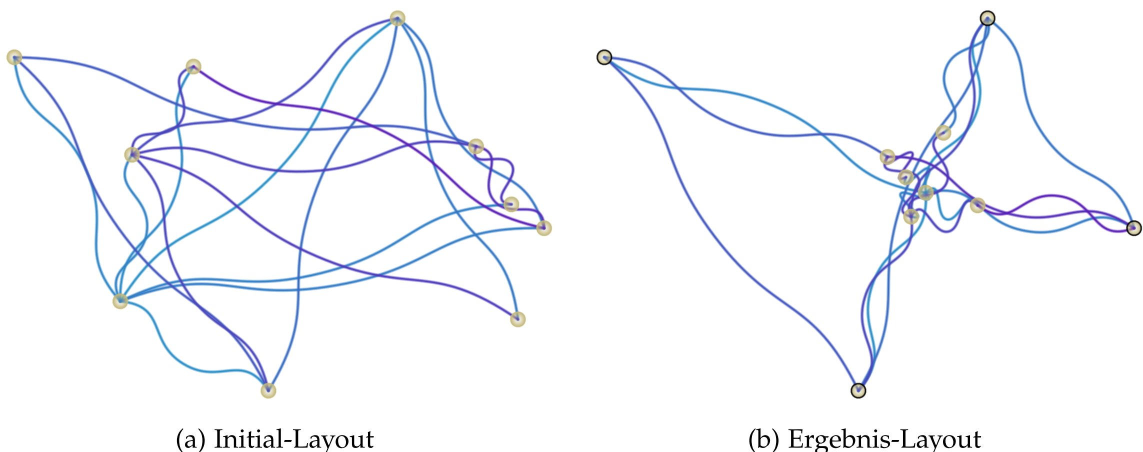
Zu den strukturellen Daten der Adjazenzliste gehört auch das Kantengewicht ω, das die Stärke jeder Verbindung beschreibt, und ein Faktor $\alpha \in [0, 1]$ mit $\alpha := {}^{l}/_{n-1}$, wobei $l = 0, \ldots, n-1$ und $n = |\mathcal{A}^*_{i,j}|$, der eine praktische Unterscheidung ehemaliger Mehrfachkanten ermöglicht (ebenfalls in Abbildung 5.2 zu sehen). Beide sind fest mit den strukturellen Daten verbunden, können aber auch als Attribute aufgefasst werden.

Insbesondere im Zusammenhang der Existenz von Mehrfachkanten ist es wichtig, dass Kanten von der Layout-Berechnung ausgeschlossen werden können. Diese und weitere Optionen müssen von der Datenstruktur unterstützt werden. Darunter sind auf jeden Fall

1. der Ausschluss vom Rendering. Dieses Flag ist bei Knoten oder Kanten gesetzt, die der Benutzer als uninteressant oder störend befunden hat und bei Kanten, die im ursprüng-

lichen Graphen nicht existierten und nur für die Ergänzung zu einem ungerichteten Graph eingefügt wurden.

2. die Ausklammerung für das Layout-Verfahren. Aufgabe dieses Flags ist es, den Einfluss eines Knotens auf andere Knoten zu verhindern oder den Effekt einer Kante auf das Layout auszuschalten. Dadurch kann der Benutzer nach ähnlichen Kriterien wie oben entscheiden, Knoten oder Kanten zu ignorieren, jedoch verändert sich dadurch das resultierende Layout. Falls die Eingabe Mehrfachkanten aufweist, werden diese zwar hinzugefügt, aber mittels dieses Flags vom Layout ausgenommen.
3. die Fixierung der Knotenposition. Dadurch können wichtige Knoten visuell und strukturell hervorgehoben werden. Fixiert man nur eine Komponente und setzt diese zuvor für verschiedene Knotengruppen auf den gleichen Wert, ordnen sich diese jeweils entlang der gleichen Ebene an. Außerdem kann durch das Festhalten eines Knotens gewährleistet werden, dass bei Fruchterman-Reigold das Layout keine gleichbleibende Bewegungsrichtung beibehält und sich damit aus dem Sichtfeld und gegebenenfalls aus dem Darstellungsraum bewegt.
4. die Selektion. Mittels dieses Flags werden Knoten und Kanten identifiziert, die vom Benutzer markiert wurden. Damit werden lokale Operationen auf einer Untermenge der Knoten oder Kanten realisierbar, wie das Setzen von Flags und damit eine generelle Umsetzung der Punkte 1–3.



(a) Initial-Layout

(b) Ergebnis-Layout

Abbildung 5.3: Effekt der Fixierung von Knoten bei Fruchterman-Reigold

Abbildung 5.3 zeigt den Effekt, den eine totale Fixierung von Knoten bei Fruchterman-Reigold hat. Fixierte Knoten sind außerdem markiert, indem ein schwarzer Rand um diese gezeichnet wird.

Außerhalb der Anwendung haben Knoten unter Umständen von $\mathcal{V} = \{0, \ldots, n-1\}$ verschiedene Repräsentationen. Deshalb ist eine externe ID $\text{ext}\, i$ hinterlegt, die zum Beispiel für das Nachschlagen in einem Texture Atlas wichtig ist. Dies gibt dem Programmierer die Freiheit mit eigenen IDs Sachverhalte zu verknüpfen, anstatt von vornherein an das Format $0, \ldots, n-1$ gebunden zu sein. Handelt es sich bei der externen ID um keine Zahl, muss eine

Abbildung erfolgen, die es dem Programmierer nur erlaubt auf Client-Seite mit diesen zu hantieren, da sich allgemeine Zeichenfolgen nicht für die Server-seitige Verwendung eignen.

Allen Objekten der Szene, also Knoten und Kanten, wird zudem ein Bounding Volume zugeordnet, um die Überlappung von Objekten oder den automatischen Zoom berechnen zu können.

Jedes dieser Attribute kann von der Knoten- beziehungsweise Kantendatenstruktur abgetrennt gesondert in einem Array gehalten werden, da stets über denselben Index darauf zugegriffen werden kann. Knoten- und Kantenattribute dienen ebenfalls als Eingabe der Iteration, wie in Abbildung 5.1 dargestellt. Allerdings sind diese noch für einen weiteren Zweck von Nutzen. Das Rendering kann den Zustand der Attribute in der Visualisierung umsetzen. Nachdem die Attribute im Anschluss an das Preprocessing übertragen wurden, können diese vom Benutzer durch Interaktion modifiziert werden.

5.1.3 Layout und Hilfsdaten

Im Prinzip handelt es sich bei der Position der Knoten auch um eines deren Attribute, jedoch wird das Layout als Ergebnis des eingesetzten Verfahrens getrennt betrachtet. Die Hilfsdaten, die vom jeweiligen Layout-Verfahren berechnet und weiterverwendet werden, können von Algorithmus zu Algorithmus unterschiedlich sein. Bei der FDL-Methode bestehen diese zum Beispiel aus einem einzigen Array, das die pro Knoten berechneten Kräfte beinhaltet und gegebenenfalls einem temporären Array für die Bestimmung der aktuellen Energie.

Die im vorherigen Abschnitt erwähnten Bounding Volumes spannen zusammen ein globales Bounding Volume auf. Für deren Berechnung wird ebenso ein temporäres Array benötigt.

Beide Datenkategorien müssen zunächst initialisiert werden. Abbildung 5.1 zeigt die Parallelität der Rollen von Layout und Hilfsdaten gegenüber der Adjazenzliste und den Attributen. Der Unterschied ist die zusätzliche Gegenrichtung des Datenflusses bei der Iteration. Die Existenz der Hilfsdaten beruht auf dieser Abhängigkeit, da sie unter anderem das Layout-Verfahren unterstützen. Das Layout kann neben der offensichtlichen Aufgabe das Ergebnis zwischenzuspeichern ebenfalls in der nächsten Iteration verwendet werden, bevor es überschrieben wird.

5.2 Programmkomponenten

Neben dem Client-seitigen Teil des Frameworks werden verschiedene Server-seitige Programmkomponenten verwendet, die in Abbildung 5.1 auf der unteren Schiene angesiedelt sind, mit einer gegebenenfalls fest verschachtelten Hybrid-Komponente, der Interaktion. Die dort eingesetzten Programme lassen sich in drei Kategorien einteilen:

1. Ein Rendering-Programm ist ein OpenGL-Programm, das die in Abschnitt 2.2 gezeigte Pipeline umsetzt. Dabei besteht die Vertex-Menge aus lediglich einem Vertex, da mittels Instancing $|\mathcal{V}|$ Instanzen der gleichen Pipeline aufgerufen werden, die sich nur durch eine Instanz-ID unterscheiden. Der Sinn dahinter ist die Tatsache, dass für jeden Knoten

und für jede Kante exakt das gleiche durchgeführt wird, bis auf einige Ausnahmen, wie der Indizierung der Datenstrukturen oder einer Farbänderung auf Grund eines gesetztem Selektions-Flags.

2. Ein Reduktions-Programm ist das Programm, das aus einem Compute-Shader entsteht, von dem $c \in \mathbb{N}$ viele Instanzen gestartet werden. Jede Instanz übernimmt eine Untermenge an Elementen, deren Ergebnis jeweils in einen Slot temporären Speichers geschrieben wird. Die letzte Instanz übernimmt dann die Berechnung des Endergebnisses aus diesen Zwischenergebnissen. Es muss das letzte sein, da unter Umständen nicht alle Instanzen parallel ausgeführt werden können und somit erst beim letzten Aufruf gewährleistet ist, dass alle Zwischenergebnisse vorliegen. Im Idealfall trifft c die Anzahl verfügbarer Recheneinheiten, damit nach einem Parallelaufruf das Ergebnis vorliegt. Auf diese Weise wird die GPU maximal ausgelastet. Allerdings ist das ideale c Hardware- und situationsabhängig.

3. Ein Knoten- beziehungsweise Kanten-Programm wird wie ein Rendering-Programm $|\mathcal{V}|$ respektive $|\mathcal{E}|$ Mal aufgerufen, sodass jede Programminstanz für einen Knoten beziehungsweise eine Kante zuständig ist.

Alle Programme vom Typ 1 werden in einem Anzeige-Callback nacheinander ausgeführt und gegebenenfalls übersprungen. Dadurch können alle durch das jeweilige Programm gezeichneten Objekte unsichtbar gemacht werden. Ein Beispiel hierfür wäre die Visualisierung der Bounding Volumes. Desweiteren können durch die in Abschnitt 5.1.2 eingeführten Flags mehrere Programme mit dem Zweck der Visualisierung der gleichen Objektart eingebaut werden. Jedes dieser Programme zeichnet gezielt nur eine Untermenge, sodass zum Beispiel zunächst alle Knoten eines bestimmten Typs als Kreis gezeichnet werden und anschließend alle anderen als Rechteck. Dieser Prozess spielt sich in der Komponente Rendering in Abbildung 5.1 ab.

Die Programmtypen 2 und 3 werden an verschiedenen Stellen aufgerufen, so ist zum Beispiel die Layout-Iteration ein Programm vom Typ 3 und die Berechnung des globalen Bounding Volumes vom Typ 2. Typ 2 hat unter idealen Bedingungen volle Last der Hardware zur Folge, da dies jedoch nicht gewährleistet werden kann, kommt Typ 3 für umfangreichere Berechnungen zum Einsatz.

5.3 Preprocessing

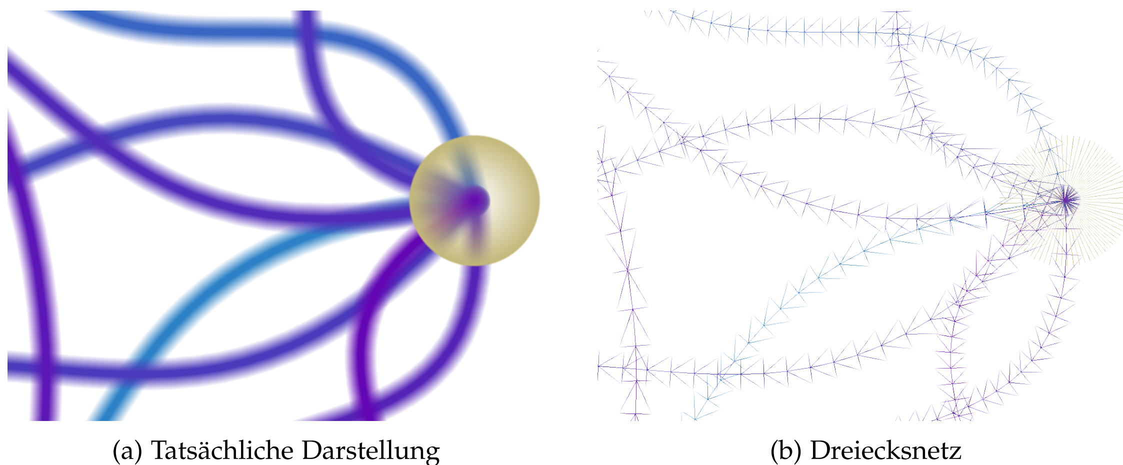
Die Aufgabe des Preprocessing ist es die Eingabedaten, in die Datenstrukturen aus Abschnitt 5.1 zu überführen. Abbildung 5.1 zeigt das Preprocessing als Einstiegspunkt innerhalb der Blackbox und damit als Bindeglied zwischen der Eingabe und dem Herzstück des Frameworks. Dabei müssen beliebige Graphen semantisch auf ungerichtete Graphen ohne Mehrfachkanten reduziert werden. Die Informationen, die dabei verloren gehen, müssen jedoch weitergegeben werden. Die Standarddatenstruktur, die das Preprocessing verarbeitet, stellt eine Liste von Kantentripeln $(a, b, c) \in \mathbb{R}^3$ dar: Quell- und Zielknoten $a, b \in \mathbb{R}$ mit einem Kantengewicht $c \in \mathbb{R}$. Dazu kommen unter Umständen noch Vorbelegungen der Bitfelder von Knoten oder Kanten, deren Aufnahme hier vernachlässigt wird.

Diese Kantenliste wird elementweise durchlaufen, wobei für jedes Tripel die folgenden Schritte durchgeführt werden. Es wird überprüft, ob für a und b bereits interne Knoten-IDs aus $\mathcal{V}$ vergeben wurden. Falls nicht wird die interne Knotenmenge jeweils um ein Element i beziehungsweise j erweitert und deren Adjazenz mit dem zugehörigen Kantengewicht c in einer Adjazenzliste abgelegt. Wird erneut eine Kante mit gleichem Quell- und Zielknoten betrachtet, wird diese hinter die zuletzt registrierte Kante gehängt, sodass eine Liste entsteht.

Ein weiterer Durchlauf über diese Adjazenzliste erzeugt für jede Kante, für die noch keine Rückkante existiert, eine neue Kante, die von der Darstellung ausgeschlossen wird. Außerdem wird in jeder Kantenliste der Faktor α bestimmt. Falls mehrere Kanten vorliegen, werden diese zu einer unsichtbaren, aber Layout-relevanten Kante zusammengefasst. Dabei ist offen, wie das Gewicht einer solchen Kante zustande kommt. Die anderen Kanten der Menge werden natürlich für das Layouting ausgeschlossen und dienen nur der Visualisierung der Mehrfachkanten.

Abschließend steht die Erzeugung der Rohdaten für die GPU. Die Kanten liegen schon im richtigen Format vor, die Knoten müssen noch in deren Datenstruktur überführt werden.

5.4 Visualisierung



(a) Tatsächliche Darstellung (b) Dreiecksnetz

Abbildung 5.4: Tessellierung von Knoten und Kanten

Davon abgesehen, dass das Knotenlayout eine entscheidende Rolle bei der Visualisierung von Graphen spielt, ergibt sich die Aufgabe die abstrakten Knoten und Kanten mit ihren Attributen darzustellen. Aus der Beschreibung eines Rendering-Programms in Abschnitt 5.2 lässt sich bereits entnehmen, dass nur ein einzelner Vertex Repräsentant jedes abstrakten Objekts ist. Um aus nur einem Vertex eine komplexere Geometrie zu erzeugen, kann mittels Tessellierung eine Vervielfachung eines Vertex zu vier Vertices vorgenommen werden, die ab diesem Zeitpunkt ein Quad beschreiben. Dieses Primitiv existiert nur in diesem Zusammenhang und besteht aus zwei Dreiecken, die zusammengesetzt ein von vier Vertices aufgespanntes Rechteck darstellen. Vorteil dieses Primitives ist die Möglichkeit, Positionen

innerhalb der Geometrie mittels bilinearer Interpolation ausdrücken zu können. Nach einer Unterteilung dieses Primitives in eine Menge von Dreiecken (TCS), können deren Vertices an beliebige Positionen verschoben werden, sodass ein völlig neues geometrisches Objekt entsteht (TES). Abbildung 5.4 zeigt eine mögliche Umsetzung dieses Konzepts, das neben den anderen Ideen im Bereich Rendering von Abbildung 5.1 zum Einsatz kommen könnte.

5.4.1 Knoten

Es ist offensichtlich, dass jede Visualisierung darauf basiert, dass die zugrunde liegenden Daten in Bezug auf deren Kontext in irgendeiner Weise erkennbar werden. Im Kontext des Graph-Drawing steht die Visualisierung mittels eines Knotenlayouts an vorderster Stelle. Das heißt die Position der Knoten muss erkennbar sein. Dies lässt sich auf sehr einfache Art realisieren. Allerdings geht die Semantik eines Knotens insbesondere bei natürlichen Graphen weit über die dessen Existenz hinaus. So besteht die Möglichkeit Teile dieser Semantik auf die Visualisierung zu übertragen. Unter diesem Aspekt kommt mehr als nur eine repräsentative Form der Darstellung von Knoten in Frage. Folgende Überlegungen befassen sich mit der Umsetzung verschiedener Knotenrepräsentationen für $\mathcal{R} = \mathbb{R}^2$.

Unsichtbare Knoten

Man verzichtet auf die Visualisierung der Knoten. Dies ist nur sinnvoll, falls jeder Knoten mit mindestens einer Kante verbunden ist. Eine Beispielanwendung wäre einer der drei Graphen aus der Einleitung. In einem Straßennetz entstehen die Knoten erst durch Kreuzungen der Kanten.

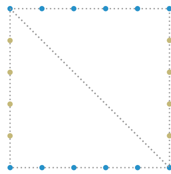
Punkte

Um die Knoten hervorzuheben und gegebenenfalls durch Farben zu unterscheiden, läge es nahe Punkte für die Visualisierung in Betracht zu ziehen. Der Begriff des Punktes zielt hierbei auf das Primitiv von OpenGL. Mit dieser Entscheidung wird der oben beschriebene Erweiterungsprozess nicht benötigt. Der einzelne Vertex wird schlicht und einfach als Punkt an die Rendering-Pipeline übergeben.

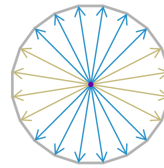
Symbole

Spätestens, wenn die Punktgrößen unterschiedlich oder größer als die von OpenGL vorgegebene Grenze sein sollen, benötigt man jedoch den Erweiterungsansatz. Einfache Beispiele sind ein Rechteck oder ein Kreis. Ersteres lässt sich direkt umsetzen, indem keine Unterteilung im TCS vorgenommen wird.

Eine Kreisform erhält man durch eine gleichmäßige Unterteilung der Seiten des Quads in $s \in \mathbb{N}$ Teile mit anschließender Positionierung der äußeren Vertices mit Radius $r \in \mathbb{R}^{>0}$. Mit



(a) Tesselierung eines Quads



(b) Positionierung durch Radius r

Abbildung 5.5: Tesselierung eines Quads zu einem kreisförmigen Polygon

Hilfe der modifizierten Normalisierung

$$n_\varepsilon(a) := \begin{cases} 0 & \text{für } \|a\| < \varepsilon, \\ \frac{a}{\|a\|} & \text{sonst} \end{cases}$$

können alle Vertices anhand ihrer bilinearen Koordinaten $t \in [0,1]^2$ um die Knotenposition $\mathcal{L}(i)$ angeordnet werden und der zentrale Vertex bleibt an seiner Ausgangsposition: $\mathcal{L}(i) + r \cdot n_\varepsilon\left(t_x - 1/2, t_y - 1/2\right)$. Abbildung 5.5 zeigt dieses Vorgehen schematisch am Beispiel $s = 5$.

Alternativ genügt auch hier das einfache Quad-Primitiv. Die Form des Knotens kann allein durch den Fragment-Shader bestimmt werden, indem dieser alle Fragmente außerhalb von r ignoriert.

Bilder und Texte

Diese Art der Visualisierung baut auf der vorherigen auf, indem einfache Symbole wie Rechtecke mit Texturen versehen werden, die ein Bild und/oder einen Text zeigen. Man kann sogar so weit gehen, in den Bereich des Rechtecks eine Szene mittels Raytracing zu rendern. Dieses Verfahren ist deutlich aufwändiger als Rasterisierung, führt jedoch zu besseren Ergebnissen, da für jeden Pixel ein Strahl in die zu zeichnende Szene geschickt wird, der nach physikalischen Gesetzen einen Pfad zu einer Lichtquelle sucht. Als Beispiel könnte man für die Analysedaten von Amazon eine Menge von Kategorien für Produkte bestimmen und jeden Knoten, dessen zugehöriges Produkt in eine bestimmt Kategorie fällt, mit einem Bild eines repräsentativen Produkts visualisieren.

5.4.2 Kanten

Bei Knoten ist deren Position im Layout ausschlaggebend für deren Visualisierung. Kanten zeichnen sich durch deren Inzidenz aus, das heißt es muss erkennbar sein, dass eine Verbindung zwischen zwei Knoten besteht, falls eine Kante diese verbindet. Die folgenden Überlegungen beschäftigen sich mit dem Verlauf von Kanten in $\mathcal{R} = \mathbb{R}^2$ und Möglichkeiten, die Richtung, das Gewicht oder den Grad einer Kante darzustellen.

Unsichtbare Kanten

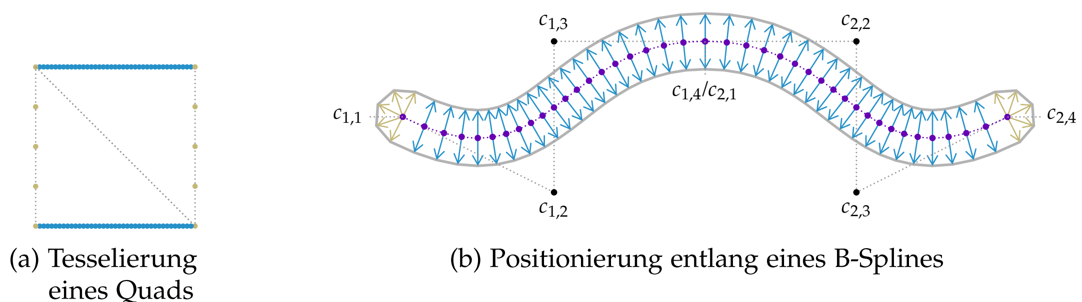
Wenn der tatsächliche Kantenverlauf weniger interessant und lediglich wichtig ist, wo die Knoten sich letztendlich versammeln, kann auch hier auf eine Visualisierung verzichtet werden. Die Verbindungsinformation liegt dann implizit in Form des Layouts vor. Allerdings ist diese Variante hier deutlich kritischer zu betrachten, da zwei Knoten, die sehr weit voneinander entfernt liegen, dennoch mit einer Kante verbunden sein können.

Linien

Wie bei den Knoten, gibt es auch hier ein besonders einfaches Primitiv, dass das Grundkriterium der Visualisierung erfüllt. Das OpenGL-Primitiv einer Linie erfordert jedoch zusätzliche Modifikationen am zuständigen Rendering-Programm und dessen Aufruf. Für eine Linie werden immer zwei Vertices benötigt, also muss das Vertex-Array um einen Vertex erweitert werden, der sich vom vorhandenen unterscheidet, damit die Positionen der Start- und Endpunkte der Linie entsprechend der Knotenpositionen von Quell- und Zielknoten gesetzt werden können.

Kurven

Ähnlich dem Punkt-Primitiv hat auch das Linien-Primitiv Grenzen in OpenGL, was deren Ausdehnung angeht. Um diese zu umgehen, reicht erneut das Quad-Primitiv aus. Wird die Breite so gewählt, dass die Länge der Verbindung überbrückt wird, kann die Höhe entsprechend der gewünschten Linienstärke angepasst werden. Unterteilt man auch hier die Seiten in geeigneter Weise, kann neben einer beliebigen Formänderung eine spezielle Art der Verformung durchgeführt werden, nämlich die Krümmung der Kante.



(a) Tesselierung eines Quads

(b) Positionierung entlang eines B-Splines

Abbildung 5.6: Tesselierung eines Quads zu einer gebogenen Kante

Das folgende Verfahren stellt eine Kante mit einer Dicke $r \in \mathbb{R}$ dar, die einer Krümmung folgt, die durch drei Kontrollpunkte definiert wird. Außerdem werden die Endpunkte als Halbkreise modelliert, um kreisförmige Überlappungsbereiche zweier eingehender Kanten zu erhalten. Dafür werden jeweils parallele Seiten gleichermaßen im TCS unterteilt, wobei in der Horizontalen deutlich mehr neue Vertices benötigt werden. h sei die Zahl der horizontalen Abschnitte, v die der vertikalen. Dazu kommt, dass das Quad auch innerhalb einmal unterteilt

werden muss, damit der Übergang zu einem Halbkreis möglich wird. Die Krümmung der Linie wird durch einen kubischen Bézier-Spline (B-Spline)

$$b(x) := \begin{cases} b_k(nx - k + 1) & \text{für } k - 1 < nx \leq k, \\ 0 & \text{sonst} \end{cases}$$

aus n Teilpolynomen b_k definiert, wobei

$$b_k(x) := \sum_{d=0}^{3} c_{k,d+1} \mathcal{B}_d(x)$$

für $k = 1, \ldots, n$ mit der kubischen Bernstein-Basis

$$\mathcal{B}_d(x) = \binom{3}{d} x^d (1 - x)^{3-d}$$

für $d = 0, \ldots, 3$. Die interpolierenden Zwischenpunkte $c_{k,4} = c_{k,1}$ mit $k = 1, \ldots, n - 1$ sind implizit über die übrigen Koeffizienten gegeben. Damit b bei der Position von Quellknoten $i \in \mathcal{V}$ startet und in der Position von Zielknoten $j \in \mathcal{V}$ endet, müssen $c_{1,1} = \mathcal{L}(i)$ und $c_{2,4} = \mathcal{L}(j)$ sein. Die restlichen Koeffizienten definieren die Krümmung von b. Eine solche Kurve ist in Abbildung 5.6b eingezeichnet.

Da die vertikale Unterteilung nicht entlang eines Rechtecks sondern nur entlang einer Linie verläuft, kann das Verfahren, das bei der Erzeugung einer Kreisform angewendet wird, hier erst nach einer Transformation zum Einsatz kommen. Der erste Schritt ist eine Modifikation der horizontalen Koordinate $t'_x = (t_x - 1/h) \cdot (h/h')$ mit $h' = h - 2$. Dadurch werden die ersten beziehungsweise letzten v Vertices herausgenommen und die restlichen Koordinaten entsprechend angepasst. Nach dieser Transformation wird der Richtungsvektor

$$d = \begin{cases} b\left(t'_x + \frac{1}{h'}\right) - b(t'_x) & \text{für } 0 \leq t'_x \leq 1 - \frac{1}{h'}, \\ b(0) - t'_x \left(b\left(\frac{1}{h'}\right) - b(0)\right) & \text{für } t'_x < 0, \\ b(1) + t'_x \left(b(1) - b\left(1 - \frac{1}{h'}\right)\right) & \text{sonst} \end{cases}$$

und die zugehörige Normale $n = (d_y, -d_x)$ bestimmt, die jeweils die Ausrichtung des betrachteten Abschnitts beschreiben. Die beiden Randfälle $t'_x < 0$ und $t'_x > 1 - \frac{1}{h'}$ werden mittels Erweiterung um die erste beziehungsweise letzte Richtung aufgelöst.

Gilt nun $t'_x \notin [0, 1]$, handelt es sich um eines der Linienenden. Je nachdem um welchen Endpunkt es sich handelt, wird das Kreiszentrum notiert mit

$$s = \begin{cases} \mathcal{L}(i) & \text{für } t'_x < 0, \\ \mathcal{L}(j) & \text{sonst.} \end{cases}$$

Anschließend erfolgt eine Transformation der y-Koordinate von $[0, 1]$ auf $[-2, 2]$. Die drei folgenden Fälle entsprechen jeweils einer der drei benötigten Seiten eines halboffenen

Rechtecks. Falls $t'_y > 1$ ergibt sich die Position

$$t'' = \begin{cases} \left(2 - t'_y\right) \cdot d + n & \text{für } t'_y < -1, \\ \left(2 + t'_y\right) \cdot d - n & \text{sonst,} \end{cases}$$

sodass für

$$\left|t'_y\right| \to 2 : \; t'' \to \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

Die Vertikale $t'_y \in [-1, 1]$ muss nur entlang der Endrichtung der Kurve gedreht werden mit $t'' = d + t'_y n$, sodass nun mittels $s + r/2 \cdot n_\varepsilon\left(t''\right)$ die finale Position gefunden ist. An dieser Stelle sei darauf hinzuweisen, dass durch die Spiegelbildlichkeit der Linienenden eine Vertauschung der Koordinaten vorliegt, da jedoch zwei Polygonkanten aufeinander fallen, dies keinen Einfluss auf das Rendering hat. Innerhalb des Intervalls $t'_x \in [0, 1]$ wird die Position entlang der Krümmung bestimmt durch $b\left(t'_x\right) + r/2 \cdot \left(t_y - 1/2\right) \cdot n$. Abbildung 5.6b zeigt das Ergebnis für $h = 20$ und $v = 4$ schematisch. Das tatsächliche Ergebnis des Verfahrens kann in Abbildung 5.4 betrachtet werden.

Die Krümmung einer Linie kann für verschiedene Zwecke eingesetzt werden. Betrachtet man das Beispiel des Straßennetzes, könnte die Krümmung der Kante die reale Krümmung der Straße approximieren. Sind zwei Knoten mit mehreren Kanten verbunden (Hin- und Rückkante oder Mehrfachkante), kann mittels des Faktors α, der in Abschnitt 5.1.2 eingeführt wurde, jede Kante derart gekrümmt werden, dass sie die anderen nicht überdeckt. Kontrollpunkte gleichen y-Werts einer jeden Kanten werden mittels α entlang der Richtung der Normalen verteilt. Das dritte Anwendungsgebiet kurvenförmiger Kanten ist das sogenannte Edge-Bundling. Ziel dieses Postprocessing ist es, Kanten, die weite Strecken entlang der gleichen Richtung verlaufen, zu Bündeln zusammenzufassen, um dadurch einen besseren Überblick über die Vernetzung zu gewinnen. Die Einbettung dieses Schritts in das konzeptionelle Framework wird hier nicht weiter betrachtet. Interessant ist auf jeden Fall die Übertragbarkeit von Konzepten. So verwenden Holten und Van Wijk [HV09] einen FDL-Ansatz, indem Kanten durch das Setzen neuer Knoten in Abschnitte unterteilt werden, die mittels Federn verbunden werden und sich für ein Kantenpaar paarweise anziehen. Andere Ansätze konstruieren ein Gerüst, an das sich die Kanten anschließend schrittweise anschmiegen [Ers+11], oder stellen ein Polygonnetz auf, dessen Bereiche bestimmen, welche Kanten gebündelt werden [Cui+08].

Richtung, Gewicht und Sonderfälle

Handelt es sich bei $\mathcal{G}$ um einen gerichteten Graphen und spielt diese Eigenschaft bei der Betrachtung von $\mathcal{G}$ eine Rolle, muss die Richtung einer Kante in der Visualisierung erkennbar werden. Auch hier folgen einige Überlegungen, wie dies umgesetzt werden könnte.

1. Die Farbe der Kanten identifiziert Quelle und Ziel. Dabei kann auch ein Übergang des Alpha-Werts zum Einsatz kommen. Mögliche Verläufe sind zum Beispiel eine Unterteilung in zwei Farben oder auch drei Farben. Dann markiert ein kurzer Teil jeweils an Start und Ende der Kante die Richtung.

2. Die Form der Kante unterscheidet Quelle und Ziel. Dies kann über einen Formverlauf geschehen. Dazu gehören zum Beispiel die Modifikation der Dicke, des Kantenverlaufs, wie der Darstellung einer Welle mit unterschiedlicher Wellenlängen, oder der Dekoration der Kante mit Geometrie progressiver Größe. Die offensichtlichste Lösung ist die Darstellung durch Pfeile, deren Start und Ende jeweils mit einem Symbol voneinander unterschieden wird. Bei hoher Kantenzahl wird jedoch durch fehlende Übersicht die Richtung nur noch schwer erkennbar.
3. Falls die Laufzeit, die mit der Kantenzahl des Graphen verknüpft wäre, eine Animation jeder Kante zulässt, kann auch durch ein sich in Richtung der Kante bewegendes Objekt eine Visualisierung dieser erfolgen. Beispiele hierfür wären ein Lauflicht, ein Farbverlauf oder ein geometrisches Objekt.

Auch die Gewichtung der Kante kann ein wichtiger Indikator sein und sollte sich in diesem Fall in der Visualisierung wiederfinden. Einige Kandidaten könnten von den folgenden Ideen abstammen.

1. Das Gewicht bestimmt die Farbe der Kante. Wie bei der Visualisierung der Richtung kann Alpha-Blending benutzt werden, um zum Beispiel wichtigere Kanten opak zu zeichnen und mit abnehmendem Gewicht transparenter.
2. Man zeichnet Kanten mit höherem Gewicht dicker, als solche mit geringem Gewicht.
3. Man lässt das Gewicht in den Layout-Algorithmus einfließen. Bei Fruchterman-Reigold multipliziert man zum Beispiel die anziehende Kraft F_a mit dem Gewicht der Kante, bei ACE wird das Gewicht automatisch miteinbezogen, sodass man im Gegenteil die Gewichte bei der Aufstellung von L vernachlässigen muss, sollte deren Berücksichtigung nicht erwünscht sein.

Wie bereits angesprochen, benötigen Mehrfachkanten eine Sonderbetrachtung. Verzichtet man auf die Auflösung der Überlappung von Kanten gleichen Indexes durch geometrische Modifikationen, könnten der Grad einer Mehrfachkante mit den selben Mitteln wie das Kantengewicht visualisiert werden.

Ein weiterer Spezialfall sind Schlingen, das sind Kanten, die denselben Knoten als Quell- und Zielknoten aufweisen. Die Definition aus Kapitel 2 schließt diese aus, falls deren Existenz in der Visualisierung jedoch wichtig ist, bieten sich zum Beispiel zwei folgende Möglichkeiten an. Man modelliert eine Schlinge als Kante und fügt sie mittels kreisförmiger Krümmung hinzu oder man erweitert die Visualisierung der Knoten, die Schlingen aufweisen. Denn diese tragen nicht zum Knotenlayout bei.

5.5 Einbettung von Layout-Verfahren

Auch wenn idealerweise jedes Layout-Verfahren in dieses Framework eingebettet werden kann, gibt es einige generelle Vorüberlegungen, die sich ohne explizite Angabe eines Algorithmus formulieren lassen. Dazu gehört die Initialisierung und die Auflösung von Positionssingularitäten, die sich jeweils in den Bereichen Initialisierung respektive Iteration von

Abbildung 5.1 wiederfinden. Ebenfalls interessant für die die Visualisierung ist die Berücksichtigung von Bounding Volumes bei den Iterationen des verwendeten Layout-Verfahrens.

5.5.1 Initialisierung

Die Initialisierung der Hilfsdaten hängt allein vom verwendeten Layout-Verfahren ab und muss deshalb für jeden umgesetzten Algorithmus mitgeliefert werden. Für iterative Verfahren muss ein Initial-Layout berechnet werden und das sollte mit möglichst wenig Aufwand passieren, da dieser von den anschließenden Iterationen aufgewendet wird. Einige mögliche Kandidaten sind:

1. Alle Knoten befinden sich zu Beginn auf der gleichen Position. Dies ist jedoch nur möglich, falls alle darauffolgenden Berechnungen robust genug sind, mit Singularitäten umzugehen, das heißt mit der Situation zweier Knoten, die sich an der gleichen Stelle befinden.

2. Die initialen Positionen werden zufällig gewählt. Auf Client-Seite ist dies durch einen globalen Seed und resultierenden Pseudozufallszahlen einfach umsetzbar, in der parallelen Verarbeitung auf dem Server jedoch nicht ohne Weiteres möglich, da der Seed für gleichzeitig ausführende Programme zwangsläufig der gleiche ist.

   Dort kann eine Hash-Funktion zu ähnlichen Ergebnissen führen, wie zum Beispiel von Lumina [Lum08] für $\mathcal{R} = \mathbb{R}^2$:

   $$\mathcal{H}(a) := 2 \cdot (f(a) - \lfloor f(a) \rfloor) - 1$$

   mit $f(a) := \sin(\gamma \cdot \langle a, b \rangle)$, wobei $b = (12.9898, 78.233)^T$ und $\gamma = 43758.5453$. Da es sich um eine Hash-Funktion handelt, wird jeder Seed s stets auf den gleichen Wert $\mathcal{H}(a) \in [-1, 1)$ abgebildet. Dies kann durch Verwendung der Eindeutigkeit der Knoten-IDs und eines zufälligen globalen Seeds behoben werden, der auf Client-Seite bestimmt wird. Für die Initialisierung einer Position höherer Dimension benötigt man entsprechend viele unterschiedliche Seeds, denn ansonsten befinden sich alle Positionen auf einer Linie. Für $\mathcal{R} = \mathbb{R}^2$ wählt man also zwei Seeds $s_1, s_2 \in \mathbb{R}^2$ mit zufälligen Koordinaten und $r \in \mathbb{R}$, sodass der Bereich in dem sich die Positionen befinden einen Kreis mit Radius r aufspannt.

   $$\begin{aligned} \mathcal{L}_x^0(i) &= r \cdot \mathcal{H}(s_1 s_{1,x}(i+1)), \\ \mathcal{L}_y^0(i) &= \sqrt{r^2 - (\mathcal{L}_x^0(i))^2} \cdot \mathcal{H}(s_2 s_{1,y}(i+1)). \end{aligned}$$

3. Die Verwendung einer geordneten Hash-Funktion, wie $\mathcal{L}^0 : \mathcal{V} \to [-k/2, k/2]^2$ in $\mathcal{R} = \mathbb{R}^2$ mit $k := \sqrt{|V|}$ und

   $$\mathcal{L}^0(i) := \begin{pmatrix} \frac{k}{2} - (i \bmod k) \\ \frac{k}{2} - \lfloor \frac{i}{k} \rfloor \end{pmatrix}.$$

5.5.2 Auflösung von Positionssingularitäten

Im vorherigen Abschnitt fiel bereits das Wort "Singularität" im Zusammenhang mit dem trivialen Initial-Layout. Befinden sich zwei Knoten an der selben Stelle, kann es sein, dass nicht ohne Weiteres klar ist, wo sich welcher Knoten als nächstes hinbewegen soll. Dies ist insbesondere wichtig für Singularitäten, wie der Überlappung zweier Bounding Volumes, die sehr häufig auftreten. Ein Beispiel wäre eine Auflösung bei Fruchterman-Reigold in $\mathcal{R} = \mathbb{R}^2$ mittels

$$\Delta l^{\tau}_{\varepsilon}(i,j) = \begin{cases} \Delta l^{\tau}(i,j) & \text{für } \|\Delta l^{\tau}(i,j)\| < \varepsilon, \\ (\varepsilon, \varepsilon) & \text{für } i < j, \\ (-\varepsilon, \varepsilon) & \text{sonst.} \end{cases}$$

Die Auflösung muss deterministisch geschehen, damit in jedem Fall gewährleistet ist, dass eine Auflösung auch tatsächlich stattfindet. Außerdem dürfen die beiden Knoten nicht in die gleiche Richtung bewegt werden, da sonst eine konstante Bewegung beider Knoten in diese Richtung beginnt, die ohne genügend große Einflüsse von außen nicht abgebrochen wird. Antiparallele Auflösung führt ebenfalls zu einem Problem, da durch ständige Auflösung die Knoten nur auf einer Linie verteilt werden.

Fruchterman und Reingold [FR91] schlagen in ihrer Arbeit die Definition eines Rahmens vor, den Knoten nicht verlassen können. Dies rührt daher, dass unverbundene Subgraphen durch deren Abstoßung zum Rest des Graphen ungehindert aus dem darstellbaren Bereich verschwinden würden. Die Laufzeitoptimierung von Fruchterman-Reigold durch eine Unterteilung des Darstellungsraums in ein Raster, sodass weit entfernte Knoten sich nicht beeinflussen, bietet automatisch eine adaptierbare Lösung, bei der die Knoten nicht nur entlang des Rahmens angeordnet werden. Andere Layout-Verfahren beschränken sich ebenfalls auf zusammenhängende Graphen [LY12; KCH03; KK89] und schlagen zum Beispiel vor, jeden zusammenhängenden Teil des Graphen einzeln zu betrachten [KCH03].

5.5.3 Berücksichtigung von Bounding Volumes

Die in Kapitel 3 und 4 vorgestellten Layout-Verfahren betrachten Knoten als unendlich kleine Objekte und somit wird die mögliche Ausdehnung eines Knotens nicht miteinbezogen. Dies kann schon ab einer geringen Ausdehnung zu Überlappungen führen. Generell besteht für jeden Layout-Algorithmus, der keinen zwei Knoten dieselbe Position zuweist, die Möglichkeit die berechneten Positionen global zu skalieren, sodass ein noch so kleiner Abstand zweier Knoten groß genug wird, um sie überlappungsfrei darzustellen (vergleiche Harel und Koren [HK02]). Dieses Vorgehen liefert jedoch nur bei ähnlich großen, kugelförmigen Bounding Volumes sinnvolle Ergebnisse. So würde eine einzelnes Bounding Volume mit enormer Ausdehnung dazu führen, das zwischen kleinen Bounding Volumes unnötig viel Platz entstehen würde. Zu diesem Thema gibt es bereits einige Ansätze, darunter beschäftigen sich Harel und Koren [HK02] mit ellipsenförmigen und rechteckigen Knoten unterschiedlicher Größen. Sie präsentieren jeweils eine entsprechende Lösung für Fruchterman-Reigold und

Kamada-Kawai. Zusätzlich befassen sie sich mit der Problematik der Überlappung von Kanten und Knoten. Frishman und Tal [FT08] lösen das Problem sich überschneidender Bounding Boxes durch eine Sortierung der Zeichenreihenfolge, sodass unwichtige Knoten zuerst und mit weniger Opazität als wichtigere Knoten gezeichnet werden. Dadurch kommt es immer noch zu überlappenden Knoten, jedoch sind wichtige Knoten stets verdeckungsfrei sichtbar. Ein Gephi-Plugin namens Noverlap beschäftigt sich ebenfalls mit der Problematik überlappender Knoten, jedoch nur im Zusammenhang kreisförmiger Knotendarstellungen [Gep].

Bei Fruchterman-Reigold und ähnlich aufgebauten FDL-Methoden gibt es mindestens drei Ansatzpunkte, um Bounding Volumes von Knoten zu berücksichtigen:

1. Man betrachtet das verwendete Verfahren als Blackbox und berechnet außerhalb einen neuen virtuellen Verbindungsvektor des betrachteten Knotenpaares, der als modifizierte Eingabe überreicht wird. Da dieser Vektor bei der Knotenabstoßung und der Federanziehung verwendet wird, werden auch beide Teilkräfte beeinflusst. Dies ist einfach umzusetzen und äußerst flexibel, jedoch weniger gut durchschaubar, da außerhalb des Modells gearbeitet wird.
2. Man bezieht das Bounding Volume bei der Abstoßung korrekt ein. Die Federanziehung wird beibehalten und nur die Abstoßung modifiziert. Je nach Bounding Volume kann dieser Ansatz komplex und damit ineffizient sein, insbesondere, wenn es durch die Krafteinwirkungen zu Rotationen kommt. Da man sich aber innerhalb des Kraftmodells befindet, ist dieser Ansatz vorhersehbarer und leichter zu modellieren.
3. Man verwendet eine Kugel als neues Bounding Volume, die das vorhandene Bounding Volume umschließt. Für diesen Ansatz liefert die direkte Berechnung einer Kugel aus den Objektabmessungen bessere Ergebnisse, da der Zwischenschritt über das Bounding Volume eventuell zu einem größeren Radius führt. Dann wird weiter verfahren wie in Ansatz 1.

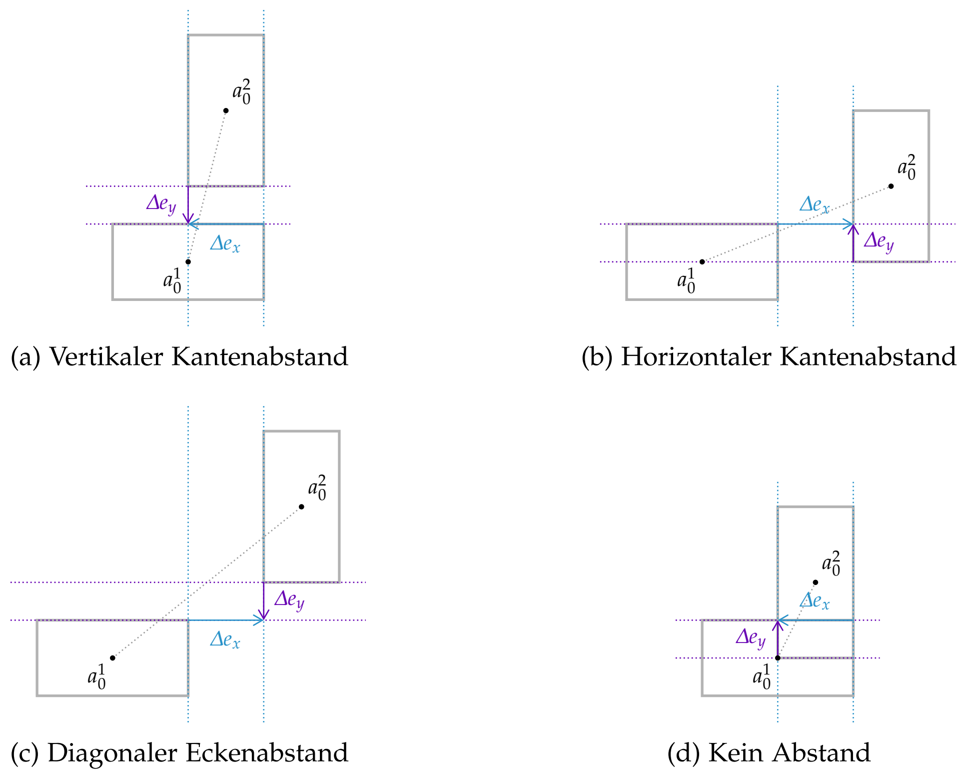
Das folgende Verfahren ist eine Ausprägung von Ansatz 1 für AABBs. Ein Szenario besteht jeweils aus zwei beliebigen Objekten mit Aufhängepunkt $\widetilde{a}_i$ und AABB $\widetilde{B}_i$. Zu Beginn werden die Bounding Boxes mittels

$$\begin{aligned} B_1 = (p_1, q_1) &:= \widetilde{B}_1, \\ B_2 = (p_1, q_1) &:= (\widetilde{a}_2 - \widetilde{a}_1) \oplus \widetilde{B}_2 \end{aligned}$$

in das lokale Koordinatensystem von $\widetilde{B}_1$ transformiert. Die Zentren $a_0^i := 1/2\,(bl_i + tr_i)$ spielen dabei eine Rolle für den normierten Verbindungsvektor $\Delta a_0 := n_\varepsilon\left(a_0^2 - a_0^1\right)$, der in die Berechnungen eingehen wird. Es folgt die Bestimmung des gerichteten Abstandsvektors Δe zwischen parallelen Kantenpaaren, die den geringsten Abstand zu den Zentren aufweisen:

$$\Delta e_d := \begin{cases} p_{2,d} - q_{1,d} & \text{für } a_{0,d}^1 < a_{0,d}^2, \\ p_{1,d} - q_{2,d} & \text{sonst} \end{cases}$$

für $d = 0, \ldots, \dim \mathcal{R} - 1$. Falls $\Delta e_d < 0$, zeigt der Abstand ins Innere der jeweiligen Bounding Box, andernfalls nach Außen. Diese Unterscheidung ist wichtig, da im zweiten Schritt



(a) Vertikaler Kantenabstand

(b) Horizontaler Kantenabstand

(c) Diagonaler Eckenabstand

(d) Kein Abstand

Abbildung 5.7: Fallbeispiele für die modifizierte Verbindungsvektorberechnung

negative Werte auf 0 gesetzt werden. In Abbildung 5.7 sind repräsentative Szenarien aller vier Fälle für $\mathcal{R} = \mathbb{R}^2$ dargestellt. Die Kantenpaare sind entsprechend für $d \in \{x, y\}$ als Hilfslinien eingezeichnet.

1. $\Delta e_x < 0 \wedge \Delta e_y \geq 0$ (vergleiche Abbildung 5.7a): Die Bounding Boxes sind vertikal angeordnet und somit wird nur die vertikale Komponente verwendet, also $\Delta e_x = 0$.
2. $\Delta e_x \geq 0 \wedge \Delta e_y < 0$ (vergleiche Abbildung 5.7b): Die Bounding Boxes sind horizontal angeordnet und somit wird nur die horizontale Komponente verwendet, also $\Delta e_y = 0$.
3. $\Delta e_x, \Delta e_y \geq 0$ (vergleiche Abbildung 5.7c): Die Bounding Boxes sind diagonal angeordnet und somit werden beide Komponenten verwendet.
4. $\Delta e_x, \Delta e_y < 0$ (vergleiche Abbildung 5.7d): Die Bounding Boxes überlappen sich und der Abstand ist damit theoretisch gleich 0, jedoch wird hier eine sehr kleine Konstante ε eingesetzt, damit eine Kraftrichtung vorliegt. Dazu wird ein Hilfsvektor

$$\Delta\varepsilon = \left(\left|\Delta e_x - \varepsilon\right|, \left|\Delta e_y - \varepsilon\right|\right)$$

bestimmt, mit dem Δe wie folgt berechnet wird:

$$\Delta e = \varepsilon \cdot \begin{cases} \left(\left|\Delta a_{0,x}\right|, \left|\Delta a_{0,y}\right|\right)^T & \text{für } \left|\Delta\varepsilon_x - \Delta\varepsilon_y\right| < \varepsilon, \\ (1,0)^T & \text{für } \Delta\varepsilon_y < \Delta\varepsilon_x, \\ (0,1)^T & \text{sonst.} \end{cases}$$

Die drei Fälle orientieren sich an den drei obigen Fällen, sodass auch innerhalb ein ähnliches Abstoßungsverhalten auftritt.

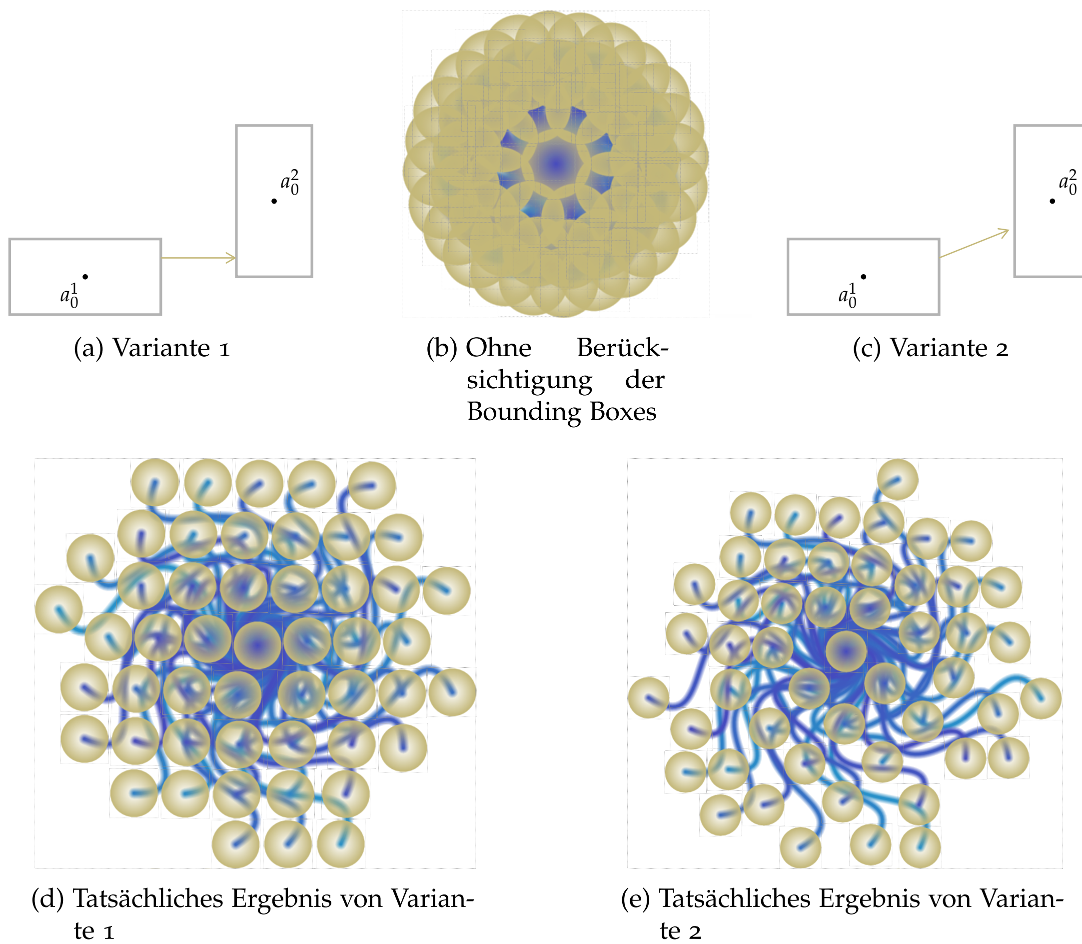
Da von nun an $\Delta e_d \geq 0$ ist ein Teil der Richtungsinformation nach diesen zwei Schritten verloren gegangen. Im letzten Schritt kann diese auf zwei Arten zurückgewonnen werden. Man multipliziert mit dem Vorzeichenvektor, also

$$\Delta l := \begin{pmatrix} \operatorname{sgn} \Delta a_{0,x} \\ \operatorname{sgn} \Delta a_{0,y} \end{pmatrix} \cdot \Delta e^T.$$

Oder man benutzt lediglich die Länge von Δe, also $\Delta l := \|\Delta e\| \cdot \Delta a_0$. Siehe Abbildung 5.8a, respektive 5.8c für einen resultierenden Verbindungsvektor Δl und die Auswirkungen auf die Darstellung eines Ringgraphen. Man sieht, dass Variante 1 deutlich bessere Ergebnisse liefert. Bei Variante 2 gibt es sogar überlappende Bounding Boxes und viele Knoten ordnen sich diagonal an, wodurch mehr Platz verbraucht wird.

Die Anpassung des Verbindungsvektors ist bei der Bestimmung eines Knotenlayouts hinderlich. Es braucht mehr Bewegung, damit ein Knoten sich an einem anderen vorbei bewegen kann. Außerdem kann ein Knoten zwischen anderen eingeschlossen werden. Die Lösung ist der Anschluss des Verfahrens an das normale Knotenlayout, sodass bei niedriger Temperatur überlappende Bounding Boxes auseinander gehen können.

Die visualisierten Kanten können ebenfalls eine Ausdehnung aufweisen. Außerdem besteht auch die von Harel und Koren [HK02] adressierte Kanten-Knoten-Überschneidung. Mögliche Verfahren für die Lösung von Konflikten bezüglich der Ausdehnung von Kanten werden hier jedoch nicht weiter betrachtet.



(a) Variante 1

(b) Ohne Berücksichtigung der Bounding Boxes

(c) Variante 2

(d) Tatsächliches Ergebnis von Variante 1

(e) Tatsächliches Ergebnis von Variante 2

Abbildung 5.8: Varianten für den resultierenden Verbindungsvektor

5.6 Interaktion

Der Benutzer hat nicht nur die Möglichkeit den Iterationsprozess des Layout-Verfahrens zeitlich zu kontrollieren. Dazu kommt eine breite Auswahl an Stellschrauben, die dem Benutzer eine bessere Steuerung und Erleichterung des Visualisierungsprozesses anbieten kann. Bereits in Abbildung 5.1 ist dies erkennbar durch den wechselseitigen Datenfluss zwischen der Interaktion und den Daten. Darunter ist das Speichern und Laden der GPU-Daten, um die Wartezeit zu verkürzen, die Veränderung von Layout-Parametern, um bessere Ergebnisse zu erzielen, die Selektion von Objekten und die Veränderung der virtuellen Kamera, um den Graphen in seiner Gesamtheit oder an verschiedenen Stellen genauer betrachten zu können. Abschließend wird das Konzept der asynchronen Berechnung von Bounding Volumes vorgestellt.

5.6.1 Speichern und Laden

Das Preprocessing der Eingabedaten führt bei hohen Datenmengen unweigerlich zu hohen Wartezeiten, bevor der Graph das erste Mal gezeichnet werden kann. Um diese Wartezeit zu verkürzen, können die vorbereiteten Adjazenzdaten in einer Datei persistiert werden, um bei Start der Anwendung direkt hochgeladen werden zu können. Eine Möglichkeit den Zwischenschritt des Kopierens der Daten in den Hauptspeicher zu überspringen, ist die Verwendung von Memory-Mapped Files. Jedoch hat dieser Ansatz einige Beschränkungen, wie die maximale Größe, die durch die Länge der verwendeten Adressen fest vorgegeben ist.

Die minimal zu speichernden Daten bestehen aus den zwei Arrays `nodeData` und `linkData` und deren Längen, der Knotenzahl $|\mathcal{V}|$ und der gesamten Kantenzahl $|\mathcal{E}|$.

Außerdem kann die hier beschriebene Funktion ohne großen Zusatzaufwand um das Speichern des Layouts erweitert werden, wodurch ein bereits gefundenes Layout gesichert und später wiederverwendet werden kann. Das lässt sich natürlich auf sämtliche Daten übertragen, deren erneute Berechnung besonders zeitaufwändig wäre.

Die Datenverwaltung in Form vom Speichern und Laden der Rohdaten fällt aus dem in 5.1 gezeigten Rahmen, da nicht nur Attribute und das Layout, sondern auch die Adjazenzliste und gegebenenfalls Hilfsdaten abgespeichert werden können und der Ladevorgang das Preprocessing ersetzt.

5.6.2 Veränderung von Layout-Parametern

Kapitel 3 zeigt, dass die verschiedenen Layout-Verfahren Parameter benötigen, die nicht ohne Weiteres klar sind. Teilweise kann eine automatische Bestimmung erfolgen, jedoch sollte die Freiheit der endgültigen Konfiguration beim Benutzer liegen, falls die gelieferten Parameter nicht gut genug sind. Dazu könnte zum Beispiel die Temperatur von FDL-Verfahren zählen, deren Verlauf über verschiedene Modi und Randparameter, wie Start- und Endtemperatur oder Abkühlungs- und Erhitzungskoeffizienten. Aber auch eine Online-Regelung eines sonst konstanten Temperaturwerts mittels Angabe einer Änderungsstärke und der schrittweisen Anpassung ist vorstellbar.

5.6.3 Selektion

Bei der Beschreibung von Knoten- und Kantenattributen in Abschnitt 5.1.2 wurde implizit angenommen, dass Interaktion mittels Selektion möglich ist. Dieses wichtige Hilfsmittel ermöglicht ein breitgefächertes Potential für die Interaktion mit dem Visualisierungsprozess. Es folgenden drei einfache Selektionskonzepte.

Die einfachste Art der Selektion ist die Selektion eines einzelnen Objekts, das heißt einem Knoten oder einer Kante. Existieren zu den in der Visualisierung umgesetzten Daten noch zusätzliche, die nach Identifikation eines interessanten Objekts von Bedeutung sind, könnten diese mittels Selektion für das ausgewählte Objekt erscheinen. Steht ein Knoten beispielsweise jeweils für einen weiteren Graphen (vergleiche mit der Hierarchisierung aus Kapitel 4), kann die Selektion eine Vergrößerung der Ausdehnung des Knotens anstoßen, um anschließend im Inneren den enthaltenen Graphen zu visualisieren. Ein weiteres Anwendungsbeispiel wäre die Hervorhebung aller inzidenten Kanten des ausgewählten Knoten, um diese in einem Kantenwirrwarr zu erkennen.

Je größer die selektierbare Menge von Objekten, desto mühseliger ist die Selektion von Objektgruppen. Eine Möglichkeit dies mittels einer Position umzusetzen ist das Konzept der kaskadierenden Selektion. Es werden solange benachbarte Knoten und/oder Kanten in alle Richtungen ausgewählt, bis ein vordefinierter Threshold erreicht ist. Nachbarschaft kann mittels Adjazenz oder räumlicher Nähe festgestellt werden. Der Threshold ist in diesen Fällen die Pfadlänge respektive Anzahl selektierter Objekte.

Eine Alternative mehrere Elemente gleichzeitig auswählbar zu machen, ergibt sich mittels Freihand-Selektion. Der Benutzer markiert einen Bereich, die darin enthaltenen Objekte werden markiert.

Zu den in Abschnitt 5.1.2 vorgestellten Aktionen der Ausblendung, dem Ausschluss aus dem Layout-Verfahren, der Fixierung und der schlichten Markierung, die zur Umsetzung von Selektion benötigt wird, kommt eine weitere wichtige Interaktionsmöglichkeit: Das Erstellen eines manuellen Layouts. Mittels Selektion kann der Benutzer das aktuelle Layout verändern, indem er Knotengruppen markiert, verschiebt und anschließend fixiert. Möchte man zuvor bestimmte Knotengruppen auf deren Interkanten überprüfen, bietet sich ein manuelles Layout an.

Es besteht die Möglichkeit das Problem der Selektion von selbst OpenGL lösen zu lassen. Jedoch wird die Verwendung des Selection Buffers oder dem Feedback Mode nicht empfohlen [Opeb]. Falls keine Umsetzung mit Color Picking möglich ist, müssen benutzerdefinierte geometrische Lösungen eingesetzt werden. Color Picking arbeitet direkt auf der Anzeige. Jedes Objekt wird mit einer einzigartigen Farbe gerendert, sodass die Farbe an der gewählten Position $p \in \mathbb{N}^2$ abgelesen werden kann, um das jeweilige Objekt zu identifizieren.

Da die Objekte der darzustellenden Szene hier mit Position und Bounding Volume definiert sind, besteht eine einfache Methode darin, für alle Knoten zu testen, ob deren aufgespanntes Volumen p enthält. Da jedoch Darstellungsebene und Anzeige verschieden sind, muss p zunächst rücktransformiert werden. Für diese Transformation wird eine homogene Koordinate $q \in [-1, 1]^4$ benötigt, die p im Einheitswürfel beschreibt. Die Koordinaten von

$p \in \{0, \dots, m-1\} \times \{0, \dots, n-1\}$ sind gegeben. Die bereits normalisierte z-Koordinate erhält man durch Lesen des Tiefenwerts von p. Nach Anwendung der inversen Kameratransformation auf q kennt man die Position von p im Koordinatensystem des Layouts.

5.6.4 Kameratransformation

Bereits in der Einleitung wurde darauf hingewiesen, dass die Sichtperspektive auf die Visualisierung eine einfache Art der Fokussierung darstellt. Für $\mathcal{R} = \mathbb{R}^2$ bestehen die zugehörigen Ansatzpunkte aus der Bewegung in der Ebene und der Skalierung der Szene zusammen mit einer automatischen Einstellung, sodass die Ausmaße des globalen Bounding Volumes mit der Anzeige abschließen.

Hat das Knotenlayout eine besonders große Ausdehnung, ergibt sich ein Problem bei der naturgetreuen Abbildung von Größenverhältnissen. Knotenrepräsentationen werden zu klein, Kanten zu dünn, sodass nur ein Bruch der Verhältnis Abhilfe schafft. Der Benutzer entscheidet zu jedem Zeitpunkt, ob er die tatsächlichen oder kompensierte Größenverhältnisse in der Visualisierung umgesetzt haben möchte. Gegebenenfalls eignet sich auch hier ein stufenweise einstellbarer Parameter für Knoten und Kanten getrennt, eventuell sogar in Verbindung mit Selektion.

Für $\mathcal{R} = \mathbb{R}^3$ erfolgt aufgrund der Erhöhung der Dimension eine Steigerung der Freiheitsgrade, sodass Rotation der Perspektive verschiedene Blickwinkel auf dieselbe Szene bietet.

5.6.5 Berechnung der Bounding Volumes

Im Prinzip handelt es sich bei dieser Funktion um eine Vorbereitung für andere Funktionen, insbesondere für die Layout-Berechnung, falls diese die Bounding Volumes miteinbezieht. Dabei kann dies ein einfaches Setzen auf konstante Werte oder eine aufwändige Berechnung aufgrund zugehöriger Texturen oder anderer Daten sein. Insbesondere ermöglicht die Verlagerung der Berechnung in den interaktiven Bereich eine Veränderung des Inhalts und die damit verbundene Veränderung des Bounding Volumes.

6 Implementierung

Bereiche	Funktionen
Entkopplung	Iteration und Zeichnen sind nicht aneinander gekoppelt, jedoch herrscht eine sequenzielle Abfolge
Visualisierung	Kreisförmige Knoten mit Farbverlauf und Alpha-Blending
	Gleichförmig gekrümmte ungerichtete Kanten mit Alpha-Blending
	Bounding Boxes als gestrichelte Rechtecke
Datenakquise und Preprocessing	Generierung von vollständigen und zufälligen Graphen, Ring- und Sterngraphen
	Einlesen von CSV-ähnlichen Dateieformaten
	Unterstützung von Graphen ohne Mehrfachkanten
Layout-Verfahren	Fruchterman-Reigold in $\mathbb{R}^2$
	Berücksichtigung von AABB der Knoten (Variante 1 und 2)
Interaktion	Speichern und Laden
	Manuelle Temperatursteuerung
	Temperaturmodi (Abkühlen, Erhitzen mit Abkühlung, periodisches Erhitzen) über Konfigurationsdatei
	2D-Kameratransformationen in 3D (Verschiebung in der Ebene, automatischer Zoom)
	Einzelselektion zum Fixieren von Knotenpositionen

Tabelle 6.1: Funktionsüberblick der Implementierung

Bei der Umsetzung von Geistesgut in die Realität stößt man häufig auf Probleme, die man vorher so nicht in Betracht gezogen hätte. Rückwirkend hat eine Implementierung also den Effekt falsche Vorstellungen zu verbessern. In der Gegenrichtung stellt eine Implementierung die Möglichkeit zur Verfügung, Messungen von Speicherverbrauch und Laufzeit vorzunehmen. Daher gehört zu dieser Arbeit eine Implementierung, die das oben beschriebene Framework teilweise realisiert. Entwickelt wurde das Framework in der Programmiersprache C++ mit Hilfe der Grafik-API OpenGL. Tabelle 6.1 zeigt eine Zusammenfassung der wichtigsten Funktionen, die in der Implementierung verwirklicht wurden. Die folgenden Beschreibungen vertiefen einige der in Kapitel 5 vorgestellten Konzepte mit deren tatsäch-

licher Umsetzung und damit verbundener Probleme. Dazu gehören die Datenstrukturen, der Einsatz von Compute-Shadern und Reduktions-Programmen. Außerdem eine kurze Beschreibung der Einbettung von Fruchterman-Reigold.

6.1 Speicherlayout und Datenstrukturen

Generell ist beim Hochladen von Daten auf die GPU Einiges zu beachten. Bei Arrays und Verbundtypen spielt die Speicherausrichtung eine Rolle. Mit dem Speicherlayout `std140` werden Array-Elemente mit Vielfachen der Größe eines vierdimensionalen Vektors **`vec4`** einfacher Genauigkeit ausgerichtet und Verbund-Elemente absteigend ihrer Größe, jedoch in der Reihenfolge der Deklaration [Opea]. `packed` richtet die Daten lückenlos an, um Speicher zu sparen und erlaubt dem Compiler unbenutzte Teile wegzuoptimieren. Dies führt bei der Verwendung von Arrays in einem Compute Shader zu Fehlern, da die Elemente unter Umständen nicht mehr gleich groß sind und je Shader andere Optimierungen vorgenommen werden. Mit dem Speicherlayout `shared` wird diese Optimierung nicht durchgeführt und damit bietet es eine halbwegs effiziente und benutzbare Option.

Code 6.a: Datenstruktur für zweidimensionale Vektoren

```
1  struct PackedVec2
2  {
3    float X;
4    float Y;
5  };
```

Die Standard-Vektordatenstrukturen `vec`d sind über die Programmbibliothek GLM auch Client-Seitig benutzbar. Jedoch bestehen diese Strukturen hier nicht nur aus d Feldern, wodurch das Hochladen eines statischen Arrays dieser Strukturen zu Fehlern führt. Daher muss ein eigener Datentyp angelegt werden, der lediglich d Fließkommazahlen beinhaltet. Für $\mathcal{R} = \mathbb{R}^2$ zeigt Code 6.a die zugehörige Datenstruktur.

Code 6.b: Knotendatenstruktur

```
1  const uint DEFAULT = 0u;
2  const uint DISABLE_DRAW = 1u;
3  const uint DISABLE_LAYOUT = 2u;
4  const uint FIXED_X = 4u;
5  const uint FIXED_Y = 8u;
6  const uint PICKED = 16u;
7
8  struct NodeData
9  {
10   uint ExternalNode;
11   uint LinkIndex;
12   int Degree;
13   uint Flags;
14 };
```

Die Knoten werden im Speicher von Instanzen der Datenstruktur `NodeData` repräsentiert (siehe Code 6.b). `ExternalNode`, `LinkIndex`, `Degree` und `Flags` halten jeweils die Informationen

ext i, k, deg i beziehungsweise das Bitfeld für die Knoten-Flags (vergleiche mit der Notation aus Abschnitt 5.1).

Code 6.c: Kantendatenstruktur

```
const uint DEFAULT = 0u;
const uint DISABLE_DRAW = 1u;
const uint DISABLE_LAYOUT = 2u;

struct LinkData
{
  uint SourceNode;
  uint TargetNode;
  float Weight;
  float LocalFactor;
  uint Flags;
};
```

Instanzen der Datenstruktur `LinkData` (siehe Code 6.c) ergeben aneinandergereiht das Kanten-Array. Neben den Knotenzeigern i, j als `SourceNode` und `TargetNode` ist die Kantendatenstruktur aus dem Kantengewicht ω, dem Faktor α als `Weight` beziehungsweise `LocalFactor` aufgebaut und wird, wie bei den Knoten, mit einem Bitfeld `Flags` abgeschlossen (vergleiche erneut mit der Notation aus Abschnitt 5.1). Der Vergleich der beiden zugehörigen Flag-Listen zeigt, dass der Fokus auf der Interaktion mit Knoten liegt.

Code 6.d: Datenstruktur für Bounding Boxes

```
struct Box
{
  PackedVec2 BottomLeft;
  PackedVec2 TopRight;
};

struct CenteredBox
{
  PackedVec2 Center;
  Box Box;
};
```

Die Umsetzung der AABBs orientiert sich an der Definition aus 2 und berücksichtigt, die angesprochene Array-Problematik. Code 6.d zeigt die resultierende Datenstruktur `Box` für einfache Bounding Boxes und `CenteredBox` für positionierte Bounding Boxes, wie der globalen Bounding Box.

Code 6.e: Server-seitiger Speicherzugriff

```
1  // Layout
2  layout(shared, binding = 0)
3    buffer block_Layout{ PackedVec2 positions[]; };
4
5  // Adjazenzliste
6  layout(shared, binding = 1)
7    buffer block_NodeData { NodeData nodeData[]; };
8
9  layout(shared, binding = 2)
10   buffer block_LinkData { LinkData linkData[]; };
11
12 // Hilfsdaten
13 layout(shared, binding = 3)
14   buffer block_Forces { PackedVec2 forces[]; };
15
16 layout(shared, binding = 5)
17   buffer block_Temp { coherent Box tempBoxes[]; };
18
19 // Weitere Attribute
20 layout(shared, binding = 4)
21   buffer block_Boxes
22   {
23     coherent CenteredBox globalBox;
24     Box boxes[];
25   };
26
27 // Transformation
28 layout(shared, binding = 6)
29   buffer block_Camera { Camera camera; };
```

Beim Zugriff auf diese Datenstrukturen kommt der flexible Pufferzugriff auf Shader Storage Buffer Objects (SSBOs) zum Einsatz. Jedem Interface Block wird eine Bindungsadresse `binding` zugeordnet (siehe Code 6.e), mit der er Client-seitig identifiziert werden kann. Dort kann der Speicher allokiert und gegebenenfalls dabei initialisiert werden. Auf Server-Seite erfolgt der Zugriff direkt über die Variablen und der `[.]`-Indizierung. In Code 6.e sind alle Interface Blocks enthalten, die diese Variablen bereitstellen.

Das Schlüsselwort **`coherent`** markiert Variablen, die durch Pufferbarrieren geschützt werden können. Dies hat allerdings auch zur Folge, das diese nicht als Parameter von Funktionen übergeben werden dürfen.

6.2 Linearisierung und der Einsatz von Compute-Shadern

Die Linearisierung des Speichers durch die Verwendung von SSBOs erzeugt ein neues Problem bei der Entsendung von Compute-Shader-basierten Programmen. Die Verteilung auf virtuelle Recheneinheiten erfolgt in Workgroups der Größe (x_w, y_w, z_w), die einen dreidimensionalen Raum gleichmäßig unterteilen. Das heißt für eine gegebene Instanz-Zahl $n_0 \in \mathbb{N}$ müssen auf Client-Seite für eine zweidimensionale Verteilung (x, y) Workgroups

gestartet werden mit

$$\begin{aligned} w &= \frac{c}{x_w \cdot y_w}, \\ x &= (w \bmod W_x) + 1, \\ y &= \left\lfloor \frac{w}{W_x} \right\rfloor + 1, \end{aligned}$$

wobei W_x die maximale Anzahl an Workgroups ist, die in x-Richtung Platz findet. Server-seitig muss aus der gelieferten Workgroup-ID (x_{wid}, y_{wid}), der lokalen ID (x_{id}, y_{id}) und der Größe (x_w, y_w) die ursprüngliche ID $id = (x_{wid} + y_{wid}x) \cdot x_w y_w + y_{id} x_w + x_{id}$ abgeleitet werden. Da n_0 nicht immer perfekt auf die Workgroups verteilt werden kann, ergibt sich ein Überschuss an Aufrufen, die mittels $id \geq n_0$ ignoriert werden müssen.

6.3 Ein Beispiel für ein Reduktions-Programm

Da Layout-Algorithmen, insbesondere FDL-Algorithmen, die Knotenpositionen von Iteration zu Iteration immer weiter weg vom Mittelpunkt setzen könnten, muss, um einem Überlauf vorzubeugen, diese globale Verschiebung rückgängig gemacht werden. Aus den Bounding Boxes der Knoten lässt sich eine globale Bounding Box bestimmen, deren Zentrum $a_0 \in \mathbb{R}^2$ gegebenenfalls nicht in der Mitte $m = (0, 0)$ liegt. Wird das gesamte Layout um $-a_0$ verschoben befindet sich dieses Zentrum in der Mitte und die Bounding Box wird symmetrisch.

Die globale Bounding Box wird, wie im Titel angekündigt, mittels Reduktion Server-seitig bestimmt. Dabei werden $n = \min\{n_0, {}^1\!/\!_2\, |\mathcal{V}|\}$ Programm-Instanzen erzeugt für ein $n_0 \in \mathbb{N}$, die jeweils für die Berechnung der globalen Bounding Box von ${}^1\!/\!_n\, |\mathcal{V}|$ Knoten zuständig sind. Dabei gilt

$$\widetilde{B}_k := \max_{i=s,\dots,e} B_i$$

mit

$$\begin{aligned} s &:= kn + 1, \\ e &:= \begin{cases} (k+1)\,n & k+1 < n, \\ |\mathcal{V}| & \text{sonst.} \end{cases} \end{aligned}$$

für $k = 1, \dots, n$. Die n Ergebnisse werden in einem temporären Array abgelegt. Die letzte Programm-Instanz übernimmt dann die Aufgabe, aus diesen das absolute Maximum, also die globale Bounding Box zu berechnen. Neben der Berechnung der Bounding Box wird auch deren Zentrum bestimmt, mit dem anschließend eine Zentrierung durchgeführt werden kann. Anschließend kann anhand der aktuellen Vertex-Transformationen berechnet werden, um wie viel die Kamera orthogonal zu Darstellungsebene verschoben werden muss, damit die Szene komplett dargestellt wird. Insgesamt betrachtet wird also ein automatischer Zoom realisiert.

6.4 Einbettung von Fruchterman-Reigold

Das Layout-Verfahren, das in die Implementierung eingebettet wurde, ist der FDL-Ansatz Fruchterman-Reigold. Dabei wurde auf jegliche Laufzeitoptimierung durch Hierarchisierung und ähnliches verzichtet, das heißt das Laufzeit verhalten ist in $\mathcal{O}\left(|\mathcal{V}|^2 + |\mathcal{E}|\right)$. Ein Knoten-Programm, wie es in Abschnitt 5.2 beschrieben wurde, führt für jeden Knoten parallel die Berechnung der wirkenden Kräfte durch, die in einem weiteren Programm gleichen Typs mit einer Temperatur begrenzt werden, bevor sie als Verschiebung der einzelnen Knotenpositionen das neue Layout bestimmen. Die Berücksichtigung der AABB findet in ersterwähntem Programm statt.

Im Zusammenhang mit der Temperatur aus Abschnitt 3.2.2 ergab sich die voraussehbare Beobachtung, dass ab einem genügend kleinen Wert eine flüssige Bewegung anstatt einem undurchsichtigen Wirrwarr eintritt. Dies geschieht für uniforme Kantengewichte mit $\forall \{i, j\} \in \mathcal{E} : \omega(i, j) = 1$ ab einem Wert von 0.001. Da das Kantengewicht in die Berechnung der Kräfte als Faktor miteinbezogen wir, verändert sich dieser Wert für andere Kantengewichte, sodass deren Produkt eine ähnliche Zahl ergibt.

Die unbegrenzte Darstellungsebene erzeugt bei vielen Knoten und wenig Kanten eine enorme Ausdehnung des Layouts, sodass, um den gesamten Graphen zu betrachten, so viel Abstand genommen werden muss, dass die Strukturen nicht mehr sichtbar sind. Dieser Umstand wurde bereits in Abschnitt 5.6.4 erwähnt. Die Wahl besonders hoher Kantengewichte führt alternativ zur visuellen Anpassung zu einer Verbesserung, da ein höheres Kantengewicht in dieser Implementierung dafür sorgt, dass die Anziehungskraft zwischen verbunden Knoten stärker ist und sie damit näher aneinander rücken.

7 Evaluation und Ausblick

Über den Entwicklungsprozess hinaus bietet eine Implementierung die Möglichkeit der Evaluation durch Testläufe. Dabei wurde die Speicherkomplexität überprüft und das Laufzeitverhalten gemessen. Außerdem wurden unerwartet niedrige, teils künstliche, Grenzen vorgefunden, die hier erörtert werden. Der zweite Teil dieses Kapitels schließt die Arbeit ab, indem auf die Inhalte der Arbeit zurückgeblickt wird und mögliche Fortsetzungen und Ideen für die Zukunft angerissen werden.

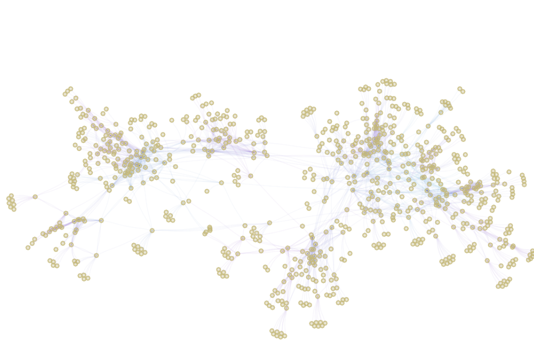
7.1 Visualisierung

In der Einleitung wurden drei Beispiele realer Daten genannt, die sich als Node-Link-Diagramm visualisieren lassen. Einer dieser Graphen ist der Amazon-Graph, der die Analyse von Verkaufsdaten enthält. Wie der gesamte Graph in der Implementierung dargestellt wird, lässt sich leider nicht zeigen. Gründe dafür finden sich in nachfolgenden Kapiteln. Die folgenden Abbildungen zeigen, wenn nicht anders erwähnt, einen Teilgraphen mit $n = 51\,200$.

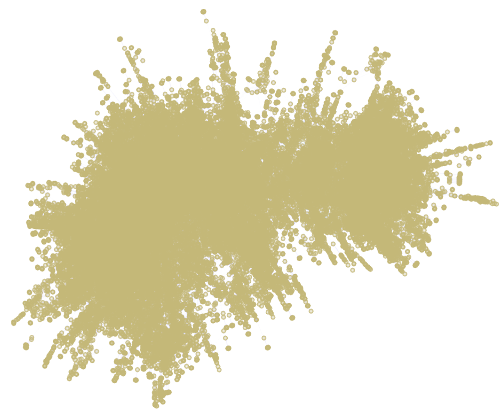
Abbildung 7.1a zeigt einen kleineren Teilgraphen mit $n = 800$, der die Struktur eines skalenfreien Graphen vorführt. Die Struktur ähnelt der des umschließenden Graphen, der in Abbildung 7.1b abgebildet ist. Man sieht bei beiden Ergebnissen die Problematik der Layout-Ausdehnung, die mittels kompensierter Geometrie gelöst wurde. Die anderen vier Bilder zeigen Strukturen, die innerhalb des Graphen auftreten. In Abbildung 7.1c sieht man einen Knoten hohen Grades, der für ein weitreichendes Produkt steht, das im Zusammenhang mit dem Kauf vieler anderer Produkte steht. Abbildung 7.1d und 7.1e zeigen kleine Gruppen, die eng miteinander verbunden sind. Letztere Struktur ist ein nahezu vollständiger Subgraph. Dabei handelt es sich um Produkte, die in einem engen Kontext vorkommen und/oder von einander abhängen. Am Rand des Graphen finden sich viele kleine Baumstrukturen, wie in Abbildung 7.1f, die Einzelabhängigkeiten darstellen.

7.2 Speicherkomplexität

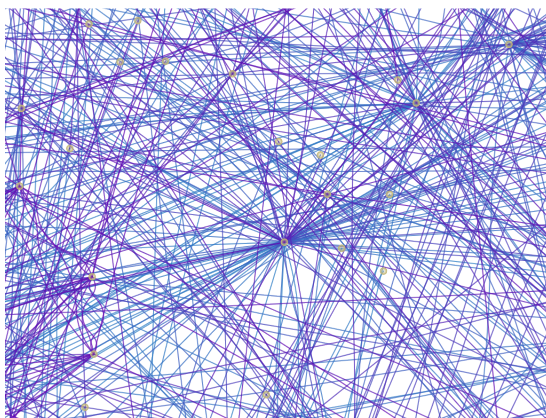
Welche Datenmengen mit den in Abschnitt 5.1 beschriebenen Datenstrukturen erreicht werden können, wird ein Blick auf deren Speicherkomplexität verdeutlichen. So werden für einen einzelnen Knoten $4 \cdot 4 = 16\,\text{B}$, für eine einzelne Kante $5 \cdot 4 = 20\,\text{B}$ Speicherplatz benötigt. Dazu kommen pro Knoten eine Bounding Box mit $2 \cdot 2 \cdot 4 = 16\,\text{B}$, die aktuell wirkende Kraft und dessen Position, jeweils mit $2 \cdot 4 = 8\,\text{B}$.



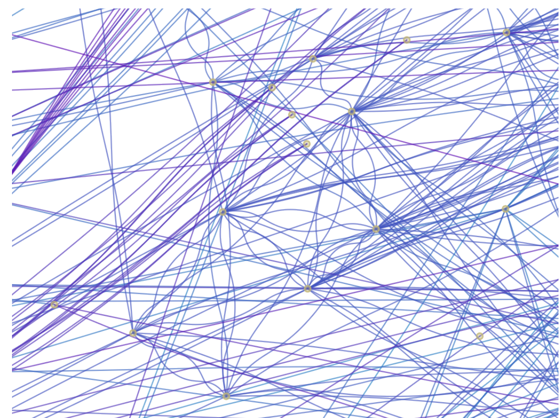
(a) Ausschnitt des Amazon-Graphen mit $n = 800$



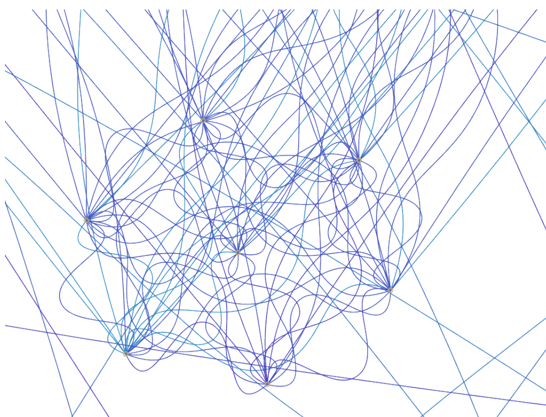
(b) Ausschnitt des Amazon-Graphen mit $n = 51\,200$



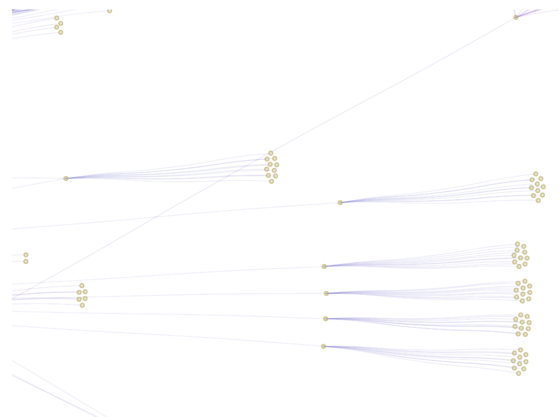
(c) Ein Knoten mit hohem Grad



(d) Eine eng verbundene Knotenmenge

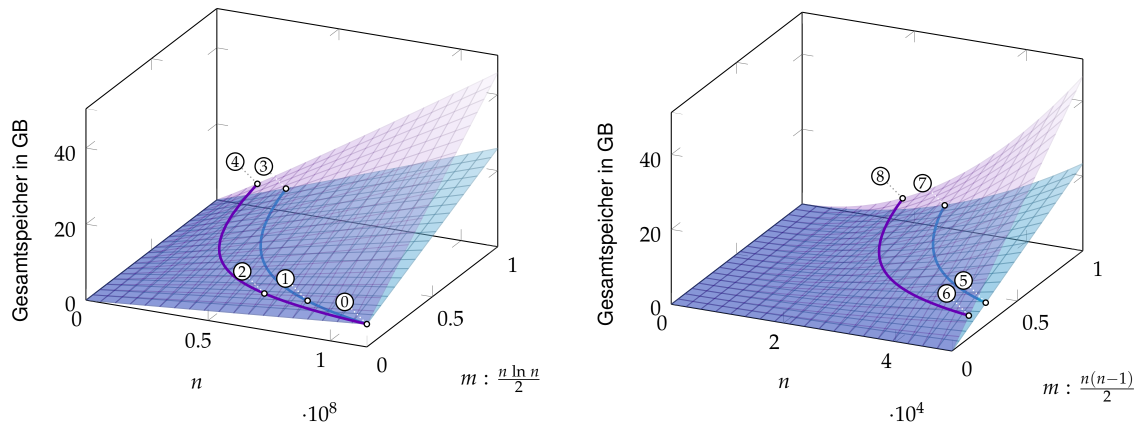


(e) Ein fast vollständiger Subgraph



(f) Baumstrukturen am Rand des Graphen

Abbildung 7.1: Visualisierung des Amazon-Graph



(a) Für dünn besetzte Graphen

(b) Für beliebige Graphen

Abbildung 7.2: Speicherkomplexität der konzipierten Datenstrukturen

k	n	m	m_s	$S_\mathcal{V}$	$S_\mathcal{E}$
0	$1.15 \cdot 10^8$	0	$0, \ldots, m$	5.14 GB	0 GB
1	$8.48 \cdot 10^7$	n	0	3.79 GB	1.58 GB
2	$6.71 \cdot 10^7$	n	m	3 GB	2.5 GB
3	$2.83 \cdot 10^7$	$2.43 \cdot 10^8$	0	1.27 GB	4.53 GB
4	$1.66 \cdot 10^7$	$1.38 \cdot 10^8$	m	0.74 GB	5.14 GB
5	$5 \cdot 10^4$	$3.22 \cdot 10^8$	0	2.29 MB	5.99 GB
6	$5 \cdot 10^4$	$1.61 \cdot 10^8$	m	2.29 MB	5.99 GB
7	$2.54 \cdot 10^4$	$3.22 \cdot 10^8$	0	1.16 MB	5.99 GB
8	$1.79 \cdot 10^4$	$1.61 \cdot 10^8$	m	0.82 MB	5.99 GB

Tabelle 7.1: Beispieleingaben für die volle Ausnutzung von 6 Gigabyte

Sei $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \omega)$ eine abstrakte Repräsentation der Eingabedaten. Dann lässt sich der Speicherverbrauch von $n = |\mathcal{V}|$ Knoten eindeutig bestimmen als $S_{\mathcal{V}}(n) = (2 \cdot 16 + 2 \cdot 8)\, n = 48n$ B. Bei den Kanten ist die Bestimmung problematisch, da die Anzahl dieser nicht nur von der Anzahl der Kanten, sondern auch von der Verteilung der Kanten abhängt. Neben $m = |\mathcal{E}|$ bestimmt ein weiterer Parameter $m_s = |\mathcal{E}_s|$ die Speicherkomplexität, der die Anzahl der Einzelkanten $\mathcal{E}_s := \{ (i,j,k) \in \mathcal{E} \mid \forall k \in \mathbb{N} : (j,i,k) \notin \mathcal{E} \}$ der Eingabe angibt. Der benötigte Speicher beträgt dann $S_{\mathcal{E}}(m, m_s) = 20\,(m - m_s) + (2 \cdot 20)\, m_s = 20\,(m + m_s)$ B. Für einen ungerichteten Graphen gilt in jedem Fall $m_s = m$. Die globale Bounding Box fällt mit konstant $S_B = 16 + 8 = 24$ B kaum ins Gewicht. Zeitweise kommt allerdings zu diesem grundsätzlichen ein temporärer Speicherbedarf von $S_t(n) = 16 \cdot {}^{n}/_{2} = 8n$ B dazu, um diese zu berechnen. Insgesamt ergibt dies einen Speicherverbrauch von $S(n, m, m_s) = S_{\mathcal{V}}(n) + S_{\mathcal{E}}(m, m_s) + S_B + S_t(n) = 56n + 20\,(m + m_s) + 24$ B.

In Abbildung 7.2 ist der Speicherverbrauch abhängig von Knotenzahl n und Kantenzahl m in Gigabyte abgebildet. Dabei begrenzen die beiden Flächen jeweils den Bereich zwischen $m_s = 0$ und $m_s = m$. Um die Schaubilder besser einschätzen zu können, sind die möglichen Kombinationen hervorgehoben, die in 6 Gigabyte verbrauchten Speicherplatz resultieren. Sie begrenzen zusammen eine horizontale Fläche, in der die Werte für beliebige $0 < m_s < m$ vorzufinden sind. Anstelle von m ist in 7.2a das Verhältnis $m : {}^{n}/_{2} \ln n$ aufgetragen. Diese Verzerrung dient dem graphischen Hinweis auf die steigenden Möglichkeiten Kanten zu platzieren. Der Speicherverbrauch ist explizit für die fünf markierten Stellen $k = 0, \ldots, 8$ in Tabelle 7.1 aufgelistet. $k \in \{1, 2\}$ kennzeichnen jeweils Graphen mit $m = n$. Abbildung 7.2b zeigt, wie der Speicherverbrauch um einiges stärker wächst in Betracht der theoretischen Grenze $\mathcal{O}(n^2)$. Um eine vergleichbare Ausgangslage zu schaffen, musste die maximal dargestellte Knotenanzahl um vier Grade herabgesetzt werden und auch die Werte $k = 5, \ldots, 8$ in der Tabelle spiegeln die Problematik hochgradig vernetzter Graphen wieder, da in jedem Fall der Speicherverbrauch von den Kanten den Gesamtverbrauch dominiert wird und nur noch circa 10^4 Knoten Platz finden.

Tabelle 7.2 zeigt den Speicherverbrauch der exportierten Rohdaten für jeweils Teilgraphen des Amazon-Graphen und vollständigen Graphen für verschiedene $n = |\mathcal{V}|$. Es ist ein deutlicher Unterschied des Wachstums festzustellen. Für den Amazon-Graph gilt $m = |\mathcal{V}| < n \ln n$. So würde für $n = 12\,800$ unter Annahme von $m_s = 0$ ein Speicherverbrauch von $16 \cdot n + 20 \cdot n \ln n \approx 2.5\,\text{MB} > 1.6\,\text{MB}$ zustande kommen. Das quadratische Verhalten des vollständigen Graphen lässt sich gut erkennen. Eine Verdoppelung der Knotenzahl führt zu einer Vervierfachung des Speicherverbrauchs. Die theoretischen Berechnungen von oben lassen sich ebenfalls validieren. Für $n = 1\,600$ ergibt sich $16 \cdot n + 20 \cdot n\,(n-1) \approx 48.82$ für einen vollständigen Graphen, was dem Wert der Tabelle entspricht.

Bei der Betrachtung des Speicherverbrauchs wurde bisher vernachlässigt, dass nicht der gesamte Speicher frei verfügbar ist, denn auch andere Programme und das Betriebssystem benötigen Grafikspeicher und neben der Haltung der Daten wird auch temporärer Speicher für die Tessellierung gebraucht, zum Beispiel. Nichtsdestotrotz wird die Anwendung von andere Seite früher limitiert. So führte der Versuch des Uploads von Daten, genauer die des vollständigen Amazon-Graphen, auf Grafikhardware mit geringem Hauptspeicher zu einer Ausnahme. Der Treiber verbietet anscheinend zu große Speicherallokationen und verhindert damit die volle Ausnutzung des freien Speichers. Desweiteren zeigte sich bereits beim

Graph	n	Speicherverbrauch
Amazon-Graph	100	8 384 B
	200	19 304 B
	400	44 584 B
	800	102 464 B
	1 600	203 104 B
	3 200	410 144 B
	6 400	848 384 B
	12 800	1 687 424 B
	25 600	3 369 024 B
	51 200	7 010 464 B
	⋮	⋮
	alle	104 190 632 B
Vollständiger Graph	100	199 608 B
	200	799 208 B
	400	3 198 408 B
	800	12 796 808 B
	1 600	51 193 608 B
	3 200	204 787 208 B

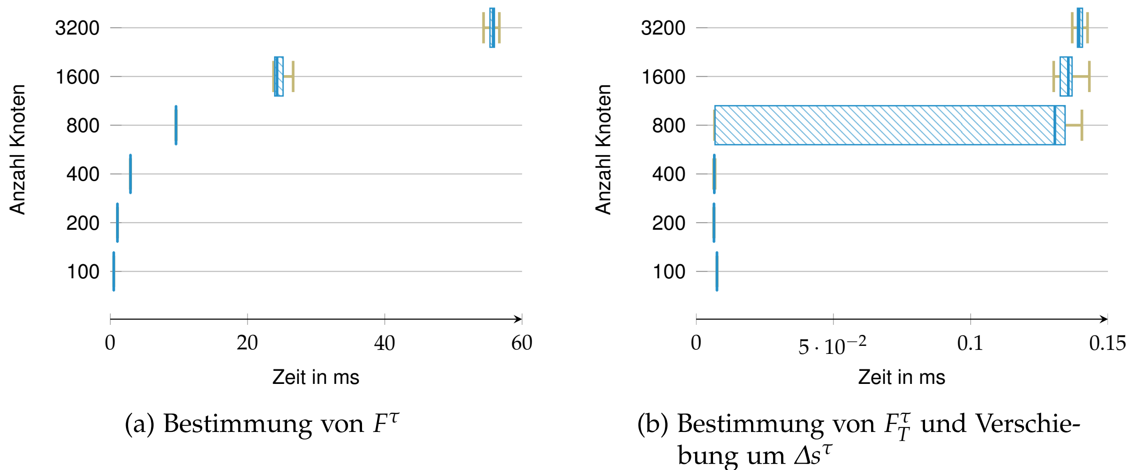
Tabelle 7.2: Speicherverbrauch gemessen an der Größe der gespeicherten Rohdaten der Adjazenzliste

Preprocessing eines vollständigen Graphen mit $n = 6\,400$ eine weitere Grenze. Die Clientseitige Allokation war nicht möglich, da nicht genügend zusammenhängender Speicher auf der Halde zur Verfügung stand. Bei 32 GB Hauptspeicher ein fragliches Ergebnis. Jedoch begrenzt der Adressraum die Ansteuerung auf Speicher unter 4 GB. Zusammen mit dem Anspruch der lückenlosen Allokation sind das die Gründe des Fehlers.

7.3 Laufzeitanalyse

Die Laufzeit der Implementierung wurde auf einem System mit einer GeForce GTX TITAN gemessen. Abbildung 7.3 bis 7.6 zeigen die Ergebnisse für verschiedene Programmkomponenten, sowie eine Gesamtlaufzeit für 100 Iterationen. Die Eingabe ist jeweils ein vollständiger Graph mit n Knoten. Die Einzelmessungen sind während der Gesamtmessung ausgeschaltet.

Bei der Überprüfung des Laufzeitverhaltens trat erneut eine Grenze auf. Auf dem verwendeten Betriebssystem Microsoft Windows 8 gibt es ein Timeout für Berechnungen auf der GPU von 2 Sekunden, die deutlich überschritten werden für schon kleine Datenmengen, wie dem Teilgraph des Amazon-Graphen mit $n = 51\,200$.



(a) Bestimmung von F^τ

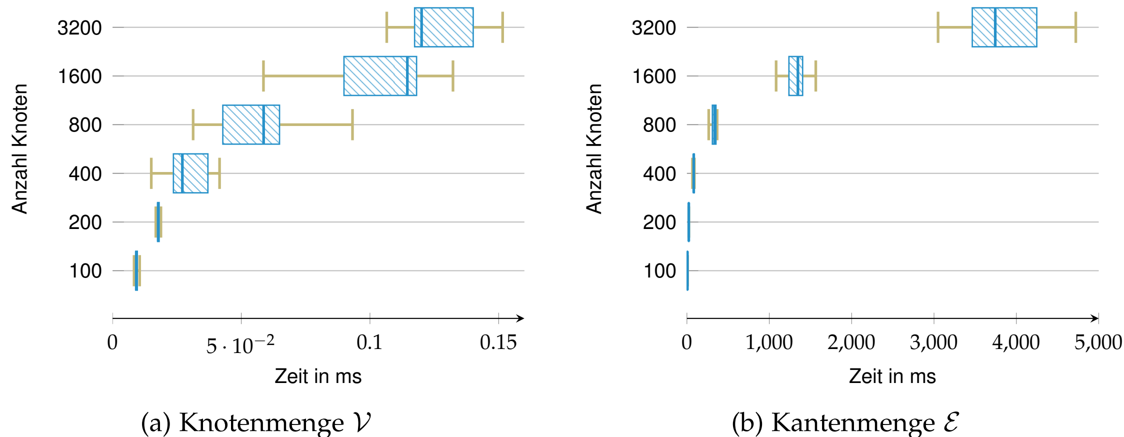
(b) Bestimmung von F_T^τ und Verschiebung um Δs^τ

Abbildung 7.3: Messergebnisse für vollständige Graphen: Einzelne Iteration von Fruchterman-Reigold

Abbildung 7.3a und 7.3b zeigen die Laufzeit des parallelisierten Fruchterman-Reigold pro Iteration. Da die Verschiebung in besonders kurzer Zeit durchführbar ist, erkennt man einen Sprung bei $n = 800$, der die bereits angesprochene Problematik der gleichzeitig freien Recheneinheiten bestätigt. Von da an muss unter Umständen mehr als nur ein paralleler Lauf gestartet werden. Dadurch ergibt sich eine breite Streuung, die für größere n wieder verschwindet, da die Zahl der Läufe mit steigender Wahrscheinlichkeit aufgeteilt werden muss.

Zu Beginn ist nicht erkennbar um welchen Proportionalitätsfaktor es sich bei Abbildung 7.3a handelt, ab $n = 200$ kann man jedoch eine Vervierfachung als Folge der Verdopplung

der Knotenzahl erkennen, die immer weiter abnimmt, bis auf $n = 3\,200$ nur noch eine Verdopplung stattfindet. Dies zeigt, das die Parallelisierung einen positiven Effekt auf die Laufzeit hat. Die Knotenzahl hängt bei einem vollständigen Graphen quadratisch mit der Knotenzahl n zusammen, da jedoch schon der bestimmende Faktor n^2 ist, fällt $m = n\,(n-1)$ bei der Laufzeit $\mathcal{O}\left(n^2 + m^2\right)$ von Fruchterman-Reigold nicht ins Gewicht. Der quadratische Zusammenhang des Verfahrens wird durch die Parallelisierung abgeschwächt. Für deutlich höhere n wird sich diese Verbesserung einstellen, da die Anzahl der Recheneinheiten konstant bleibt. Dies konnte nicht überprüft werden, da die in diesem und im vorherigen Abschnitt angesprochenen Grenzen gelten.



(a) Knotenmenge $\mathcal{V}$

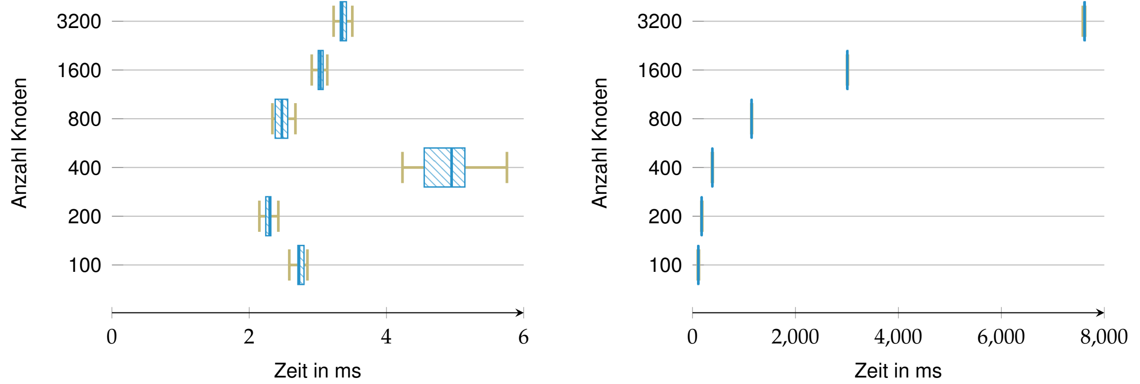
(b) Kantenmenge $\mathcal{E}$

Abbildung 7.4: Messergebnisse für vollständige Graphen: Rendering

Die Laufzeit des Rendering von Knoten und Kanten wurde ebenfalls gemessen. Siehe dazu Abbildungen 7.4a und 7.4b. Die Laufzeit der Knoten ist annähernd linear, teilweise sublinear, die der Kanten aufgrund des quadratischen Zusammenhangs zur aufgetragenen Größe zu Beginn quadratisch und gegen Ende etwas abnehmend. Weiterhin lässt sich erkennen, dass durch die aufwändigere Tessellierung der gekrümmten Kanten eine deutlich höhere Laufzeit zustande kommt als beim Rendering der kreisförmigen Knoten.

Die Gesamtlaufzeit der Initialisierung bietet auf den ersten Blick eine Überraschung. In Abbildung 7.5a nimmt die Laufzeit zunächst ab, bevor nach einem Ausreißer ein lineares Steigungsverhalten erkennbar wird. Auch hier wird durch die Parallelisierung für größere n Zeit eingespart. Das Absinken der Laufzeit zu Beginn lässt darauf schließen, dass bei geringen Datenmengen eine parallele Ausführung von Nachteil ist. Dieses Resultat und der Ausreißer könnten auf die Workgroup-Problematik aus Abschnitt 6.2 zurückzuführen sein. Das Schaubild in Abbildung 7.5b deckt sich, wie erwartet, mit dem Verhalten der Einzeliterationen.

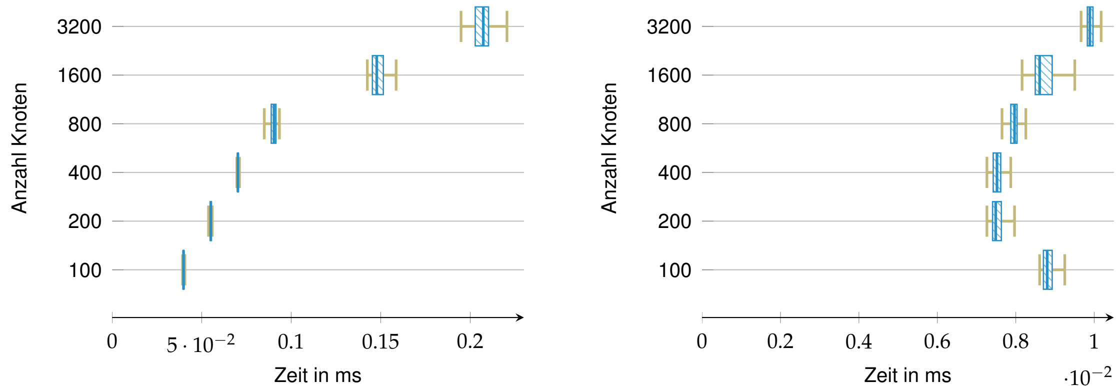
Es wurde außerdem die Laufzeit von Reduktions-Programmen gemessen, wie sie in Abschnitt 5.2 vorgestellt wurden. Ähnlich der Initialisierung zeigt sich hier für die aufwändigere Berechnung der globalen Bounding Box (Abbildung 7.6a) sublineares Verhalten mit zwei verschiedenen Steigungen. Erneut befindet sich der Übergang bei $n = 800$. Die Verschiebung der Knotenpositionen für die Zentrierung der Szene unterscheidet sich deutlich von der



(a) Bestimmung der Bounding Boxes $B_i \ \forall i \in \mathcal{V}$, des Initial-Layouts $\mathcal{L}^0$ und der globalen Bounding Box B_g

(b) 100 Iterationen ohne ohne Rendering

Abbildung 7.5: Messergebnisse für vollständige Graphen: Gesamtzeiten



(a) Bestimmung der globalen Bounding Box

(b) Zentrierung der Knotenpositionen

Abbildung 7.6: Messergebnisse für vollständige Graphen: Reduktions-Programme

Verschiebung durch eine Knoten-Programm (Abbildung 7.3b). Das Zeitverhalten ähnelt dafür dem der Initialisierung, nur ohne den Ausreißer.

7.4 Zusammenfassung

In den vorhergehenden Kapiteln wurde die Visualisierung abstrakter Daten als Node-Link-Diagramme beleuchtet. Basis bildet dabei die mathematische Struktur des Graphen, die aus verschiedenen Blickwinkeln betrachtet werden kann, um ein Knotenlayout zu bestimmen. Jedoch gibt es darüberhinaus weitere Aspekte, die bei der Visualisierung von Interesse sein können. Es wurde unter Berücksichtigung dieser Facetten ein Framework vorgestellt, das einen generellen Ansatz für das Zeichnen von Node-Link-Diagrammen darstellt, und auch teilweise implementiert. Diese Implementierung lieferte Aufschlüsse hinsichtlich der Laufzeit und Speicherkomplexität, insbesondere im Kontext der Parallelisierung durch Grafikhardware, aber auch hinsichtlich der Tauglichkeit des konzipierten Frameworks.

7.5 Ausblick

In der Einleitung wurde ein Diskussionsthema übersprungen, das die Anstrengungen effiziente Layout-Algorithmen zu konzipieren für sinnlos erklären könnte. Es spalten sich die Geister, inwieweit sehr große Datenmengen ohne jegliche Vorverarbeitung bei Analyseprozessen sinnvoll sind. Dieser Frage könnte man eine Arbeit widmen, die mittels Durchführung von Studien den aktuellen Trend erfassen soll. Bei dieser Gelegenheit bietet es sich auch an zu klären, welche Wünsche und Vorstellungen im Bereich des Graph-Drawing vorherrschen, um eine Basis für neue Ideen und Verfahren zu legen. Abschnitt 4.3 gibt einen kurzen Überblick über Software, die im Bereich des Graph-Drawings anzusiedeln ist. Wie dort bereits erwähnt, könnte auch hier eine genauere Untersuchung angebotener Funktionen und deren Umsetzung von Interesse sein.

Das vorgestellte Framework wurde hier nur teilweise implementiert, sodass bei Weitem nicht alle Probleme dieses generellen Ansatzes aufgedeckt werden konnten. Weitere Versuche, Layout-Algorithmen und Visualisierungsstrategien einzubetten, würden zu einer verbesserten, konkreteren Fassung führen, die sich, richtig umgesetzt, als ein nützliches Tool für verschiedene Themengebiete herausstellen könnte. Damit verbunden lohnt sich die Überlegung, wie dieses Framework mit anderer vorhandener Software eine Kommunikation über eine durchdachte Schnittstelle aufbauen kann.

Die Versuche, die zur Messung der Zeit- und Speicherkomplexität durchgeführt wurden, stellen einen weiteren Ansatzpunkt für zukünftige Arbeiten dar. Für eine taugliche Umsetzung mit ausführlicher Evaluation müssen die drei erfahrenen Grenzen der Speicherverwaltung und der Laufzeit überwunden werden. Denn was bringen hohe Speicher- und Rechenkapazität, wenn man sie nicht sinnvoll nutzen kann? Nach Beantwortung dieser Frage, rücken weitere neue Probleme ins Licht, wie der Grenze maximal darstellbarer Knoten-IDs.

Abschließend lässt sich sagen, dass die Entwicklung der Grafikhardware und deren Schnittstellen weitergehen wird und man so gespannt sein kann, was in Zukunft alles möglich sein wird.

Literatur

[AH07] Adel Ahmed und S-H Hong. „Navigation techniques for 2.5 D graph layout". In: *Visualization, 2007. APVIS'07. 2007 6th International Asia-Pacific Symposium on*. IEEE. 2007, S. 81–84.

[AM] David Auber und Patrick Mary. *Tulip – Better Visualization Through Research*. URL: `tulip.labri.fr/TulipDrupal` (besucht am 03. 11. 2013).

[BB03] Albert-László Barabási und Eric Bonabeau. „Scale-Free". In: *Scientific American* (2003).

[Ben75] Jon Louis Bentley. „Multidimensional binary search trees used for associative searching". In: *Communications of the ACM* 18.9 (1975), S. 509–517.

[Cui+08] Weiwei Cui u. a. „Geometry-based edge clustering for graph visualization". In: *Visualization and Computer Graphics, IEEE Transactions on* 14.6 (2008), S. 1277–1284.

[Ers+11] Ozan Ersoy u. a. „Skeleton-based edge bundling for graph visualization". In: *Visualization and Computer Graphics, IEEE Transactions on* 17.12 (2011), S. 2364–2373.

[FR91] Thomas M.J. Fruchterman und Edward M. Reingold. „Graph drawing by force-directed placement". In: *Software: Practice and experience* 21.11 (1991), S. 1129–1164.

[FT07] Yaniv Frishman und Ayellet Tal. „Multi-level graph layout on the GPU". In: *Visualization and Computer Graphics, IEEE Transactions on* 13.6 (2007), S. 1310–1319.

[FT08] Yaniv Frishman und Ayellet Tal. „Online dynamic graph drawing". In: *Visualization and Computer Graphics, IEEE Transactions on* 14.4 (2008), S. 727–740.

[Gep] Gephi.org. *Gephi – Makes graphs handy*. URL: `gephi.org` (besucht am 03. 11. 2013).

[God+09] Apeksha Godiyal u. a. „Rapid multipole graph drawing on the gpu". In: *Graph drawing*. Springer. 2009, S. 90–101.

[HK02] David Harel und Yehuda Koren. „Drawing graphs with non-uniform vertices". In: *Proceedings of the Working Conference on Advanced Visual Interfaces*. ACM. 2002, S. 157–166.

[Hu05] Yifan Hu. „Efficient, high-quality force-directed graph drawing". In: *Mathematica Journal* 10.1 (2005), S. 37–71.

[HV09] Danny Holten und Jarke J Van Wijk. „Force-Directed Edge Bundling for Graph Visualization". In: *Computer Graphics Forum*. Bd. 28. 3. Wiley Online Library. 2009, S. 983–990.

[KCH03] Yehuda Koren, Liran Carmel und David Harel. „Drawing huge graphs by algebraic multigrid optimization“. In: *Multiscale Modeling & Simulation* 1.4 (2003), S. 645–673.

[KK89] Tomihisa Kamada und Satoru Kawai. „An algorithm for drawing general undirected graphs“. In: *Information processing letters* 31.1 (1989), S. 7–15.

[Les] Jure Leskovec. *Stanford Network Analysis Project*. URL: snap.stanford.edu (besucht am 27. 10. 2013).

[lib] The igraph library. *iGraph*. URL: igraph.sourceforge.net (besucht am 03. 11. 2013).

[Lum08] Lumina. *Welcome to the third Lumina tutorial*. 2008. URL: web.archive.org/web/20080211204527/http://lumina.sourceforge.net/Tutorials/Noise.html (besucht am 12. 09. 2013).

[LY12] Chun-Cheng Lin und Hsu-Chun Yen. „A new force-directed graph drawing method based on edge–edge repulsion“. In: *Journal of Visual Languages & Computing* 23.1 (2012), S. 29–42.

[Mic] Microsoft. *Microsoft Automatic Graph Layout*. URL: research.microsoft.com/en-us/projects/msagl (besucht am 03. 11. 2013).

[NVI] NVIDIA. *NVIDIA GeForce GTX TITAN*. URL: www.nvidia.de/object/geforce-gtx-titan-de.html (besucht am 16. 10. 2013).

[Opea] OpenGL. *ARB_uniform_buffer_object*. URL: www.opengl.org/registry/specs/ARB/uniform%5C_buffer_object.txt (besucht am 06. 11. 2013).

[Opeb] OpenGL. *Common Mistakes*. URL: www.opengl.org/wiki/Common%5C_Mistakes (besucht am 02. 10. 2013).

[Pre] Prefuse. *Prefuse – Information Visualization Toolkit*. URL: prefuse.org (besucht am 03. 11. 2013).

[Res] AT&T Research. *Graphviz – Graph Visualization Software*. URL: www.graphviz.org (besucht am 03. 11. 2013).

[Shr+13] Dave Shreiner u. a. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3, Eighth Edition*. Addison Wesley, 2013. ISBN: 9780321773036.

[Tea] NodeXL Team. *NodeXL – Network Overview, Discovery and Exploation for Excel*. URL: nodexl.codeplex.com (besucht am 03. 11. 2013).

[Ubi] Inc. Ubiety Lab. *Ubigraph*. URL: ubietylab.net/ubigraph/index.html (besucht am 03. 11. 2013).

[Von+11] Tatiana Von Landesberger u. a. „Visual Analysis of Large Graphs: State-of-the-Art and Future Research Challenges“. In: *Computer graphics forum*. Bd. 30. 6. Wiley Online Library. 2011, S. 1719–1749.

[Wol] Wolfram. *Wolfram Mathematica 9 Documentaation Center – GraphLayout*. URL: reference.wolfram.com/mathematica/ref/GraphLayout.html (besucht am 03. 11. 2013).

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift