Bachelorarbeit

FilterGraph

Entwicklung eines nicht-destruktiven, graphenbasierten Bildeditors

von

Kurt Böhm

Brinkstraße 7A, 38704 Liebenburg

Studiengang: Informatik, 6. Fachsemester

Matrikelnummer: 422376

10. Oktober 2018

eingereicht bei

Prof. Dr. Thorsten Grosch

Institut für Informatik

Abteilung für Graphische Datenverarbeitung und Multimedia

Fakultät für Mathematik/Informatik und Maschinenbau

Technische Universität Clausthal

zusätzlich geprüft durch

Prof. Dr. Jürgen Dix

Institut für Informatik

Abteilung für *Computational Intelligence*

Fakultät für Mathematik/Informatik und Maschinenbau

Technische Universität Clausthal

betreut durch

Johannes Jendersie, M. Sc.

Diese Arbeit beschreibt den Aufbau eines graphenbasierten Bildbearbeitungssystems, dessen Fokus darauf liegt, auch bei beschränkter Arbeitsspeichernutzung Ergebnisse schnell berechnen zu können. Zu diesem Zweck wurde ein Algorithmus entwickelt, der für einen Graphen aus Filtern mit einer beliebigen Bewertungsfunktion entscheidet, welche Zwischenergebnisse in den Arbeitsspeicher geladen werden.

Auf Basis dieses Systems wurde eine *C++*-Implementierung erstellt, die zusätzlich eine Benutzeroberfläche bietet, die auch während Berechnungen responsiv bleibt. Das Bildverarbeitungssystem auf Basis von *libvips* [MC05] [CM96] arbeitet schnell sowie mit geringem Speicher-*Overhead* und kann mit Filtern, die nicht in den Speicher geladen wurden, effizient umgehen.

Mit dieser Implementierung wurde geprüft, wie gut die Laufzeit bei eingeschränktem Speicherverbrauch ist, was herausgestellt hat, dass auch bei Nutzung der Hälfte des Speichers, der für alle Zwischenergebnisse benötigt wird, gute Ergebnisse erzielt werden.

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind, und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Kurt Böhm
Liebenburg, den 10. Oktober 2018

Inhaltsverzeichnis

1 Einleitung **1**

2 Motivierendes Beispiel **1**

3 Anforderungen und Probleme **4**

4 Struktur und Funktionsweise **5**
4.1 Filtergraph 5
4.2 Bilderknoten und Bildergraph 6
4.3 Bilder-*Thread* 6

5 Realisierung **15**
5.1 Der Filtergraph 16
5.2 Bilder 17
5.2.1 Genutzte Operationen 19
5.2.2 Eigenheiten und Probleme 22
5.3 Die Benutzeroberfläche 24

6 Resultate **25**
6.1 Laufzeiten der Filter 25
6.2 Laufzeiten und Speicherverhalten des Programms 27

7 Weiterentwicklungsmöglichkeiten **33**

8 Fazit **34**

9 Literatur **37**

10 Anhang **39**

1 Einleitung

Wie in vielen Bereichen der *Software*welt, steht der potentielle Nutzer eines Bildbearbeitungsprogramms bei der Wahl des Programms, das er nutzen möchte, vor einigen schwierigen Entscheidungen: *Adobe Photoshop* bietet zwar umfangreiche Funktionalität an, doch mit einer alten und stellenweise problematischen Basis, und ist insbesondere kostenpflichtig; bei kostenlosen – oder gar quelloffenen – Alternativen wie *Paint.NET* oder *GIMP* findet man zwar viele derselben Möglichkeiten wieder und gewinnt einiges an Erweiterbarkeit hinzu, doch verliert man einige fortgeschrittene Funktionen.

Darunter besonders hervorzuheben ist die Möglichkeit der *nicht-destruktiven* Bildbearbeitung, also einer, bei der das Eingabebild eines Filters nicht direkt modifiziert wird. In *Photoshop* ist solche Funktionalität auf zwei Arten verfügbar: Einerseits durch *Smartobjekte*, die ein gesamtes Bild in einem Objekt kapseln, auf das sogenannte *Smart-Filter* nicht-destruktiv angewandt werden können, andererseits durch *Einstellungsebenen*, die einfache pixelweise Filter als Ebene darstellen, die das darunterliegende Gesamtbild nicht-destruktiv filtert. Beide Optionen sowie andere nicht-destruktive Programme wie *PhotoFlow*[18d] sind jedoch als Ebenenstapel strukturiert, sodass die beschriebenen Transformationen stets von unten nach oben durchgeführt werden, was eine bedeutende Einschränkung darstellt. Möchte man beispielsweise auf ein Bild mehrere verschiedene Filter ausführen und diese Zwischenergebnisse dann wieder vereinigen, müssen größere Umwege (insbesondere Kopien der Eingabe) eingeschlagen werden, die den Arbeitsfluss stark beeinträchtigen.

In Anbetracht der beschriebenen Unzulänglichkeiten ist es Ziel dieser Arbeit, einen Prototypen eines nicht-destruktiven Bildeditors auf Basis eines Filtergraphen zu entwerfen und zu implementieren, mit Fokus darauf, die Aktualisierungszeit bei Veränderung des Filtergraphen zu minimieren und dabei eine eingeschränkte Menge an Arbeitsspeicher zu verwenden. Außerdem wurde die Entwicklung einer Nutzeroberfläche, die brauchbar ist und bei Berechnungen responsiv bleibt, als Ziel gesetzt.

Damit besteht die Arbeit im Wesentlichen aus drei Abschnitten: Zuerst wird der Aufbau eines Systems erläutert, das die obigen Ziele erfüllt, woraufhin die Realisierung des Systems beschrieben und die Leistung dieses Systems in Hinblick auf die genannten Kriterien bewertet wird.

2 Motivierendes Beispiel

Als erstes möchte ich an einem Beispiel verdeutlichen, dass der hier gewählte, graphenbasierte Ansatz sinnvoll ist, und einige grundlegenden Ideen einführen, wozu Abb. 1 dienen soll. Diese Graphik zeigt den Filtergraphen dieses Beispiels, also den Graphen, mit dem der Nutzer den Bildverarbeitungsprozess steuern kann, wobei die kleinen Kreise auf der linken Seite der Knoten Eingänge und diejenigen

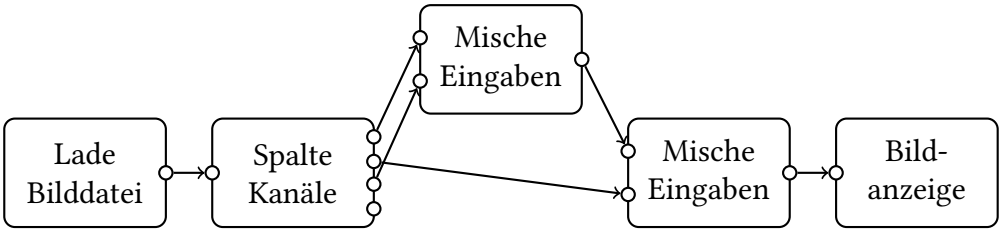

Abbildung 1: Ein erstes Beispiel als Filtergraph.

auf der rechten Seite Ausgänge des Filters darstellen. Jeder dieser Ausgänge stellt ein Bild dar – ein Zwischenergebnis einer Berechnung, die im „Anzeige“-Knoten mündet – und kann somit wiederum als Eingabe für einen weiteren Filter genutzt werden Dabei hat der Anzeigeknoten die besondere Aufgabe, zu ermöglichen, sein Eingabebild in voller Größe zu betrachten – dieses Bild wird im Folgenden „Ergebnis“ oder „Ansicht“ genannt.

In diesem Beispiel werden die Kanäle einer vierkanäligen Eingabe voneinander getrennt und bloß die ersten drei Kanäle miteinander gemischt, um ein Graustufen-Bild mit vom Nutzer vorgegebenen Kanal-Anteilen zu erzeugen. Der Einfachheit wegen wurden diese Anteile an den Mischknoten weggelassen. Schon dieser einfache Vorgang ist in einem stapelförmigen System nur durch mehrfaches Kopieren der Eingabe auf nicht-destruktive Art zu modellieren, woran die größere Flexibilität eines Filtergraphen für die Bildbearbeitung klar wird.

Eine wichtiges Problem, das hier bereits ersichtlich wird, wenn im ersten Knoten ein sehr großes Bild geladen wird, ist die Speicherverwaltung. Liegt hier nämlich eine große Eingabe vor, ist keine der einfachen Lösungen dazu in der Lage, sowohl eingeschränkte Speichernutzung als auch schnelle Berechnungen zu erzielen: Werden alle Zwischenergebnisse in den Speicher geladen, wird viel Speicher eingenommen, wird hingegen nur der Ergebnisknoten permanent in den Speicher geladen, so muss der gesamte Graph bei jeder Veränderung neu berechnet werden. Bei letzterer Option bleibt zusätzlich die Frage offen, auf welche Weise der Ergebnisknoten in den Speicher geladen werden kann, ohne während der Berechnung große Mengen an Speicher für das Laden von Zwischenergebnissen zu verwenden – dieses Problem wird im Realisierungsteil genauer behandelt. Dieser letzte Punkt ist auch der Grund, wieso ein System wie *nip2*[18c] im Kontext dieser Arbeit unzureichend ist, da es ohne Aufforderung von Seiten des Nutzers keine Zwischenergebnisse in den Speicher lädt, was zwar zusätzliche Flexibilität liefert, aber im Normalfall zu längeren Berechnungszeiten führt.

Wählt man beispielsweise ein Bild mit 2^{25} Pixeln (etwa 33.5 Megapixel) und 4 Kanälen im 32Bit-*float*-Format innerhalb des „Lade Bilddatei“-Knotens und lädt alle Zwischenergebnisse in den Speicher, so nimmt der gesamte Graph (eine vierkanälige Eingabe und 7 einkanälige (Zwischen-) Ergebnisse) $(4 \cdot 4 + 7 \cdot 4) \cdot 2^{25} = 1.375 \cdot 2^{30}$Bytes $>$ 1GiB ein, was auf einem *Notebook* mit 4GiB Arbeitsspeicher bereits kritisch sein kann.

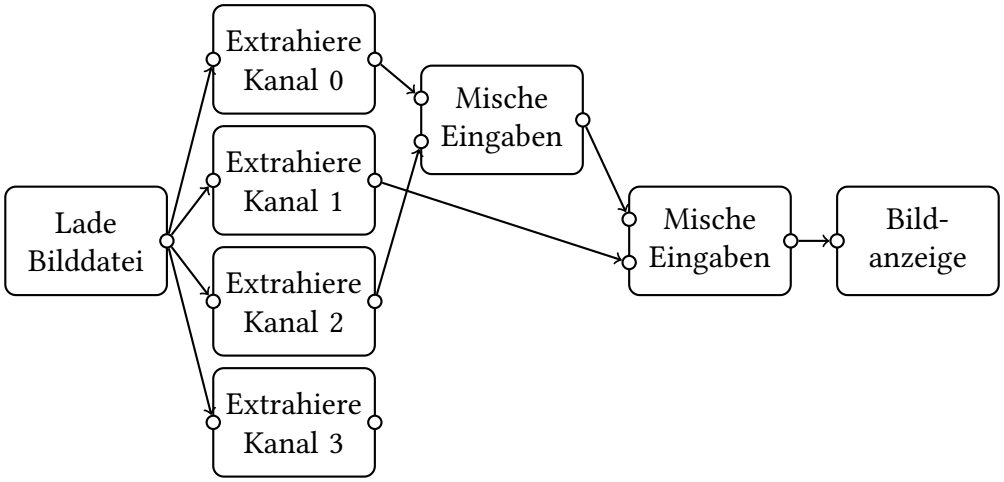

Abbildung 2: Das erste Beispiel als Bildergraph.

In einer solchen Situation bietet es sich an, eine Speicherschranke vorzugeben, die die gespeicherten Bilder nicht überschreiten dürfen, und somit nur bestimmte Zwischenergebnisse in den Speicher zu laden. Dabei stellt sich wiederum die Frage, auf welche Weise die Zwischenergebnisse, die in den Speicher geladen werden, ausgewählt werden sollen, was später genauer diskutiert werden soll. Hierbei ist es jedoch wenig sinnvoll, diese Entscheidung anhand der Knoten des Filtergraphen zu treffen, wie im Beispiel am „Spalte Kanäle“-Knoten ersichtlich ist: Wird hier entschieden, den Knoten in den Speicher zu laden, würde dies alle vier Ausgänge umfassen, da sie verschiedene Bilder produzieren, doch nur drei dieser Ausgänge werden später benötigt. Vielmehr ist es ratsam, diese Entscheidung anhand der Ausgänge der Knoten im Filtergraphen zu treffen, also anhand der Bilder, die in diesem Graphen als (Zwischen-) Ergebnisse auftreten.

Zu diesem Zweck wird der Bildergraph eingeführt, in dem jede Ausgabe eines Filters einen eigenen Bilderknoten (wie sie in der Folge bezeichnet werden) erhält und diese Knoten den Verbindungen der Ursprungsknoten des Filtergraphen entsprechend verknüpft sind. Abb. 2 zeigt den Bildergraphen zu dem in Abb. 1 gezeigten Filtergraphen.

Im Bildergraphen ist es nun sinnvoll, für jeden Knoten separat zu entscheiden, ob er geladen werden soll. Dabei wird sofort ersichtlich, dass kein Bilderknoten von „Extrahiere Kanal 3“ abhängt, weshalb er bei Speichermangel nicht in den Speicher geladen werden sollte. Dies ist ein Bewertungskriterium, dass später wieder aufgegriffen wird. Ist genug Speicher da, könnte er dennoch geladen werden, da eine zukünftige Verbindung davon profitieren würde.

3 Anforderungen und Probleme

Aus den bereits beschriebenen Zielen des Projektes (schnelle Berechnung, eingeschränkter Speicher, responsive Nutzeroberfläche) lassen sich einige Anforderungen an das in dieser Arbeit dokumentierte System herleiten, für die klar werden soll, wie sie in diesem System realisiert werden. Damit die Anforderungen klar referenziert werden können, werden sie hier nummeriert aufgeführt:

1. Es muss möglich sein, einzelne Bilderknoten nicht in den Arbeitsspeicher zu laden bzw. aus diesem zu entfernen, aber auch, diese zu generieren, um abhängige Bilderknoten oder (zu einem späteren Zeitpunkt) den Bilderknoten selbst berechnen zu können – dass dies nötig ist, wenn der Graph zu viel Arbeitsspeicher in Anspruch nehmen möchte, wurde bereits diskutiert.

2. Die Bilderknoten, die nicht in den Speicher geladen werden, müssen so ausgewählt werden, dass für die Berechnung der Ergebnisse des Graphen möglichst wenig Zeit verloren geht. Diese Entscheidung muss dabei auch zukünftige Neuberechnungen in Betracht ziehen – so wäre es innerhalb einer Berechnung selbst vielleicht am schnellsten, bloß die Ergebnisse in den Speicher zu laden, doch für jede darauf folgende Berechnung müsste wieder der gesamte Graph neu berechnet werden, was insgesamt deutlich langsamer wäre.

3. Damit der Nutzer während der Berechnung von Bildern nicht warten muss, bevor er den Graphen weiter verändern oder Ansichten bereits berechneter Knoten anschauen kann, müssen die Berechnungen asynchron durchgeführt werden.

4. Damit der Nutzer sich die Ergebnisse seines Graphen anschauen kann, muss es möglich sein, einen Ansichtenknoten in voller Größe betrachten zu können.

5. Damit bei mehreren aufeinander folgenden Teilveränderungen nur eine Berechnung durchgeführt wird, ist es sinnvoll, die Aktualisierung des Graphen pausieren und später wieder fortsetzten zu können. Dies erlaubt es, die Zeit, die durch unnötige Berechnungen bei nur teilweise vollzogenen Veränderungen verloren geht, zu verringern.

Für die Umsetzung dieser Anforderung eröffnen sich jedoch folgende maßgebliche Probleme, die für Rückbezüge nummeriert werden:

1. Wie wird entschieden, welche Bilderknoten in den Arbeitsspeicher geladen werden?

2. Wie wird während der asynchronen Berechnung eines Bilderknoten sichergestellt, dass Veränderungen am Filtergraphen den Zustand der Berechnung nicht stören?

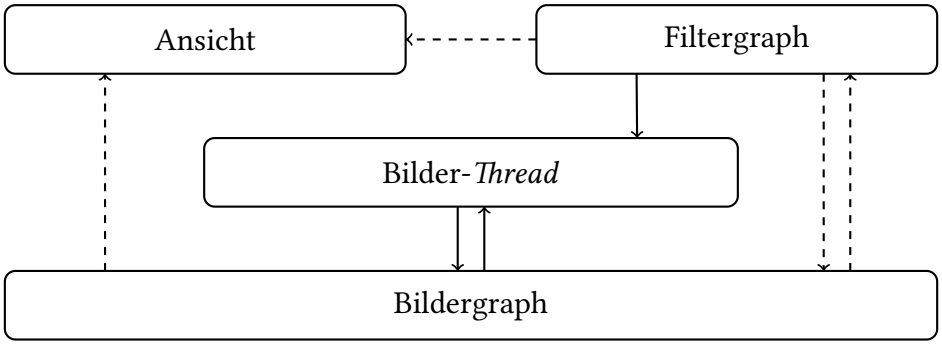

Abbildung 3: Die grundlegende Struktur des Systems.

3. Wie können Bilderknoten dargestellt und verarbeitet werden, um jederzeit berechnet werden zu können und dabei möglichst wenig Arbeitsspeicher einzunehmen?

4 Struktur und Funktionsweise

Der in Abb. 3 visualisierte Aufbau des Programms dient der Lösung der zuvor geschilderten Anforderungen und Probleme. Dabei teilt der Filtergraph dem Bilder-*Thread* seine Veränderungen mit, der mit ihnen drei Schritte durchführt: Er wendet sie asynchron am Bildergraphen an, er entscheidet, welche Bilderknoten in den Speicher geladen bzw. aus diesem entfernt werden, und führt diese Entscheidungen dann durch. Die gestrichelten Linien stellen dabei Vorgänge dar, die zur Anzeige von Ergebnissen dienen.

4.1 Filtergraph

Der Filtergraph stellt den Bildbearbeitungsprozess, wie bereits in den einleitenden Kapiteln motiviert und am Beispiel gezeigt, als Graphen aus Filtern dar, zu denen hier auch Bildquellen und -senken gezählt werden. Er ist dabei die Hauptkomponente, die dem Nutzer offen liegt und die dieser modifizieren kann, indem er Knoten oder ihre Verbindungen hinzufügt oder löscht bzw. die Parameter dieser Knoten verändert.

Unter den Filterknoten nimmt der bereits thematisierte Anzeigeknoten als Senke eine besondere Rolle ein. Er zeigt eine verkleinerte Vorschau seines Eingabebildes an, die entlang des Pfeils vom Bildergraphen aus übermittelt wurde, und steuert die übrigen gestrichelten Pfeile aus Abb. 3, um dem Nutzer eine Ansicht in voller Größe anzubieten (Anforderung 4). Wird ein Anzeigeknoten ausgewählt, benachrichtigt er die Ansicht, dass sie nun das Ergebnis dieses Knotens anzeigen soll, und fordert beim Filtergraphen an, dass dieser so schnell wie möglich das entsprechende Bild an die Ansicht übermittelt. Wie dies im Fall von fortschreitenden

Veränderungen auf sinnvolle Weise gelöst wird, findet sich in folgenden Abschnitt.

4.2 Bilderknoten und Bildergraph

Der Begriff „Bilderknoten“ wurde bereits eingeführt, da er ein Konzept darstellt, das vom üblichen Verständnis des Wortes „Bild“ abweicht, denn ein Bilderknoten repräsentiert nicht, wie für Bilder üblich, die direkte Repräsentation einer Pixelmatrix im Speicher (mit einigen zusätzlichen Informationen), sondern vielmehr einen Knoten im Bildergraphen. Wie dieser Bildergraph zustande kommt, wurde bereits in einleitenden Beispiel angerissen: Jeder Ausgang im Filtergraphen (sowie der Anzeigeknoten, da er eine Ansicht generieren kann) ist ein Knoten im Bildergraphen, und zwei Knoten sind genau dann verbunden, wenn die Ursprungsknoten im Filtergraphen verbunden waren. Wie bereits diskutiert, kann es dabei sinnvoll oder erforderlich sein, nur bestimmte Bilder im Speicher zu halten, sodass jeder dieser Knoten sowohl in den Speicher geladen als auch wieder aus diesem entfernt werden kann, was genau Anforderung 1 entspricht. Um dies zu unterstützen, enthält ein Bilderknoten folgende Informationen:

ρ**:** Eine Funktion, die aus den Eingabeknoten die Ausgabe bestimmen kann.

τ**:** Eine Funktion, die näherungsweise bestimmt, wie lange die Berechnung der Ausgabe aus den Eingaben dauert, falls alle Eingaben eine Speicherrepräsentation besitzen – andernfalls müssen die Eingaben ebenfalls berechnet werden, was über τ hinaus Zeit einnimmt.

Speicherrepräsentation: Ein Bild im traditionellen Sinne (also inklusive Pixel-Matrix), das nur dann vorliegt, wenn der Bilderknoten in den Speicher geladen wurde, und in diesem Fall durch ρ berechnet wurde.

Innerhalb der Arbeit soll die Unterscheidung zwischen Bilderknoten, die eine Speicherrepräsentation besitzen, und denen ohne diese auch in der Sprache widergespiegelt werden: Der erste Fall wird durch den Begriff „Speicherbild“, der zweite durch „Filterbild“ bezeichnet werden. Außerdem werden „einen Bilderknoten in den Speicher laden“ bzw. „einen Bilderknoten aus dem Speicher entfernen“ als Ausdrücke genutzt, um das Erstellen bzw. Entfernen der Speicher-Repräsentation zu beschreiben, und der Terminus „Speicherzustand“ beschreibt, ob der Bilderknoten eine Speicherrepräsentation besitzt oder nicht.

4.3 Bilder-*Thread*

Der Bilder-*Thread* ist der Teil des Systems, der die schwierigsten Aufgaben übernimmt und dabei zwei Anforderungen des Systems erfüllt und ein Problem löst.

Solange das System läuft, durchläuft der Bilder-*Thread* asynchron von der Benutzeroberfläche (gemäß Anforderung 3) die in Abb. 4 dargestellte Schleife, wobei

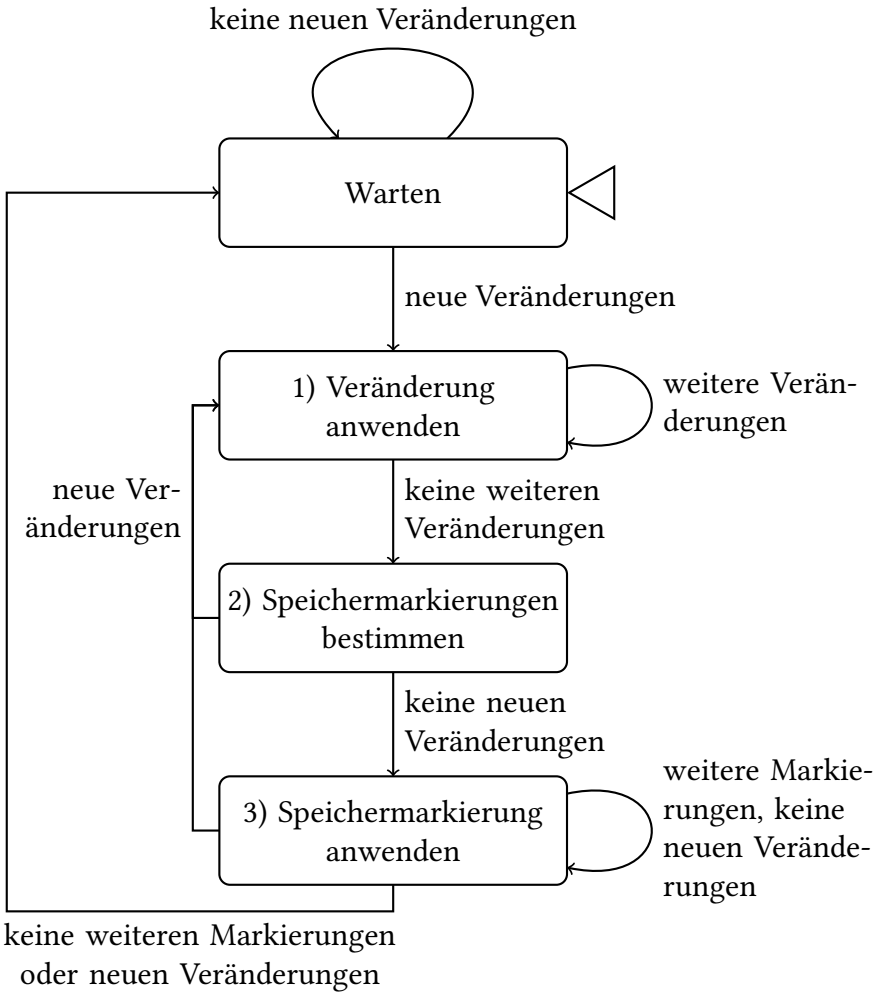

Abbildung 4: Ein vereinfachter Zustandsautomat des Bilder-*Thread*.

der Übersicht wegen in der Graphik der Endzustand nicht aufgenommen wurde – aus jedem der dargestellten Zustände besteht ein Übergang, der bei Ende des Programms bedingungslos zum Endzustand wechselt. Außerdem wird hier nicht beachtet (in der Realisierung wird dies beachtet), dass es gemäß Anforderung 5 möglich sein sollte, die Aktualisierung zu pausieren – in diesem Fall wird der Wartezustand nicht verlassen oder, falls dies im „Speicheraktion anwenden"-Zustand geschieht, in den Wartezustand gewechselt. Wird die Pausierung aufgehoben, wird in den „Veränderung anwenden"-Zustand gewechselt.

Treten Veränderungen am Filtergraphen auf, so werden diese zeitlich sortiert und auf *Thread*-sichere Weise im Bilder-*Thread* gespeichert und der *Thread* geht dazu über, einen Aktualisierungsdurchlauf auszuführen. Zunächst werden dabei die Veränderungen in Teil 1 des Automaten auf den Bildergraphen übertragen (es werden also entsprechend Bilder oder Verbindungen entfernt bzw. gelöscht oder die Bilderknoten verändert), wobei keiner der veränderten Bilderknoten in den Speicher geladen wird. Dies ist ein zusätzlicher Vorteil des Bilder- vom Filtergraphen in Hinblick auf Problem 2, denn auf diese Weise lassen sich die Veränderungen am Filtergraphen zeitlich von denen am Bildergraphen entkoppeln, sodass sich letzterer Graph während des Ladens von Bilderknoten nicht ändert und zu keinen inkonsistenten Ergebnissen führen kann.

Wurde der Bildergraph aktualisiert, so wird in Teil 2 des Automaten entschieden, welche Bilderknoten eine Speicherrepräsentation haben sollen. Dieser Entscheidungsprozess wurde konstruiert, um Anforderung 3 zu erfüllen, also die Bilderknoten, die nicht in den Speicher geladen werden, so zu wählen, dass für die Berechnung der Ergebnisse möglichst wenig Zeit verloren geht. Dazu wird eine Bewertungsfunktion φ genutzt, die angibt, wie wichtig es ist, ein Bild im Speicher zu halten, und für ihre Berechnungen keine Einschränkungen von Seiten des Algorithmus auferlegt bekommt. Es gibt jedoch einige sinnvolle Komponenten, die für die Bewertung eines Knoten in Betracht gezogen werden können:

Berechnungszeit: Dies bezeichnet die Gesamtzeit T, die benötigt wird, um das Bild zu berechnen, inklusive der Berechnungszeiten der Eingaben. Dies ergibt sich als Summe aus der Ausführungszeit τ des Knoten selbst sowie der Summe der τ aller Eingänge, die keine Speicherrepräsentation besitzen, was sich wie folgt für ein Bild i formulieren lässt (wobei $N^-(i)$ die Eingabebilder des Bildes i angibt und $\mu(i)$ genau dann 0 ist, wenn i im Speicher ist, und sonst 1):

$$T(i) \coloneqq \tau(i) + \sum_{j \in N^-(i)} \mu(j)\, T(j)$$

Diese Komponente bewertet also Bilder, die länger für ihre Berechnung brauchen, höher als die, die weniger lange benötigen, und ist damit insbesondere darauf fokussiert, die Berechnungszeit zukünftiger Aktualisierungen zu verringern, da diese Speicherbilder, die nicht durch die Veränderungen am Filtergraphen betroffen

sind, nicht noch einmal berechnen müssen, was für Bilder mit langer Berechnungszeit besonders vorteilhaft ist. Dabei verändert das Laden eines Bilderknotens i den Wert T der abhängigen Knoten (bis zum jeweils nächsten Speicherbild hin), da sie i und seine Filter-Vorgänger nicht mehr berechnen müssen, und damit ihre Relevanz.

Die Berechnungszeit ist ein äußerst sinnvolles Kriterium und tritt somit in allen Bewertungsfunktionen, die im Folgenden diskutiert werden, auf.

Anzahl an Ergebnispfaden: Dies bestimmt für einen Bilderknoten i die Anzahl p^+ an i-j-Pfaden, bei denen j ein Ansichtenknoten ist, was sich wie folgt berechnen lässt (wobei $N^+(i)$ die Menge aller Nachfolger des Bildes i darstellt):

$$p^+(i) \coloneqq \begin{cases} 1 & i \text{ ist ein Ansichtenknoten} \\ \sum_{j \in N^+(i)} p^+(j) & \text{sonst} \end{cases}$$

Dies ist also die Zahl an verschiedenen Berechnungsflüssen, für die i benötigt wird und die in Ergebnissen enden. Für diese Berechnungsflüsse muss i als Zwischenergebnis berechnet werden, falls es nicht in den Speicher geladen wird. Der Wert eignet sich gut als Faktor eines Produkts mit der Berechnungszeit, um die Gesamtzeit zu bestimmen, die damit verbracht wird, das Bild als Zwischenergebnis zu berechnen, falls es nicht im Speicher ist. Dabei trägt er insbesondere dazu bei, die Bilderknoten höher zu bewerten, die in der gegenwärtigen Aktualisierung mehrfach berechnet und damit unnötig Zeit verwenden würden.

Benötigter Speicher: Wird mit μ_B bezeichnet und gibt an, wie viel Speicher der Bilderknoten einnehmen würde, falls er im Speicher wäre bzw. wie viel er einnimmt, falls es im Speicher ist.

Dieser Wert kann sinnvoll als Divisor in einem Quotienten mit einer Kombination der vorigen Werte genutzt werden, um zu berechnen, wie groß der Wert pro Speichereinheit ist.

Damit lässt sich der Algorithmus zur Bestimmung der Speicherbilder beschreiben. Dieser startet mit der Menge aller Filterbilder als Menge von Kandidaten, die zum Laden ausgewählt werden können, und markiert alle Bilder mit dem Zustand, in dem sie gegenwärtig sind. Diese Markierung kann innerhalb des Algorithmus' verändert werden und gibt nach dessen Ende an, in welchen Speicherzustand der entsprechende Bilderknoten überführt wird, weshalb er innerhalb des Algorithmus' für die Unterscheidung zwischen Speicher- und Filterbildern genutzt wird. Angefangen wird also mit Markierungen, die den Speicherzustand der Bilderknoten nicht verändern.

Aus den Kandidaten wird nun der wertvollste Bilderknoten $f_{\max}$ gemäß φ bestimmt, und falls ein solcher Knoten existiert, wird geprüft, ob der verfügbare Arbeitsspeicher ausreicht, um $f_{\max}$ in den Speicher zu laden. Falls nicht, wird geprüft,

ob es ein Speicherbild $m_{\min}$ mit minimalem $\varphi(m_{\min})$ gibt, das gemäß φ weniger wertvoll ist, als $f_{\max}$, und falls dies der Fall ist, wird $m_{\min}$ als Filterbild markiert und zu den Kandidaten hinzugefügt, woraufhin $f_{\max}$ neu bestimmt werden muss, da T von den Speicherzuständen der eingehenden Knoten abhängt, der sich hier verändert haben kann. Dies wird wiederholt, bis der Arbeitsspeicher für das Laden von $f_{\max}$ ausreicht oder es keine als Speicherbilder markierten Bilderknoten mehr gibt, woraufhin $f_{\max}$ aus der Menge der Kandidaten entfernt wird.

Falls der Speicher ausreicht, wird $f_{\max}$ in den Speicher geladen, falls nicht, war es nicht wertvoll genug, um andere Speicherbilder zu ersetzen, und hat somit es seine Chance, geladen zu werden, vertan und wird in Zukunft nicht mehr beachtet. Dies verhindert, dass diese Schleife mehrfach für einen Bilderknoten durchlaufen wird, der sich bereits als zu wenig wertvoll herausgestellt hat, und dient somit auch als gute Abbruchbedingung: Die Schleife wird fortgeführt, bis keine Kandidaten mehr vorliegen, also keine Bilderknoten mehr vorliegen, die wertvoll und klein genug sind, um geladen zu werden. In diesem Fall werden die Markierungen topologisch sortiert und der Algorithmus endet. Dieses Vorgehen wird in Abb. 5 visualisiert.

Die Funktionsweise dieses Algorithmus wird nun an einem Beispiel mit der folgenden Bewertungsfunktion durchgegangen, wobei i einen Bilderknoten darstellt:

$$\varphi(i) \coloneqq T(i) \cdot p^{+}(i)$$

Hier wird, wie zuvor diskutiert, die Berechnungszeit mit der Zahl an Ergebnispfaden multipliziert, um die Gesamtzeit zu bestimmen, die zur Berechnung des Bildes verwendet werden muss, wenn das Bild nicht in den Speicher geladen wird. Dabei wird zusätzlich festgelegt, dass Anzeigeknoten nie geladen werden, da sie bloß eine Kopie ihrer Eingabe darstellen und es daher sinnvoller ist, diese Eingabe zu laden.

Dieses φ soll jetzt genutzt werden, um die Bilderknoten aus dem bereits zuvor eingeführten Bildergraphen Abb. 6 auszuwählen, die mit einer Speicherschranke von 50 in den Speicher geladen werden sollen. Wie zu sehen ist, ergibt sich T stets als Summe aus τ und den Ts der Filter-Eingangsknoten, und φ ist (gemäß Definition) $T \cdot p^{+}$.

Zu Beginn sind noch keine Bilderknoten in den Speicher geladen worden, also sind alle Knoten Kandidaten und der zweite Mischknoten ist der Kandidat mit dem größten Wert von φ. Da es keine Speicherbilder gibt, wird die erste Schleife nicht betreten, und da die 50 restlichen Speichereinheiten für die 10, die das Bild einnimmt, ausreichen, wird es zum Laden markiert und 40 Speichereinheiten stehen noch zur Verfügung.

Da der einzige Knoten, der vom markierten Knoten abhängt, der Ansichtknoten ist, der ohnehin eine Bewertung von 0 hat und nie geladen wird, verändern sich die relevanten Werte auch nur an dieser einen Stelle. Das Ergebnis dieser Veränderungen ist in Abb. 7 zu sehen. Der wertvollste unter den restlichen Knoten ist der Ladeknoten, und da der restliche Speicher von 40 genau ausreicht, um ihn zu

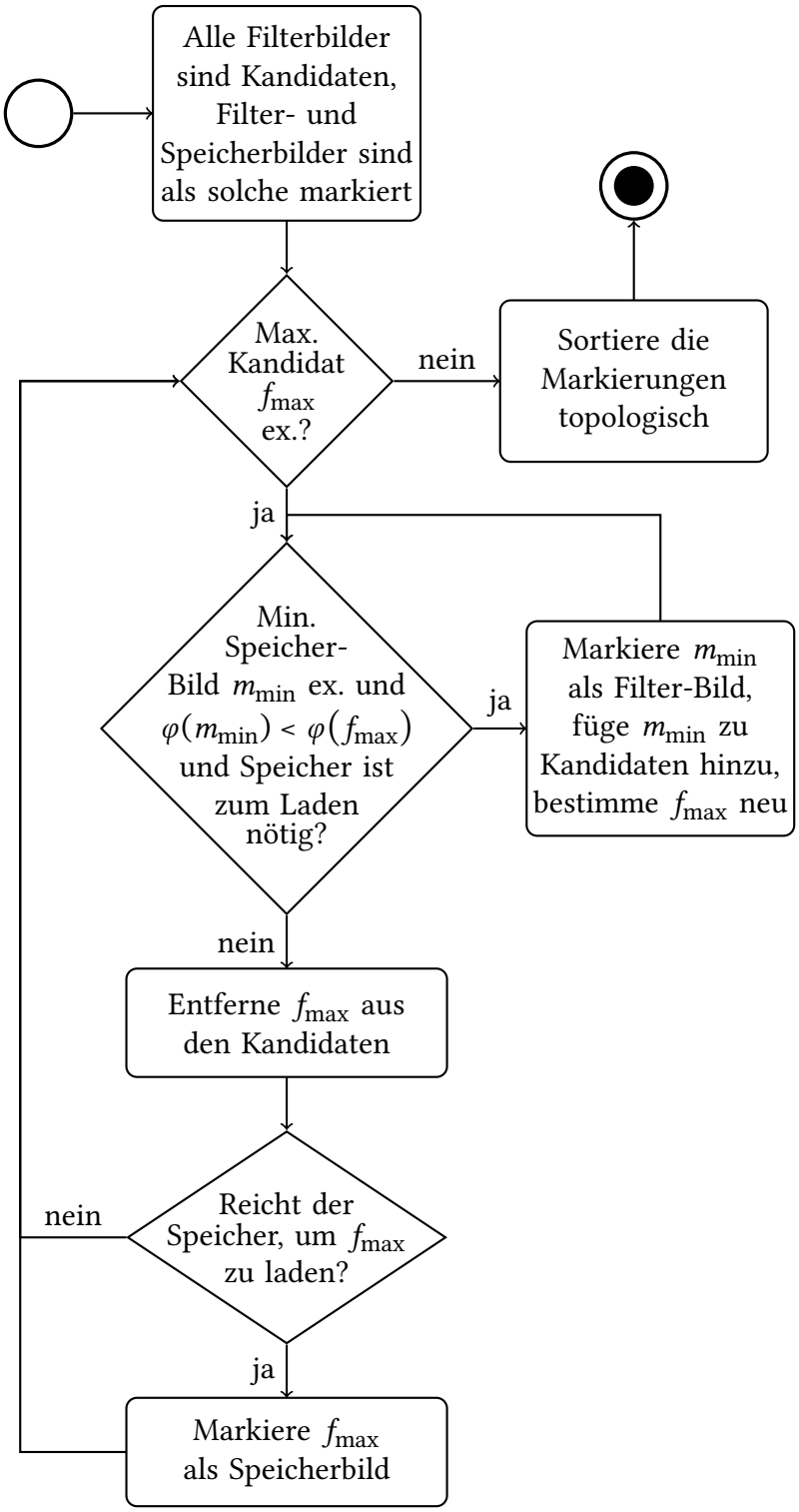

Abbildung 5: Ein vereinfachtes Flussdiagramm der Auswahl von Speicherbildern.

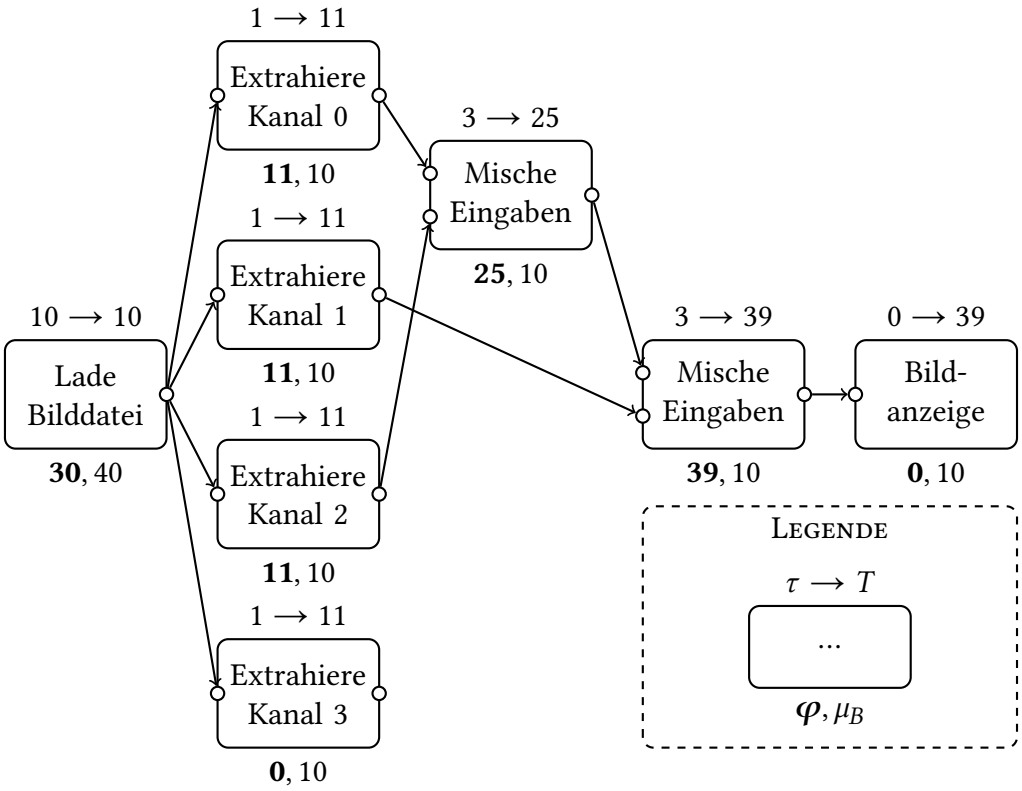

Abbildung 6: Ein Beispiel vor Beginn der Speicherentscheidungen. Es soll entscheiden werden, welche der Knoten bei einer Speicherschranke von 50 geladen werden sollen.

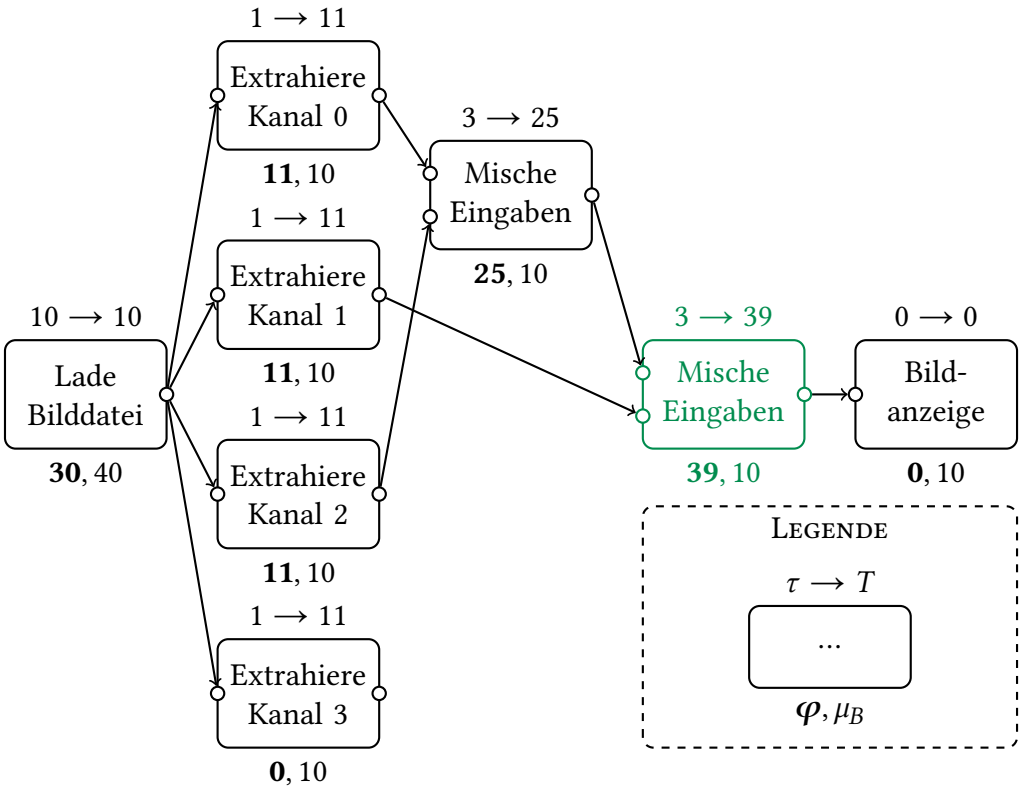

Abbildung 7: Das Beispiel während der Speicherentscheidungen. „Mische Eingaben“ wurde als wichtigster Bilderknoten bewertet und zum Laden markiert.

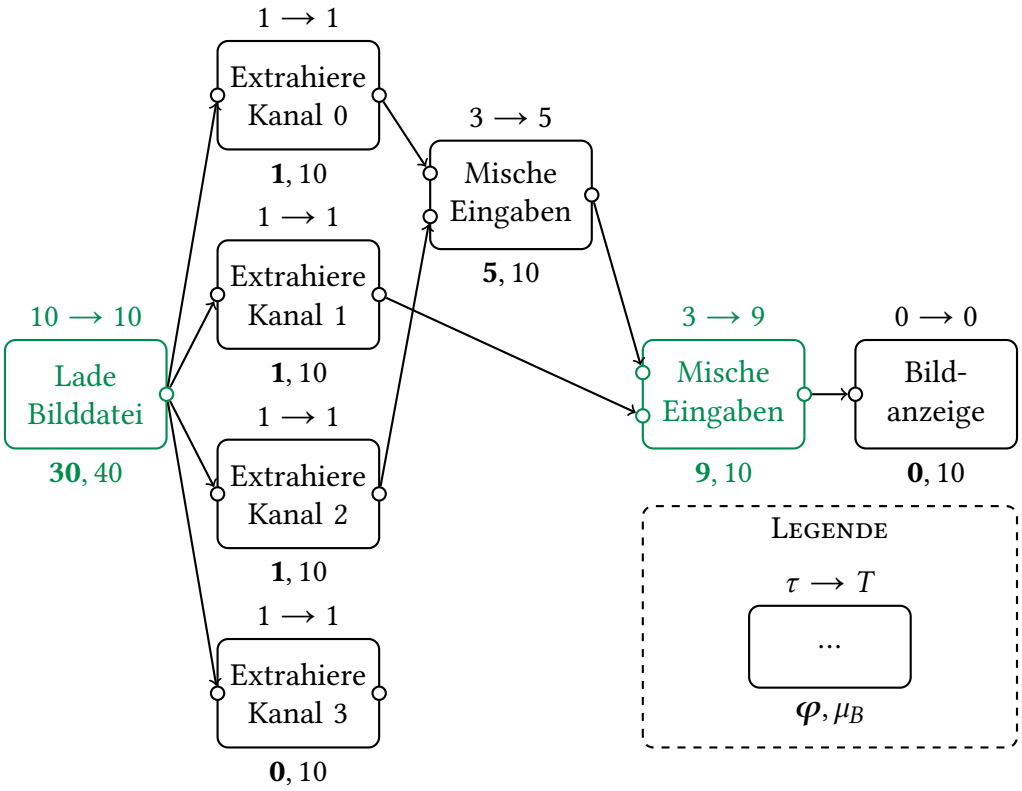

Abbildung 8: Das Beispiel nach den Speicherentscheidungen. Zusätzlich zu „Mische Eingaben“ wurde „Lade Bilddatei“ zum Laden markiert und der Speicher ist voll.

laden, wird die erste Schleife nicht betreten und der Knoten zum Laden markiert, womit 0 Speicherinheiten übrig bleiben.

In Abb. 8 sind die beiden bereits angesprochenen Knoten zum Laden markiert und keiner der übrigen Knoten ist wertvoller, also wird keine der Markierungen ersetzt und der Algorithmus terminiert, nachdem er fruchtlos über den Rest der Knoten iterierte.

Die beiden Knoten, die gewählt wurden, sind eine gute Wahl, da der Anzeigeknoten seine Ansicht sofort bereit hat und in Zukunft der lange Ladevorgang nicht mehr ausgeführt werden muss. Hieran kann schon erahnt werden, dass das für dieses Beispiel definierte φ eine gute Bewertungsfunktions darstellt; Weiteres dazu folgt in Abschnitt 6.1.

Wurde auf die beschriebene Weise bestimmt, welche Bilder in den Speicher geladen bzw. aus diesem entfernt werden sollen, werden diese Markierungen danach gefiltert, ob der zur Markierung gehörige Knoten seinen Speicherzustand verändern soll oder durch eine der Veränderungen am Filtergraphen, durch die die Aktualisierung initiiert wurde, betroffen ist. Dies dient sowohl dazu, nur die Markierungen, die etwas verändern, auch wirklich anzuwenden, als auch dazu, den

Fortschritt der Berechnung sinnvoll messen (und dem Nutzer anzeigen) zu können.

Diese gefilterten Markierungen werden dann in Teil 3 des Zustandsautomaten (Abb. 4) angewandt, verändern, falls nötig, den Speicherzustand und benachrichtigen die abhängigen Knoten bzw. die Ansicht, falls die initiierende Veränderung am Graphen sie betrifft.

5 Realisierung

Bevor die eigentliche Realisierung beschrieben wird, bietet es sich an, etwas über die Randbedingungen der Entwicklung zu sagen.

Der Entwicklungsprozess war stark iterativ gestaltet, mit dem Ziel, circa alle zwei Wochen einen lauffähigen Prototypen mit neuer Funktionalität oder anderen Veränderungen vorweisen zu können. Dies war hilfreich, um Probleme mit den genutzten Bibliotheken und Ansätzen aufzudecken und zu beheben, hatte aber gleichzeitig den Nachteil, dass es zu einer größeren Menge an verworfenen Ideen und *Code*-Fragmenten geführt hat. Aufgrund dieser Tatsache wird in diesem Abschnitt neben der Realisierung auch die Entwicklung derselben Erwähnung finden, um die Probleme des vorherigen bzw. die Vorteile des späteren Ansatzes illustrieren zu können.

Zu Beginn dieses Prozesses stellte sich die Frage, welche Programmiersprache mit welchen Werkzeugen und Bibliotheken genutzt werden sollte. Diese Entscheidung war insofern unangenehm, dass ich bis zu dieser Arbeit hauptsächlich mit *Java* bzw. *Kotlin* sowie *Python* als Programmiersprachen gearbeitet habe, aber die erhöhte Leistungsfähigkeit und größere Kontrolle über die Speichernutzung von *C++* als für das Ziel der Arbeit relevante Eigenschaften anerkannte. Aus diesen Gründen fiel die Wahl auf *C++*, was eine größere Lernkomponente in die Entwicklung dieser Arbeit einführte und, durch fehlende Erfahrung mit der Sprache und den zur Verfügung stehenden Bibliotheken, den Trend des iterativen Arbeitens verstärkte.

Mit dieser Entscheidung kam auch die Wahl eines *C++-Compilers*. Diese fiel hauptsächlich deswegen auf den *Microsoft Visual C++-Compiler*, da sich eine relativ einfache Einrichtung erhofft wurde – dies war, in Zusammenhang mit der guten Integration mit *Qt*, eine brauchbare Umgebung für die frühe Entwicklung des Programms. Diese Entscheidung erwies sich jedoch als problematisch, als ein Wechsel zu *libvips* als Graphik-Bibliothek geplant wurde (siehe Abschnitt 5.2), denn *libvips* bietet keine für *Microsoft Visual Studio* vorkompilierte Version an, und Informationen zum eigenen Kompilieren sind widersprüchlich und wenig hilfreich. Dies führte zu einem Wechsel zu einer *MinGW-w64*-basierten *Pipeline* mit *CMake* als *Build*-System, und nach einer aufwändigen Einrichtung funktionierte das Programm auch unter den veränderten Bedingungen wie gewünscht.

Die letzte in diesem Kontext nennenswerte Veränderung war der Wechsel von

Microsoft Windows 10 zu einem *Linux*-basierten Betriebssystem und somit zu *GCC* als *Compiler*, was sich durch *Package Manager* und das einfachere Kompilieren von Bibliotheken als vorteilhaft erwies. Außerdem zeigt dies auf, dass das Programm auf zwei der großen *Desktop*-Betriebssysteme problemlos funktioniert und somit auch eine angemessene Portabilität bietet.

Im Folgenden wird die Realisierung der einzelnen Komponenten des Programms diskutiert. Bildergraph und -*Thread* erhalten dabei keinen Abschnitt, da sie bereits hinreichend genau beschrieben wurden und hier nur noch unnötige Implementierungsdetails diskutiert werden könnten.

5.1 Der Filtergraph

Der Filtergraph erfordert, als vom Nutzer veränderbarer Anteil des Programms, sowohl einer graphischen Nutzerschnittstelle, die den Graphen anzeigt und Veränderungen an ihm zulässt, als auch eines internen Systems, das den Zustand des Graphen speichert und Veränderungen verarbeitet. Zu diesem Zweck wurde *NodeEditor* [Pin+18] als Bibliothek ausgewählt, da die Programmierschnittstelle einen adäquaten Funktionsumfang und einige nützliche Beispiele bietet. Außerdem ist die graphische Nutzerschnittstelle flexibel genug, um alle erwünschten Filter darstellen zu können.

Innerhalb des *NodeEditor* gibt jeder Knoten an, wie viele Ein- und Ausgänge er besitzt, welche Benutzeroberfläche er bietet (um Parameter bearbeiten zu lassen), auf welche Weise Eingabedaten verarbeitet und welche Daten durch welchen Ausgang ausgegeben werden. Diese Knoten benachrichtigen den *NodeEditor*, wenn sich ihre Ausgabe für einen Ausgang verändert hat (bspw. durch Nutzereingaben), der diese neue Ausgabe an die mit dem Ausgang verbundenen Knoten weiterleitet. Diese führen ihre Datenverarbeitung durch und entscheiden, ob sich ein Ausgang verändert hat, was wiederum zur Benachrichtigung des *NodeEditor* führt. Dasselbe System wird bei der Erstellung oder Entfernung einer Verbindung zwischen Knoten genutzt: In diesem Fall wird dem Eingang, in den die Verbindung mündet bzw. mündete, die Ausgabe des neu verknüpften Ausgangs bzw. `nullptr` übergeben.

Dieser letzte Punkt weist bereits auf, dass ein Knoten neue Daten über eine neue oder bestehende Verbindungen erhalten kann, und da ein Ausgang auch absichtlich `nullptr` als Wert haben kann, überlappt sich dies zusätzlich mit der Entfernung eines Knoten – diese Fälle kann ein Knoten nicht klar voneinander unterscheiden. Dies ist ausreichend, wenn die Berechnung neuer Daten direkt in den Knoten stattfinden kann, aber problematisch, wenn das Vorgehen von der Struktur des Graphen abhängt oder ein Knoten Informationen über seine Nachfolger benötigt, wie in diesem System für die Speicher-Entscheidungen bzw. der Bewertungsfunktion der Fall ist.

Aus diesen Gründen wurde die Entscheidung getroffen, die Datenverarbeitung nicht innerhalb von *NodeEditor* durchzuführen und anstelle dessen das zuvor beschriebene System zu erstellen, bei dem die Veränderungen am Filtergraphen di-

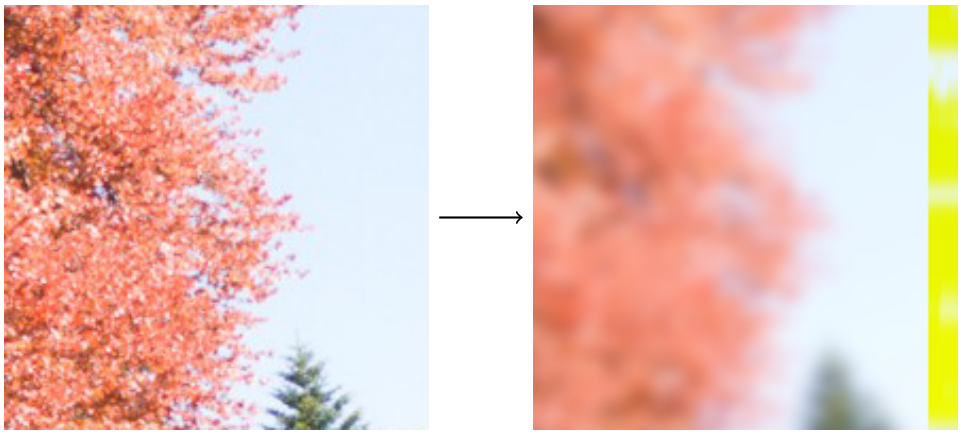

Abbildung 9: Artefakte von *OpenCV* 3.4.1 bei Anwendung des `GaussianBlur`-Filters mit Radius 4 und sonst den Standardeinstellungen.

rekt an den Bilder-*Thread* weitergegeben werden. Um dabei das Problem der Mehrdeutigkeit von Eingabedaten zu lösen, habe ich *NodeEditor* modifiziert und dadurch insbesondere ermöglicht, dass Knoten die Erstellung einer neuen Verbindung, die Entfernung einer bestehenden Verbindung und die Veränderung der Eingabedaten durch eine bestehende Verbindung separat behandeln können.

Für die Nutzbarkeit des Systems ist außerdem von Interesse, dass *NodeEditor* die Möglichkeit bietet, den gegenwärtigen Filtergraphen in einer *JSON*-Datei abzuspeichern und eine solche Datei zu laden. Dazu ist jeder Knoten in der Lage, im Rahmen des Speichervorgangs seine Parameter im *JSON*-Format aufzubereiten sowie diese wieder einzulesen und auf sich anzuwenden, wenn der Graph geladen wird.

Auf Basis dieses Systems wurden 10 Knotentypen definiert, die der Nutzer innerhalb des Filtergraphen nutzen kann und die jeweils einer der Operationen aus dem folgenden Abschnitt 5.2 entsprechen. Diese Knotentypen enthalten alle Parameter, die die entsprechende Operation anbietet, als Element der Benutzeroberfläche. Dabei werden Zahlen über Textfelder modifiziert, die invalide Eingaben erkennen und im Fehlerfall rot markiert werden, die Auswahl aus einer endlichen Anzahl an Optionen (bspw. Skalierungsarten) wird wiederum als *Combo*-Box aufbereitet. Um Wiederholungen mit dem folgenden Abschnitt zu vermeiden, werden alle weiteren Details diesem überlassen.

5.2 Bilder

Zu Beginn der Projektes wurde *OpenCV* [Bra00] als Grafikbibliothek ausgewählt, da ich bereits Erfahrungen mit ihr gemacht hatte, sie eine großzügige Auswahl an vorgefertigten Filtern bietet und recht schnell arbeitet.

OpenCV weist in der Berechnung einiger Bilder jedoch visuelle Artefakte auf,

die nur schwer zu rechtfertigen ist; besonders deutlich ist dies bei der Anwendung einer Gaußschen Weichzeichnung auf Bilder, auf denen der Himmel zu sehen war – in diesem Fall wurden Teile des weichgezeichnet Himmels gelb (wahrscheinlich durch einen Überlauf im Blau-Kanal), wie in Abb. 9 zu sehen ist.

Andererseits eignet sich *OpenCV*s Vorgehensweise bei der Bildverarbeitung äußerst schlecht für die Nutzung in dem hier vorgestellten System, denn in *OpenCV* werden alle Zwischenergebnisse von Operationen vollständig in den Speicher geladen, was innerhalb des Bildergraphen problematisch ist, wenn ein Bild, das in den Speicher geladen werden soll, mehrere Eingabebilder hat, die nicht wertvoll genug waren, um in den Speicher geladen zu werden. Diese Bilder müssen für die Berechnung des Zwischenergebnisses dennoch vollständig in den Speicher geladen werden, was die Speicherschranke bereits überschreiten kann und den Nutzen der Entscheidung, welche Bilder in den Speicher kommen, stark schmälert.

Als Alternative Bibliothek wurde *libvips* [MC05][CM96] gewählt, da es besonders auf Laufzeit- und Speichereffizienz ausgelegt ist. Dies wird u. a. dadurch erreicht, dass *libvips*' Bilder im Normalfall bloß eine Sammlung von Filtern (inklusive ihrer Parameter) darstellen, ohne eine direkte Speicherrepräsentation aufzuweisen – diese wird erst berechnet, wenn das Bild (oder Ausschnitte desselben) in den Arbeitsspeicher geladen werden soll. Dieser Berechnungsprozess wird stark parallelisiert in kleinen Regionen durchgeführt, auf die die entsprechenden Filter nacheinander angewandt werden und die am Ende zum Ergebnis vereinigt werden.

Dieses System ist im Kontext des Bildergraphen besonders vorteilhaft, da es – im Gegensatz zu *OpenCV*s Vorgehen – jeweils nur kleine Ausschnitte jedes Eingabebildes im Speicher halten muss, um den entsprechenden Ausschnitt des Ausgabebildes zu berechnen. Dieses Verhalten erzielt also einen geringen Speicher-*Overhead* bei der Berechnung eines Speicher-Bildes aus Filter-Bildern, ist aber schwer quantifizierbar, weshalb dieser Speicher nicht unter die Speicherschranke fällt – i. d. R. hat dies kaum einen Einfluss auf die Speicherschranke.

Neben dieser vorteilhaften Vorgehensweise weist *libvips* jedoch auch weitere Vorzüge auf, die im Kontext dieser Arbeit relevant sind:

libvips nutzt einen *Cache* mit einstellbarer Größe, der Zwischenergebnisse temporär abspeichert und wiederverwendet, wenn möglich. Dies ist besonders effektiv, da Bilder in *libvips* (bis auf wenige Ausnahmen) unveränderlich sind und *Cache*-Einträge dadurch nur selten ungültig werden, wodurch die begrenzte *Cache*-Größe den einzigen Grund zur Entfernung aus dem *Cache* darstellt. Dieses System kann das wiederholte Berechnen derselben Zwischenergebnisse deutlich beschleunigen, in anderen Fällen ist es jedoch wenig effizient, wie in Abschnitt 6.2 zu sehen ist.

In Hinblick auf die Zahlentypen, die als Helligkeitswerte genutzt werden können, bietet *libvips* Einiges an Flexibilität, denn sowohl vorzeichenlose als auch vorzeichenbehaftete 8 Bit-, 16 Bit- und 32 Bit-Festkommazahlen, 32 Bit- und 64 Bit-Fließkommazahlen sowie zwei komplexe Zahlentypen als Paare der Fließkommatypen stehen zur Verfügung. Dabei ist es in *libvips* häufig der Fall, dass Opera-

tionen Ergebnisse im „kleinstmöglichen Fließkommatypen“ generieren, also im 32Bit-Fließkommatyp für alle Festkommatypen und andernfalls in unverändertem Typ.

5.2.1 Genutzte Operationen

Dies bietet eine gute Möglichkeit, auf die Operationen einzugehen, die innerhalb dieses Projektes genutzt werden, was zweigeteilt geschehen wird: Zunächst werden die bereits von *libvips* bereitgestellten Operationen und danach die von mir implementierten Operationen behandelt. Dabei besitzt, wie in Abschnitt 5.1 angekündigt, jede dieser Operationen einen zugeordneten Filterknotentyp hat, der es erlaubt, die hier geschilderten Parameter zu bearbeiten – gibt es dabei weitere Dinge zu beachten, wird darauf hingewiesen.

libvips selbst bietet eine beachtliche Menge an Operationen an, die viele einfache und einige komplexere Bildverbeitungsmöglichkeiten abdecken, wenn auch die Zahl an komplexen Operationen (bspw. im Vergleich zu *OpenCV*) etwas gering ist. Folgende vom Programm angebotenen Operationen bauen direkt auf von *libvips* angebotenen Operationen auf:

***Copy*:** *Eingabe*: Bild *i*. *Ausgabe*: Bild vom Eingabetyp.

Diese Operation führt bloß eine *Pointer*-Kopie mit Referenzzählung durch, benötigt also vernachlässigbar wenig Zeit.

***Gamma*:** *Eingabe*: Bild i; `double` γ. *Ausgabe*: Bild vom kleinstmöglichen Fließkommatyp.

Wendet auf jeden Helligkeitswert λ aus i die Funktion λ^γ an.

***Gaussian Blur*:** *Eingabe*: Bild i; `double` σ, m. *Ausgabe*: Bild vom kleinstmöglichen Fließkommatyp.

Berechnet das Gaußschen Weichzeichners mit Standardabweichung σ, wobei m die kleinste Amplitude der Gaußschen Funktion angibt, die beachtet werden soll. Zu beachten ist, dass *libvips* stets eine direkte Faltung durchführt, ohne jemals eine *Fourier*-Transformation durchzuführen.

***Join Channels*:** *Eingabe*: Maximal 4 Bilder i_j. *Ausgabe*: Bild vom kleinsten gemeinsamen Typ.

Erstellt ein Bild, dass aus den kombinierten Kanälen der Eingabebilder besteht, die alle zum kleinsten Typen konvertiert wurden, der alle Werte aufnehmen kann.

***Levels*:** *Eingabe*: Bild i; `double` α, ω. *Ausgabe*: Bild vom kleinstmöglichen Fließkommatyp.

Wendet auf jeden Helligkeitswert λ aus i die Funktion $\frac{\lambda-\omega}{\alpha-\omega}$ an, bildet also das Intervall $[\omega, \alpha]$ linear auf $[0, 1]$ ab.

***Load*:** *Eingabe*: `string` *s*. *Ausgabe*: Bild.

Lädt die Bild-Datei unter dem Pfad *s*, falls der Pfad gültig ist und der Dateityp unterstützt wird. *libvips* nutzt einige spezialisierte Bibliotheken sowie das generelle *ImageMagick*, um eine große Menge an Dateiformaten zu unterstützen.

***Scale*:** *Eingabe*: Bild *i*; `double` ξ, υ; *Kernel* κ. *Ausgabe*: Bild vom Eingabetyp.

Skaliert i in x-Richtung um Faktor ξ und in y-Richtung um Faktor υ unter Nutzung der Skalierungsmethode κ. *libvips* unterstützt als Skalierungs-*Kernel* die Optionen „Nächster Nachbar“, „linear“, „kubisch“, „*Lanczos* 2“ und „*Lanczos* 3“.

***Split Channels*:** *Eingabe*: Bild *i*. *Ausgabe*: 4 Bilder vom Eingabetyp.

Teilt das Eingabebild in max. 4 Kanäle auf. Liegen mehr als 4 Kanäle vor, werden alle weiteren verworfen, liegen weniger vor, sind alle weiteren Ausgaben leere Bilder.

libvips bietet außerdem die Möglichkeit, neue Operationen zu definieren, die die zu Beginn beschriebenen Leistungsvorteile von *libvips*-Operationen genießen, falls sie korrekt implementiert wurden, was ich ausgenutzt habe, um die folgenden Operationen zu definieren:

***Clamp*:** *Eingabe*: Bild *i*; `double` $\nu_{\min}$, $\nu_{\max}$. *Ausgabe*: Bild *j* vom Eingabetyp.

Behält das Intervall $[\nu_{\min}, \nu_{\max}]$ bei, setzt alle Werte, die kleiner als $\nu_{\min}$ sind, auf $\nu_{\min}$ und alle, die größer als $\nu_{\max}$ sind, auf $\nu_{\max}$.

***Convert Depth*:** *Eingabe*: Bild *i*; Zahlentyp *t*. *Ausgabe*: Bild *j* vom Typ *t*.

Konvertiert das Eingabebild unter Beibehaltung des Weißpunktes in den Zahlentyp *t*.

libvips bietet zwar eine Funktion an, um diese Umwandlung unter Beibehaltung der Zahlenwerte durchzuführen, doch nur für die Konversion zwischen Ganzzahltypen besteht die Möglichkeit, den Weißpunkt des Bildes durch einen *bit shift* beizubehalten. Die Problematik bei einem direkten *cast* ist jedoch, dass bspw. nach dieser Umwandlung von einem Ganzzahl- in einen Fließkommazahl (ohne weitere Informationen) nicht klar ist, wo der Weißpunkt liegt – bei 1, wie bei Gleitkommabildern üblich, oder bei 255, oder 65535, etc. Um den Weißpunkt beizubehalten, normalisiert `ConvertDepth` die Werte des Eingabebildes mit dem Quotienten der Weißpunkte des Ein-/ bzw. Ausgabe-Zahlentyps und konvertiert diese dann in den Ausgabetypen.

Diese Operation ist zusätzlich derart optimiert, dass bei gleichen Ein- und Ausgabetypen bloß kopiert wird sowie zwischen ganzzahligen Typen die bereits beschriebene *bit-shift*-Option ausgenutzt wird.

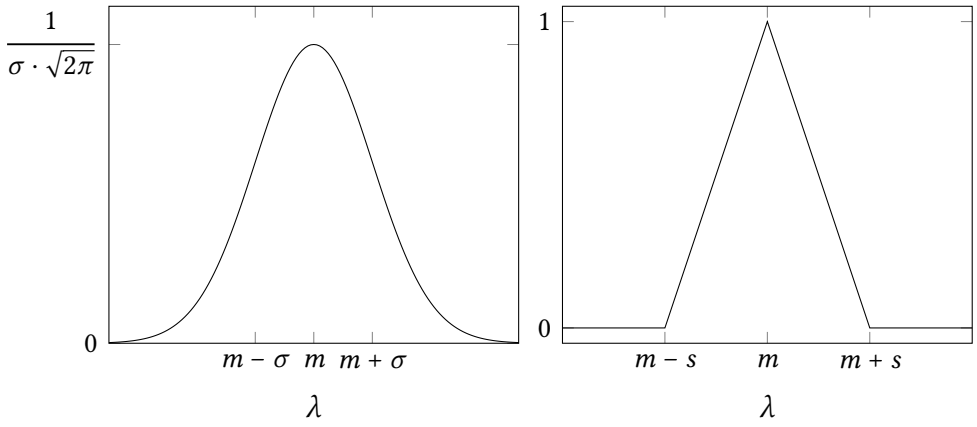

Abbildung 10: Die Graphen der durch *Intensity Gaussian* und *Linear Weights* angewandten Funktionen.

***Linear Blend*:** *Eingabe*: Bild i_1, i_2; `double` ζ. *Ausgabe*: Bild j vom kleinstmöglichen Fließkommatyp.

Wendet auf die Helligkeitswerte λ_1, λ_2 aus i_1, i_2 die Funktion $\zeta \cdot \lambda_1 + \big(1 - \zeta\big) \cdot \lambda_2$ an, mischt also i_1 mit Anteil ζ und i_2 mit Anteil $1 - \zeta$.

***Intensity Gaussian*:** *Eingabe*: Bild i; `double` σ, m. *Ausgabe*: Bild j vom kleinstmöglichen Fließkommatyp.

Berechnet für jeden Helligkeitswert λ aus i den Wert der Gaußschen Funktion $g_\sigma\big(\lambda - \mu\big)$, wie im ersten Graph von Abb. 10 dargestellt wird.

***Linear Weights*:** *Eingabe*: Bild i; `double` s, m. *Ausgabe*: Bild j vom kleinstmöglichen Fließkommatyp.

Wendet auf jeden Helligkeitswert λ aus i die Funktion $\max\left\{1 - \frac{|i-m|}{s}, 0\right\}$ an, wie im zweiten Graph von Abb. 10 dargestellt wird.

***Simple Channel Bilateral*:** *Eingabe*: Bild i; `double` σ_S, m_S, σ_I, s; `unsigned` κ. *Ausgabe*: Bild j vom kleinstmöglichen Fließkommatyp.

Berechnet auf jedem Farbkanal einen bilateralen Filter gemäß [DD02], wobei σ_S und μ_S die räumliche Weichzeichnung, σ_I die Intensitätsabweichung, κ die Anzahl Schritte und s den internen Skalierungsfaktor angibt.

Innerhalb der Berechnung dieser Operation werden die soeben beschriebenen *Intensity-Gaussian*- und *Linear-Weights*-Operationen genutzt. Diese Operation wird in der Folge noch einige Male Erwähnung finden, da er einige Probleme bereitet und aufgezeigt hat.

***Simple Leak*:** *Eingabe*: Bild i; `double` $\sigma_{\min}$, $\sigma_{\max}$, m; `unsigned` κ. *Ausgabe*: Bild j vom kleinstmöglichen Fließkommatyp.

Diese Operation berechnet das Ergebnis folgender Operation, wobei g_σ eine Gaußsche Weichzeichnung mit Radius σ und kleinster Amplitude m (siehe *Gaussian Blur*) darstellt:

$$\sum_{\lambda=0}^{\kappa-1} g_{\sigma_\lambda}(i) \text{ mit } \sigma_\lambda \coloneqq \sigma_{\min} + \frac{\lambda\,(\sigma_{\max} - \sigma_{\min})}{s-1}$$

Dies ist ein einfache Operation, der erstellt wurde, um einige Probleme, die beim bilateralen Filter auftraten, in einer weniger komplexen Operation diagnostizieren und demonstrieren zu können. In welcher Form dies geschah, wird weiter unten diskutiert.

5.2.2 Eigenheiten und Probleme

Damit bietet es sich an, einige der Eigenheiten und Probleme von *libvips* zu behandeln, die im Laufe dieses Projektes ersichtlich geworden sind. Bevor jedoch die tiefer gehenden Probleme diskutiert werden, möchte ich kurz einige Unannehmlichkeiten thematisieren, die die Behandlung anderer Aspekte erschwerten oder viel Zeit in Anspruch nahmen. Einerseits ist dabei *libvips*' Dokumentation zu nennen, die zwar die *Grundlagen* adäquat erklärt, aber bei jeglicher weiterer Beschäftigung mit dem Projekt unzulänglich ist, andererseits wies das Programm durch Missverständnisse bezüglich der Funktionsweise von *libvips* Speicherlecks auf, deren Auflösung einige Zeit in Anspruch nahm. Diese haben sich jedoch lösen lassen, und das Programm weist in seinem gegenwärtigen Stadium keine erkennbaren Speicherlecks mehr auf – weder *libvips*' eigene Leckentdeckung noch `valgrinds` [NS07] `memcheck` oder *Googles LeakSanitizer* [Goo18] kann nennenswerte Lecks im Programm finden.

Leider wies *Simple Channel Bilateral* auch nach der Korrektur der Speicherlecks noch immer ein fragwürdiges Speicherverhalten auf, bei dem selbst auch nach der Zerstörung eines Bildes Teile des Speichers allokiert bleiben. Dieser Speicher wird dann für das nächste Bild teilweise wieder verwendet, doch die Gesamtspeichernutzung steigt dennoch mit jedem verarbeiteten Bild in kleiner werdenen Schritten an, bevor ein *Plateau* erreicht wird. Ein ähnliches Verhalten lässt sich im ersten Graphen aus Abb. 11 beobachten, wobei zu beachten ist, dass *Caching* für dieses Beispiel deaktiviert wurde.

Dieses Verhalten war sehr verwirrend und führte zu einer länglichen Diskussion mit dem Autor der Bibliothek, John Cupitt, die unter [18b] zu finden ist. Um diese Diskussion auf das Speicherverhalten zu beschränken, wurde *Simple Leak* als einfachere Operation mit zum bilateralen Filter ähnlichem Speicherverhalten konzipiert, das sich im ersten Graphen aus Abb. 11 beobachten lässt, in dem die Operation 6 mal (die „Hügel“) mit jeweils 1 s Unterbrechung (die „Täler“) ausgeführt wird.

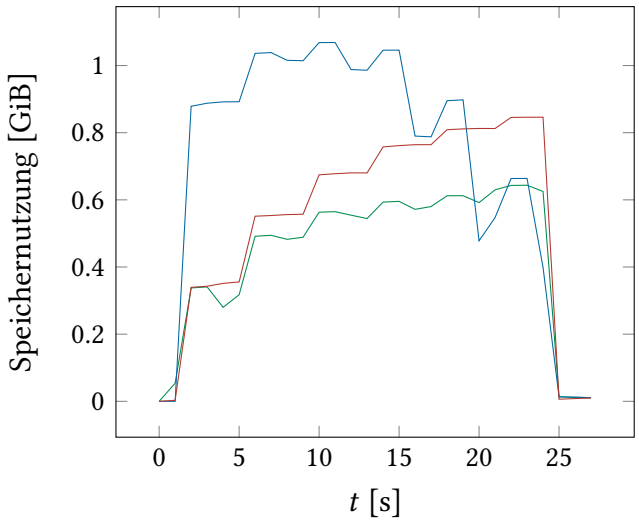

Abbildung 11: Speichernutzung bei sechsfacher Ausführung von *Simple Leak*. Die Eingabe besitzt dabei einen Kanal mit $2^{10} \times 2^{10}$ 32Bit-Fließkomma-Pixeln, die übrigen Argumente sind $\sigma_{\min} = 1, \sigma_{\max} = 4, m = 0.05, \kappa = 64$. Der *GCC*-Speicherallokator ist in grün, `jemalloc` in blau und `tcmalloc` in rot.

Im Laufe des Austausches wurde diagnostiziert, dass das Problem *Heap*-Fragmentierung zu sein scheint, dass also auch jeder bereits allokierte Speicherblock, von dem nur noch ein kleiner Teil genutzt wird, in seiner Gänze reserviert bleibt. Dieses Problem scheint bereits anderenorts aufgetreten zu sein, und als Lösungsansatz wurde die Nutzung eines alternativen Speicherallokators vorgeschlagen. Solche Bibliotheken implementieren u. a. die Funktionen, die unter *C* für die Zuteilung bzw. Freigabe von Speicher zuständig sind, auf andere Weise als in der Standard-Implementierung des *Compilers* und ermöglichen es somit, die Laufzeit und das Speicherverhalten des Programms in gewissen Maßen zu verändern. Im Rahmen dieses Problems wurden zwei solche Bibliotheken getestet, deren Speicherverläufe in Abb. 11 ebenfalls aufgezeigt werden: `jemalloc` [18a] und `tcmalloc` [G+18]. Diese Werkzeuge sind recht beliebt, und gerade `jemalloc` behauptet, eine geringe *Heap*-Fragmentierung zu bieten.

Wie in Abb. 11 aber bereits angedeutet wird, bietet keine dieser Alternativen ein konsistent besseres Verhalten: `jemalloc` ist in einigen Fällen besser, in anderen schlechter als der Standardallokator von *GCC*, und `tcmalloc` ist in keinem Fall besser und in vielen deutlich schlechter. Aus diesem Grund wurde die Entscheidung getroffen, den *GCC*-Allokator beizubehalten, das Problem hat sich also nicht lösen lassen.

Dies ist auch der Grund, weshalb keiner der Tests in Abschnitt 6.2 den bilateralen Filter beinhaltet: Wird dieser auch nur auf einem moderat großen Bild bei moderat niedrigen Speicherschranken benutzt, ist es geradezu unmöglich, diese

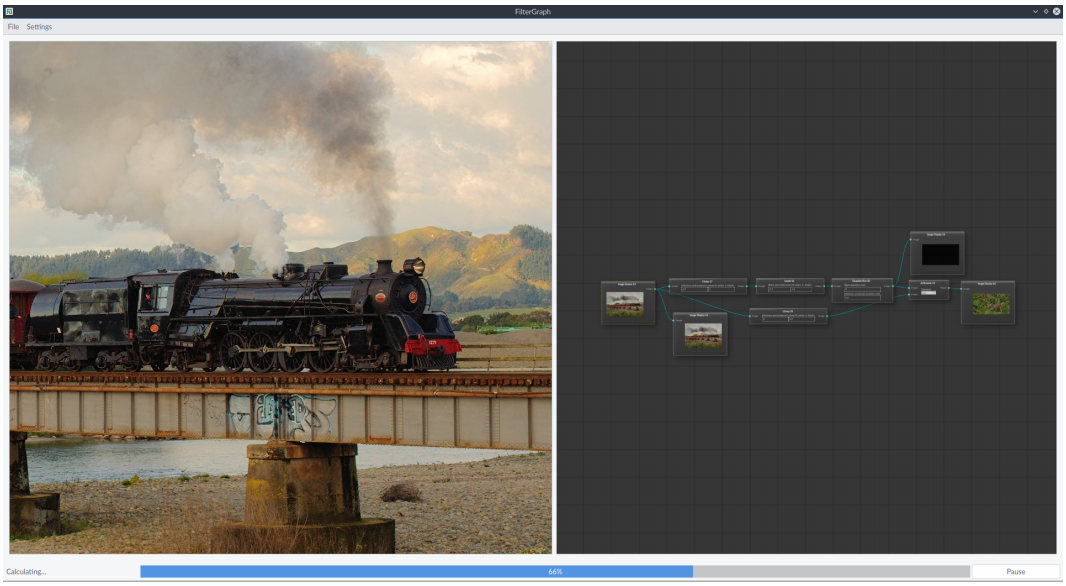

Abbildung 12: Eine Bildschirmaufnahme des FilterGraph. Die Ansicht zeigt– [Kar].

Schranke einzuhalten. Weitere Gedanken dazu sind in den Weiterentwicklungsmöglichkeiten (Abschnitt 7) zu finden.

5.3 Die Benutzeroberfläche

Die Benutzeroberfläche erfuhr nach ihrer initialen Erstellung nur wenige größere Veränderungen, denn für die graphische Nutzerschnittstelle wurde *Qt* 5 [18e] gewählt, das einen großen Funktionsumfang, großteils sinnvolle Schnittstellen und ein Aussehen, das sich dem „*Look and Feel*“ der Plattform gut anpasst, bietet. *Qt*s gute Dokumentation und die Existenz mehrerer Bibliotheken, die eine graphenbasierte Benutzeroberfläche für *Qt*5 anbieten (siehe *NodeEditor* aus Abschnitt 5.1), sind zusätzliche Vorteile dieser Bibliothek.

Der Hauptteil der Benutzeroberfläche ist, wie in Abb. 12 zu sehen, aufgeteilt in zwei Spalten, deren Größe vom Nutzer veränderbar ist. Der linke Teil der Benutzeroberfläche ist für die Ansicht eines Ergebnisses in voller Größe vorgesehen und beinhaltet somit ein Bild sowie horizontale bzw. vertikale *Scroll*-Balken, falls das Bild größer ist, als der zur Verfügung stehende Platz. Die Ansicht bleibt leer, bis der Nutzer einen Ansichtknoten auswählt, dessen Ergebnis angezeigt werden soll. Falls gegenwärtig keine Aktualisierung des Bildergraphen stattfindet, wird das gewählte Ergebnis asynchron berechnet, falls es nicht im Speicher ist, und die Ansicht mit diesem Bild aktualisiert. Falls doch auf das Ende einer Aktualisierung gewartet wird, wird die Ansicht während der Bearbeitung des Bilderknoten innerhalb der Aktualisierung aktualisiert. Der rechte Teil beinhaltet die Graphenansicht, die bereits beschrieben wurde.

Über diesen Spalten ist noch eine Menüleiste zu finden, die im Menü „*File*“ zwei Einträge anbietet: Einen zum Speichern des gegenwärtigen und einen zum Laden eines zuvor gespeicherten Graphen; die Speicherung wird, wie bereits angerissen, durch *NodeEditor* gehandhabt, der die relevanten Daten in einer *JSON*-basierten Textdatei mit der Endung „*flow*“ speichert. Das zweite Menü „*Settings*“ erlaubt es, über den Eintrag „*Memory Limit*“ die Speicherschranke einzustellen.

Daneben ist in der Statusleiste eine Textbox für eine Statusmeldung, eine Fortschrittsanzeige sowie ein Knopf zum Pausieren oder Fortsetzen der Bildberechnungen zu finden. Die beiden ersteren werden während der Aktualisierung des Bildergraphen aktualisiert, während die Einbindung des Knopfes bereits in Abschnitt 4.3 beschrieben wurde.

6 Resultate

Dieser Abschnitt dient primär zur Bewertung des beschriebenen Systems in Hinblick darauf, wie gut es in der Lage ist, die zu Beginn der Arbeit beschriebenen Ziele zu erreichen: Möglichst schnelle Berechnung bei eingeschränkter Speichernutzung. In Abschnitt 4.3 wurde die Vorgehensweise beschrieben, mit der angestrebt wird, dieses Ziel zu erfüllen, bei der die Berechnungszeit des Bildes als essenzieller Bestandteil von sinnvollen Berechnungsfunktionen eingeführt wurde. Um diesen Wert innerhalb der Bewertung nutzen zu können, ist es also notwendig zu wissen, wie viel Zeit die Operation, die ein Bilderknoten auf seine Eingaben anwendet, benötigt, um eine Speicherrepräsentation von sich selbst zu erstellen.

Die folgenden Tests wurden unter *OpenSUSE Tumbleweed* auf einem *AMD Ryzen 5 2600X* (6 Kerne, 12 *Threads*, 3.6 GHz Basis- und 4.2 GHz *Boost*-Takt) ausgeführt. Es lagen Arbeitsspeicher 16 GB an Arbeitsspeicher mit einem Takt von 2666 MHz vor, und Bilder wurden von einer *NVMe M.2 SSD* vom Modell *Samsung 970 EVO* geladen.

6.1 Laufzeiten der Filter

Um diese Laufzeiten während der Ausführung des Programms effizient approximieren zu können, wurden für jede der unterstützten Operationen ausgiebige Messungen ihrer Ausführungsdauer bei verschiedenen Belegungen der Parameter durchgeführt, aus denen Funktionen erstellt wurden, die die gemessenen Daten gut annähern und für die Nutzung im Programm implementiert wurden. Die Menge der dadurch gesammelten Daten ist sehr groß, was insbesondere durch die Zahl an Variablen, von denen die Laufzeit der Operationen abhängt, begründet ist – so hängt die Laufzeit jedes Filters von der Größe und dem Zahlenformat der Eingabe ab, wozu jeder Parameter, der vom Nutzer verändert werden kann, dazukommt. Abhängig vom Filter kann die eine Menge von über 30000 Messwerten bedeuten, weshalb diese nicht abgedruckt werden, sondern als *CSV*-Dateien auf dem beige-

legten Datenträger zur Verfügung stehen.

Diese Daten wurden in *Wolfram Mathematica* zur Parameterschätzung genutzt, bei der versucht wird, bestimmte Parameter einer vorgegebenen Funktionenform optimal zu belegen. Dies führt bei einer sinnvollen Wahl der Funktionenform und geringen Anzahl von Parametern meist zu guten Ergebnissen, doch gerade bei vielen Parametern, die potentiell Abhängigkeiten aufweisen, ist eine gewisse Ungenauigkeit zu erwarten. Aus diesem Grund wurde für die komplexeren Funktionen mit einer recht allgemeinen Funktionenform angefangen, deren Parameter die Parameterschätzung mit relativ hoher Streuung bestimmte; ließ sich jedoch für einzelne dieser Parameter ein einfaches Muster in ihrer Wertebelegung erkennen (waren die Werte bspw. fast konstant), so wurde der veränderbare Parameter durch dieses Muster ersetzt, woraufhin sichergestellt wurde, dass sich die Approximation nicht merklich verschlechtert hat. Dieses Vorgehen wurde wiederholt, bis sich keiner der Parameter sinnvoll ersetzen ließ, wodurch sich in den meisten Fällen gute Näherungen finden ließen, die eine geringe Anzahl an veränderbaren Parametern aufweisen, in anderen Fällen war es jedoch nötig, die Messdaten in mehrere Kategorien aufzuteilen, wodurch in einzelnen Kategorien ein gewisser Datenmangel ergeben konnte.

In der folgenden Aufzählung werden die Ergebnisse dieses Verfahrens zusammengefasst, wobei n die Anzahl an Pixeln und a, b die Parameter darstellen, die in den Tabellen im Anhang aufzufinden sind. Diese Tabellen enthalten neben den Werten der Parameter auch die Standardabweichung, die von *Mathematica* angegeben wurde, um einen Eindruck der Genauigkeit der Werte zu vermitteln.

***Arithmetic*:** Folgt der Funktionenform $a \cdot n$ mit den Werten aus Tab. 1.

***Clamp*:** Folgt der Funktionenform $a \cdot n$ mit den Werten aus Tab. 2, wobei leere Einträge den Wert 0 darstellen.

***Convert Depth*:** Folgt der Funktionenform $a \cdot n^b$ mit den Werten aus den Tabellen 4, 6 und 8.

***Extract Channel*:** Folgt der Funktionenform $a \cdot n$ mit den Werten aus Tab. 9.

***Gamma*:** Folgt der Funktionenform $a \cdot n$, wobei a entsprechend der Fallunterscheidung in Tab. 10 und den Werten aus den Tabellen 11 gewählt wird.

***Gaussian Blur*:** Folgt der folgenden Funktionenform (wobei σ und m die Argumente von *Gaussian Blur* sind) mit den Werten aus Tab. 12:

$$\left(a \cdot n \cdot \left(\log n + 12.5\right)\right) \cdot \left(0.4 \cdot \sigma^{1.64} \cdot \left(m^{-0.185} - 1.165 \cdot m\right) + 1\right)$$

***Join Channels*:** Folgt der Funktionenform $a \cdot n \cdot m$, wobei m die Anzahl an Eingabebildern darstellt, mit den Werten aus Tab. 13.

***Levels*:** Folgt der Funktionenform $a \cdot n^b$ mit den Werten aus Tab. 14.

Linear Blend: Folgt der Funktionenform $a \cdot n^{1.078}$ mit den Werten aus Tab. 15.

Load: Folgt der Funktion $8 \cdot 10^{-9} \cdot n$. Diese Funktion richtet sich an der Zeit, die es benötigt, ein Bild aus Gaußschen Rauschen im 8Bit-*PNG*-Format zu laden, und lässt sich nur schlecht für andere Formate verallgemeinern, fungiert also bloß als eine grobe Schätzung.

Scale: Folgt der Funktionsform aus Tab. 16, deren Bedingungen erfüllt werden, wobei s das Produkt der Skalierungsfaktoren in x- und y-Richtung darstellt und f sowie p Werte sind, die von einer Kombination aus Zahlenformat und Skalierungs-*Kernel* abhängen. Mit diesen Funktionsformen ergeben sich 300 Parameterwerte, weshalb auch diese Werte auf dem beigelegten Datenträger zu finden sind.

Simple Channel Bilateral: Folgt der folgenden Funktionenform (wobei die in der Einführung des bilateralen Filters genutzten Symbole wieder verwendet werden), mit den Werten aus Tab. 17:

$$\begin{aligned} & n \cdot \left(\log n + a\right) \cdot \left(0.4 \cdot \sigma_S^{1.64} \cdot \left(m_S^{-0.185} - 1.165 m_S\right) + 0.5\right) \\ & \cdot \left(s^{-2.84} + 0.0005\right) \cdot \left(\kappa + 0.51\right) \cdot \left(n^{10^{-11}} - 1\right) \\ & + 10^{-7} \cdot \left(s^{-0.66} + 2.5\right) \cdot \left(\kappa^{1.06} + 3.6\right) \cdot \left(n^{0.81} + 15800\right) \end{aligned}$$

Wie sich erahnen lässt, war es sehr aufwändig, die Konstanten dieses Ausdrucks mit der beschriebenen Methode der „Muster-Suche“ zu bestimmen, doch diese Funktion bietet (in Hinblick darauf, dass bloß ein einziger Parameter vorliegt) eine sehr gute Approximation der Messdaten.

6.2 Laufzeiten und Speicherverhalten des Programms

Mit diesen Funktionen lässt sich nun das Gesamtprogramm testen, wobei es zu zeigen gilt, dass das System bei eingeschränktem Speicher in der Lage ist, eine ähnliche Laufzeit zu erreichen, wie beim Laden aller Zwischenergebnisse, und in jedem Fall deutlich besser ist als das Nicht-Laden der Zwischenergebnisse. Diese Tests werden auf vier Beispiel-Graphen ausgeführt, für die zwei Freiheitsgrade festgelegt sind: Die Bewertungsfunktion und die Speicherschranken-/*Cache*-Konfiguration.

Innerhalb der Tests werden zwei Bewertungsfunktionen miteinander verglichen, deren Komponenten in Abschnitt 4.3 bereits motiviert und beschrieben wurden. $\varphi_1(i) \coloneqq T(i) \cdot p^+(i)$ wurde in ebendiesem Abschnitt schon als Bewertungsfunktion genutzt und beschreibt die Gesamtzeit, die mit der Berechnung des Bildes verbracht wird, falls es nicht in den Speicher geladen wird. $\varphi_2(i) \coloneqq \frac{\varphi_1(i)}{\mu_B(i)}$ berechnet den Quotienten aus dieser Dauer und dem Speicherverbrauch, um die Bilder zu laden, deren Gesamtberechnungsdauer pro Byte groß ist.

Neben diesen Bewertungsfunktionen wurden auch verschiedene Speicherkonfigurationen verglichen, also verschiedene Kombinationen aus Speichergrenze und *Cache*-Größe. Dies diente einerseits dazu, den Anstieg der Berechnungszeit bei geringer werdender Menge an verfügbarem Speicher einzuschätzen, andererseits sollte die Effektivität von *libvips*' *Cache* als Unterstützung oder gar Ersatz von Speicherbildern untersucht werden. Um dabei sinnvolle Werte zu nutzen wurde für jeden der Graphen bestimmt, wie viel Speicher benötigt wird, um alle Zwischenergebnisse zu laden, was hier mit $\mu_{\max}$ bezeichnet werden soll, sodass die Speicherkonfigurationen als Anteile dieses maximal benötigten Wertes entstehen:

$(\mu_{\max}, 0)$: Alle Bilder werden in den Speicher geladen, was üblicherweise der beste Fall ist.

$(0, 0)$: Keine Bilder werden in den Speicher geladen, was üblicherweise der schlechteste Fall ist.

$(0, \mu_{\max})$: Keine Bilder werden in den Speicher geladen, dafür kann ein theoretisch für alle Bilder ausreichender *Cache* genutzt werden. Dies soll prüfen, ob *libvips Cache* eine sinnvolle Alternative zum in dieser Arbeit beschriebenen System darstellt.

$\left(\frac{1}{2}\mu_{\max}, 0\right)$: Der halbe Maximalspeicher wird als Speicherschranke gesetzt. Dies prüft die Qualität der Bewertungsfunktionen in einem realistischen Szenario, in dem sie entscheiden müssen, wofür sie ihren Speicher verwenden.

$\left(\frac{1}{2}\mu_{\max}, \frac{1}{2}\mu_{\max}\right)$: Der Maximalspeicher wird gleichmäßig auf die Speicherschranke und den *Cache* verteilt. Dies prüft, ob eine Kombination aus eigenem Speichermanagement und *Cache* vorteilhaft ist.

$\left(\frac{1}{4}\mu_{\max}, \frac{1}{4}\mu_{\max}\right)$: Jeweils ein Viertel des Maximalspeichers wird der Speicherschranke und dem *Cache* zugewiesen. Ähnlich zur vorherigen Kombination, fordert aber eine harschere Auswahl der Speicherbilder.

Die ersten drei Speicherkonfigurationen sind unabhängig von der Bewertungsfunktion, da entweder alle oder keine Bilder geladen werden, für alle anderen werden alle bereits genannten Bewertungsfunktionen durchgegangen.

Damit lassen sich die Ergebnisse der eigentlichen Tests beschreiben, innerhalb derer für jede Kombination aus Graph, Bewertungsfunktion und Speicherkonfiguration eine für den Graphen festgelegte Folge von Aktionen ausgeführt wird, die aus vier Phasen besteht:

1. Erstellung des Graphen und initiales Laden in den Speicher. Die Ladeknoten der ersten beiden Graphen laden das erste Bild aus Abb. 13, die beiden letzten das zweite, wobei beide Bilder als vorzeichenlose 8 Bit-*JPEG*-Bilder mit 3 Kanälen vorlagen.

Abbildung 13: Die Bilder, die innerhalb der Beispiele genutzt wurden. Das erste Bild [Pav] wurde in den ersten beiden Tests genutzt und weist eine Größe von $4560 \cdot 3066$ Pixeln (ca. 14 Megapixel) auf, während das zweite [Kar] aus $2918 \cdot 1937$ Pixeln (etwa 5.7 Megapixel) besteht und in den beiden späteren Graphen genutzt wird.

2. Auswahl eines Ansichtknotens und dessen Ladevorgang.

3. Durchführung einer Veränderung am Graphen und darauffolgende Neuberechnung inkl. der zuvor gewählten Ansicht.

4. Auswahl eines anderen Ansichtknotens und dessen Ladevorgang.

Für jeden dieser Schritte wurde die Laufzeit gemessen, und für jede Veränderung am Filtergraphen wurde nachverfolgt, welche Bilderknoten in den Speicher geladen wurden – die dabei gesammelten Daten finden sich im Anhang, innerhalb dieses Kapitels werden die Ergebnisse ausgewertet. Aufgrund der begrenzten Seitenbreite finden sich Phasen 1 und 2 sowie 3 und 4 jeweils in separaten Tabellen, in denen die Knoten durch die Anfangsbuchstaben der Operationen und, falls nötig, ihre Argumente spezifiziert werden. Die Bilderknoten, die in einer Zeile mit einem „✓“ versehen sind, wurden mit der entsprechenden Speicherkonfiguration und ggf. Bewertungsfunktion geladen, „×“ gibt das Gegenteil an.

Der erste derart behandelte Graph ist der bereits eingeführte Beispielsgraph (siehe Abb. 14) mit $\mu_{\max} = 188$. Als Veränderung wurden die Verbindungen des zweiten und dritten Ausgangs des „Spalte Kanäle“-Knotens vertauscht und auf die letzte Phase wurde verzichtet, da es für diesen Graphen keinen zweiten Ansichtknoten gibt. Die Daten finden sich in der Tabelle 18.

In der Initialisierungsphase verhalten sich die Bewertungsfunktionen nahezu optimal, doch bei der Ansicht und der Aktualisierung finden sich Werte Berechnungszeiten, die um einen Faktor von 2 bis 2.5 schlechter sind als der beste Fall, aber dennoch weniger als ein Fünftel des schlimmsten Falls. Diese Werte sind bei φ_1 leicht besser als bei φ_2, insbesondere ohne *Cache*, und während der *Cache* gerade bei der Veränderung die Berechnungszeit für φ_2 durchaus verbessern kann, werden leicht bessere Ergebnisse erreicht, wenn dieser Speicher der Speicherschranke

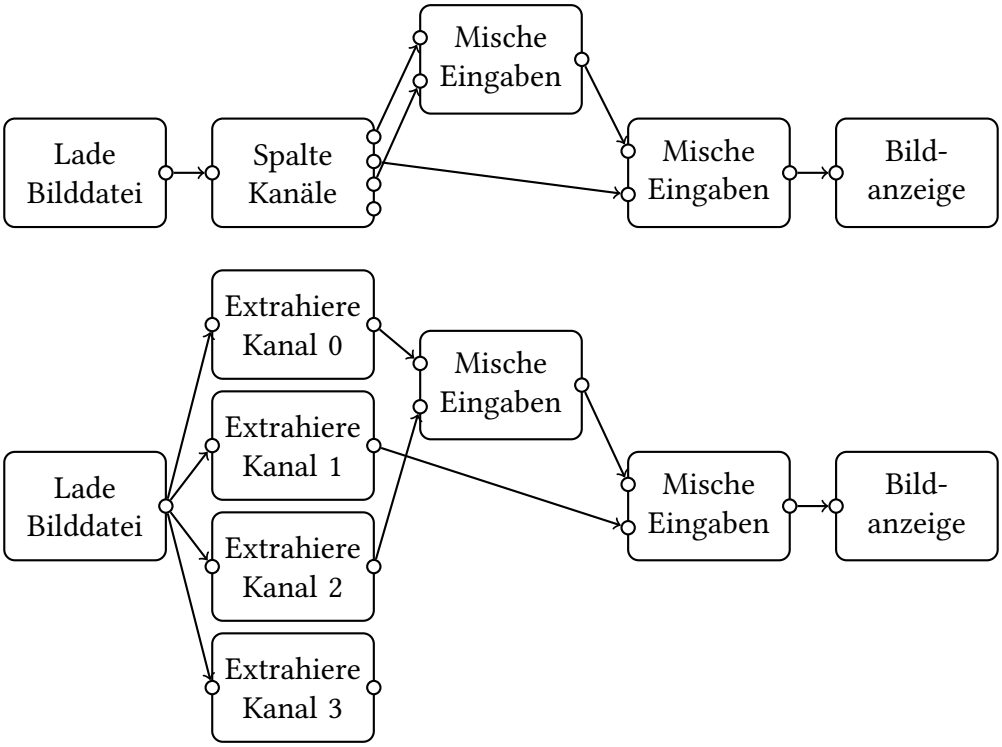

Abbildung 14: Der erste Testgraph, sowohl als Filter- als auch als Bildergraph.

zugewiesen wird.

Der nächste Graph berechnet einen Scharfzeichnungsfilter, wie in Abb. 15 zu sehen ist: Mithilfe einer Gaußschen Weichzeichnung sowie eines Subtraktionsknotens werden die Kanten des Bildes isoliert, die daraufhin skaliert und auf den Quellknoten addiert werden. Zusätzlich existiert ein Zweig, auf dem die Differenz mit einer Verschiebung des Wertebereichs berechnet wird. In der Veränderungsphase wurde die Stärke der Scharfzeichnung verringert, indem der erste *Levels*-Knoten 2 (anstelle von 5) als Weißwert übergeben bekommt. Dies benötigt deutlich mehr Speicher als die übrigen Tests – um alle Bilderknoten im Speicher zu halten, werden 840 MiB benötigt.

Für diesen Graphen sind fast alle Werte aus Tab. 19 kleiner als das 1.3-fache des Optimums, bloß das Laden des aktualisierten Graphen dauert bei nur 210 MiB an Arbeitsspeicher etwa das Doppelte, da der Skalierungsknoten nicht mehr geladen werden konnte – insgesamt sehr gute Ergebnisse. Der *Cache* stellt sich dabei als weniger nützlich heraus, als im vorherigen Beispiel – bei 420 MiB Speicherschranke verändert sich nichts durch den *Cache*, und bei ausschließlicher Nutzung eines *Cache*s voller Größe sind die Ergebnisse weitaus schlechter als selbst bei nur 210 MiB an Speicherschranke.

Der in Abb. 16 dargestellte Graph stellt eine interessante Verarbeitung dar, bei der die hellen Teile des Bildes von den dunklen getrennt, weichgezeichnet und

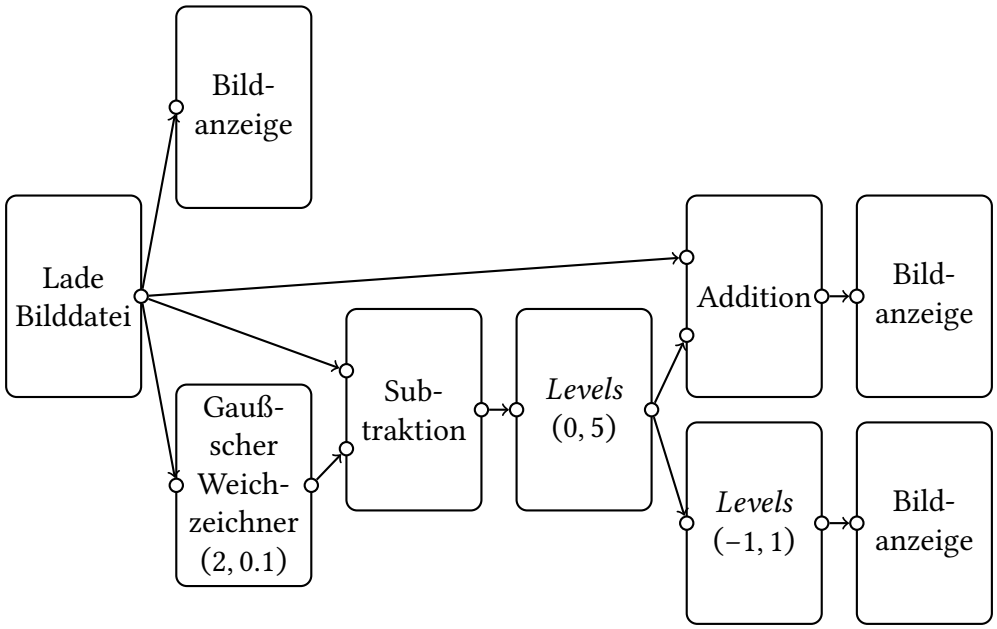

Abbildung 15: Der zweite Testgraph, bei dem Filter- und Bildergraph identisch sind.

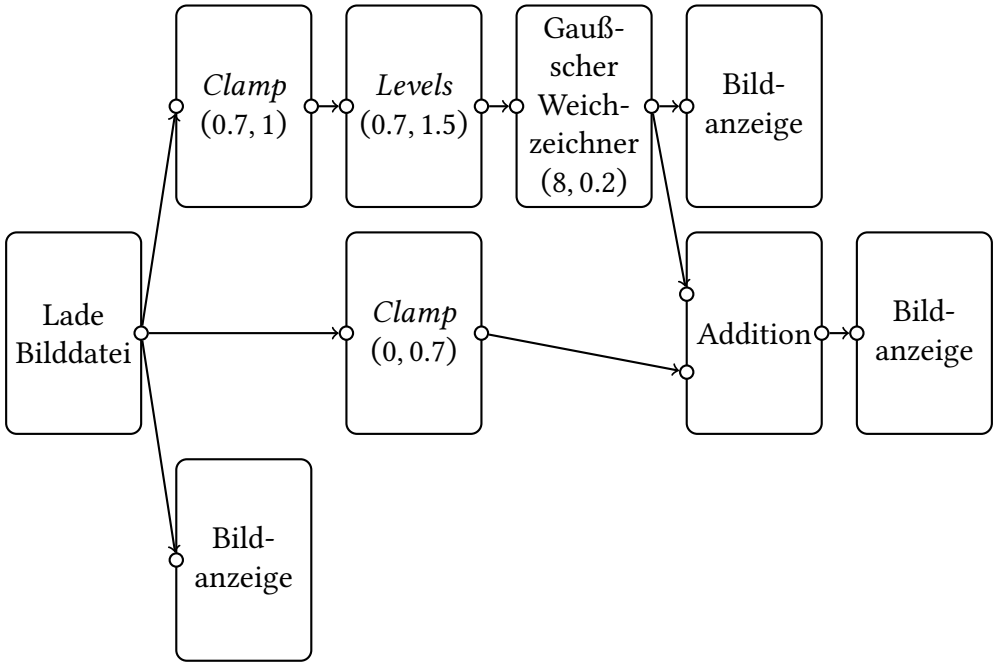

Abbildung 16: Der dritte Testgraph, bei dem Filter- und Bildergraph identisch sind.

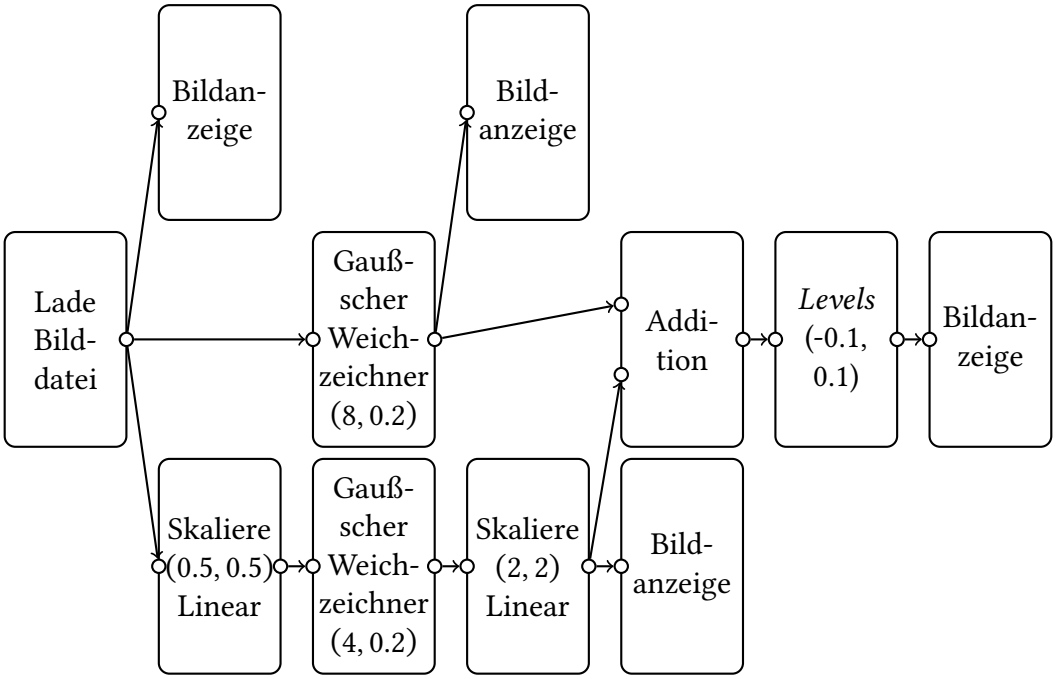

Abbildung 17: Der vierte Testgraph, bei dem Filter- und Bildergraph identisch sind.

dann wieder auf den dunklen Teil addiert werden. Der Graph erfordert 244 MiB an Arbeitsspeicher für alle Zwischenergebnisse und setzt in seiner Veränderungsphase den Weißpunkt des *Levels*-Knoten auf 1.7, wodurch die hellen Bereiche des Bildes nicht mehr aufgehellt werden. Die Daten finden sich in Tab. 20.

Auch hier ergeben sich für 122 MiB als Speicherschranke sehr gute Werte, die sich hier kaum von den besten Werten unterscheiden, während bei 61 MiB der Speicher nicht mehr ausreicht, um den Gaußschen Weichzeichner zu laden, was die Laufzeit auf das Niveau des schlimmsten Falles verschlechtert. Der *Cache* ergibt nur für die Ansicht nach der Aktualisierung messbare Unterschiede, und während der *Cache* voller Größe hier sehr gute Werte liefert, schwanken die übrigen Werte zu stark, um etwas Definitves über seine Auswirkungen zu sagen.

Der letzte und, aus Sicht der Ergebnisse, interessanteste Graph ist Abb. 17 mit Daten aus Tab. 21, der einen Gaußschen Weichzeichner und eine Simulation des Gaußschen Weichzeichners durch Skalieren und verringerte Weichzeichnung voneinander abzieht und diese Differenz sichtbar macht. Innerhalb der Veränderungsphase werden die Radii der Gaußschen Weichzeichner jeweils auf 12 und 6 gesetzt, und im letzten Schritt wird der Anzeigeknoten, der mit „Skaliere (2,2) linear“ verbunden ist, ausgewählt. Dies erfordert 296 MiB für alle Bilder.

Im Falle einer Speicherschranke von 148 MiB ergibt φ_2 insgesamt um ca. 15% bessere Werte, da es den ersten Skalierungsknoten anstelle des Ladeknotens lädt und damit verhindert, dass diese Skalierung mehrfach durchgeführt wird. Beide Funktionen ergeben jedoch, in Anbetracht der vielen Zwischenergebnisse, die nicht ge-

laden werden konnten, sehr gute Werte für die Ladevorgänge – sie sind um einen Faktor von 2 bis 2.5 langsamer als das Optimum, aber um einen Faktor von 9 bis 13 schneller als der schlechteste Fall. Das Berechnen der ersten Ansicht ist dabei ähnlich schnell, doch die Berechnung der zweiten Ansicht ist in jedem der Fälle äußerst nah am schlimmsten Fall, da auf dem unteren Ast des Graphen nur der Lade- bzw. erste Skalierungsknoten im Speicher ist, was insbesondere die Gaußsche Weichzeichnung der erneuten Berechnung überlässt. Werden nur 74 MiB zur Verfügung gestellt, liegen die Laufzeiten bei ca. der Hälfte des *Worst Case*, bis auf die End-Ansicht, die ebenso schlecht ist wie eben beschrieben. Der *Cache* bringt auch hier kaum messbare Verbesserungen – im Besten Fall ca. 5%.

Anhand dieser Auswertungen lässt sich ein Trend beobachten, bei dem beide Bewertungsfunktionen bei halbem Speicher gute Laufzeiten erreichen, ohne das eine konsistent oder bedeutend besser war, als die andere. Diese Werte liegen sehr wenigen Ausnahmen zwischen dem 1- und 2.5-fachen des besten Wertes, doch sind sie in den schlechteren Fällen noch immer um einen Faktor von 5 bis 13 schneller als der schlechteste Fall. Wird der verfügbare Speicher jedoch auf ein Viertel der für alle Zwischenergebnisse nötigen Menge gesetzt, sind die Laufzeiten sehr inkonsistent – mal nahe am besten, mal am schlechtesten Wert, meistens aber dennoch deutlich besser als der letztere Wert. Der *Cache* hat sich wiederum insgesamt als wenig bis gar nicht sinnvoll erwiesen, und in keinem Fall war es besser, den Speicher dem *Cache* zuzuweisen, als den Speicherbildern.

7 Weiterentwicklungsmöglichkeiten

Auch wenn das System in seinem gegenwärtigen Stand funktionsfähig und performant ist, gibt es noch Punkte, die verbessert oder erweitert werden könnten.

So können Veränderungen am Filtergraphen nicht rückgängig gemacht werden, was jedoch nur beim Löschen von Knoten signifikant ist, da das Hinzufügen von Filterknoten, das Verändern ihrer Verbindungen oder Parameter einfach manuell rückgängig gemacht werden kann. Auch wurden Funktionalitäten wie das Zeichnen mit Pinseln von der Implementierung ausgeschlossen, da sie in Hinblick auf Nicht-Destruktivität eine signifikante Vergrößerung des Problems herbeigeführt hätte – mit einem weitaus größerem Zeitrahmen wäre dies sicherlich eine sinnvolle Erweiterung.

Das gegenwärtige System ist weitgehend unabhängig von der genutzten Grafikbibliothek, weshalb es interessant wäre interessant, Unterstützung für weitere Bibliotheken zu implementieren und mit *libvips* zu vergleichen oder eine spezialisierte Bibliothek zu entwickeln, die einige zusätzliche Funktionalität für die Nutzung innerhalb eines Bildbearbeitungssystems besitzt.

Zur weiteren Verringerung der Zeit, die benötigt wird, um nach einer Veränderung die Ansicht neu zu berechnen, wäre es potentiell sinnvoll, zunächst bloß den sichtbaren Teil der Ansicht zu berechnen, um dann den Rest hinzuzufügen.

libvips erlaubt es auch, nur Ausschnitte eines Bildes zu berechnen, doch hätte dies zur Folge, dass während der Berechnung neuer Ergebnisse zuerst nur der sichtbare Ausschnitt und dann das gesamte Bild berechnet werden müsste, der sichtbare Ausschnitt also doppelt berechnet würde. Um dies zu verändern, wären massive Veränderungen am Bildverarbeitungssystem, die Teile der *libvips*-Funktionalität modifiziert nachbilden müssten, oder die Nutzung einer von *libvips* sehr verschiedenen Graphikbibliothek notwendig.

Damit das System auch auf anderen Rechnern gut funktioniert könnte es sinnvoll sein, ein zusätzliches System zu konzipieren, das systemspezifische Laufzeitmessungen durchführen kann und diese in einer Datei abspeichert, die dieses System dann einlesen, zur Berechnung verwenden und bei eigenen Berechnungen erweitern könnte.

Die Messung des verbrauchten Speichers stellt eine Einschränkung des hiesigen Ansatzes dar: Während der Speicherentscheidungen wird bloß der für den Bildergraphen verbrauchte Speicher gezählt und eingeschränkt, doch während der Berechnung von Ansichten können temporär beträchtliche Mengen an Speicher benötigt werden, die hier unbeachtet gelassen werden. Dies ließe sich beheben, indem bei jeder Aktualisierung die Größe des größten Anzeigeknotens von der Speicherschranke abgezogen wird, um so instgesamt innerhalb der Speicherschranke zu bleiben, hier wurde jedoch entschieden, nur den durch den Bildergraphen genutzten Speicher zu betrachten.

8 Fazit

Zu Beginn der Arbeit wurden einige Ziele und Anforderungen formuliert, die im Laufe dieser Arbeit referenziert, erläutert und erfüllt wurden: Es liegt ein graphenbasierter Bildeditor vor, der es erlaubt, die Speichernutzung einzuschränken und dabei gute Laufzeiten zu erzielen, falls die Einschränkung nicht zu extrem ist. Das dazugehörige System zur Auswahl der Speicher-Bilder ist flexibel und erlaubt es, Bilder auf beliebige Weise zu bewerten und somit die Funktionsweise zu optimieren, und durch entsprechende Tests wurden einige Kandidaten für besonders gute Funktionen bestätigt. Das Bildverarbeitungssystem auf Basis von *libvips* ist effizient und in der Lage, auch dann mit geringem *Overhead* zu arbeiten, wenn Zwischenergebnisse nicht im Speicher vorliegen, und durch die Nutzung bestehender und Implementierung neuer Operationen bietet das Programm bereits genug Vielfalt, um einige nützliche Verarbeitung durchzuführen – vier solche sinnvolle Filtergraphen wurden für die Tests genutzt. Auch die Nutzerschnittstelle ist funktional und bietet, im Zusammenhang mit der Bildverarbeitung, einige Komfortfunktionen wie das Pausieren und Fortsetzen von Berechnungen und arbeitet entkoppelt von der Bildverarbeitung, bleibt also stets responsiv.

Stellenweise wurde der schwierige, durch fehlende Erfahrung gekenntzeichnete Entwicklungsprozess erwähnt, doch gerade in Anbetracht dieser Probleme ist

das Ende des gewundenen Pfades ein solide funktionierendes und performantes System, das seine Ziele bereits erfüllt und eine gute Basis dafür bildet, den Funktionsumfang zu einem vollständigen Bildbearbeitungssystem zu erweitern.

9 Literatur

[18a] *jemalloc*. https://github.com/jemalloc/jemalloc. 2018.

[18b] *Memory usage when VIPS claims everything is okay*. https://github.com/libvips/libvips/issues/1064. 2018.

[18c] *nip2*. https://github.com/libvips/nip2. 2018.

[18d] *PhotoFlow*. https://github.com/aferrero2707/PhotoFlow. 2018.

[18e] *Qt 5*. http://doc.qt.io/qt-5/. 2018.

[Bra00] G. Bradski. „The OpenCV Library“. In: *Dr. Dobb's Journal of Software Tools* (2000).

[CM96] John Cupitt und Kirk Martinez. „VIPS: An image processing system for large images“. In: *SPIE Conference Proceedings*. Bd. 2663. 1996, S. 16–28.

[DD02] Frédo Durand und Julie Dorsey. „Fast Bilateral Filtering for the Display of High-dynamic-range Images“. In: *Proceedings of the Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '02. San Antonio, Texas: ACM, 2002, S. 257–266. ISBN: 1-58113-521-1. URL: http://doi.acm.org/10.1145/566570.566574.

[G+18] Sanjay Ghemawat, Paul Menage u. a. *gperftools*. https://github.com/gperftools/gperftools. 2018.

[Goo18] Google. *Sanitizers*. https://github.com/google/sanitizers. 2018.

[Kar] Karora. *JA1271 crossing the Otaki River*. https://commons.wikimedia.org/wiki/File:JA1271_crossing_the_Otaki_River.jpg.

[MC05] Kirk Martinez und John Cupitt. „VIPS – a highly tuned image processing software architecture“. In: *IEEE International Conference on Image Processing*. 2005, S. 574–577.

[NS07] Nicholas Nethercote und Julian Seward. „Valgrind: A framework for heavyweight dynamic binary instrumentation“. In: *In Proceedings of the Programming Language Design and Implementation Conference*. 2007.

[Pav] Pavlikhin. *Gothic Chapel Peterhof*. https://commons.wikimedia.org/wiki/File:Gothic_Chapel_Peterhof_tonemapped.jpg.

[Pin+18] Dmitry Pinaev u. a. *Qt5 Node Editor*. https://github.com/paceholder/nodeeditor. 2018.

10 Anhang

Format	a	a Std.-Abw.
Unsigned 8	$2.07 \cdot 10^{-9}$	$5.86 \cdot 10^{-11}$
Signed 8	$2.29 \cdot 10^{-9}$	$6.13 \cdot 10^{-11}$
Unsigned 16	$2.08 \cdot 10^{-9}$	$5.83 \cdot 10^{-11}$
Signed 16	$2.10 \cdot 10^{-9}$	$5.81 \cdot 10^{-11}$
Unsigned 32	$2.10 \cdot 10^{-9}$	$5.88 \cdot 10^{-11}$
Signed 32	$2.09 \cdot 10^{-9}$	$5.91 \cdot 10^{-11}$
Float 32	$1.05 \cdot 10^{-9}$	$3.92 \cdot 10^{-11}$
Float 64	$1.87 \cdot 10^{-9}$	$1.11 \cdot 10^{-10}$
Complex 32	$2.14 \cdot 10^{-9}$	$1.18 \cdot 10^{-10}$
Complex 64	$4.27 \cdot 10^{-9}$	$2.46 \cdot 10^{-10}$

Tabelle 1: Werte für *Arithmetic*.

Format	a	a Std.-Abw.
Unsigned 8	$4.73 \cdot 10^{-10}$	$6.58 \cdot 10^{-12}$
Signed 8	$5.21 \cdot 10^{-09}$	$1.28 \cdot 10^{-10}$
Unsigned 16	$9.73 \cdot 10^{-10}$	$7.28 \cdot 10^{-12}$
Signed 16	$3.06 \cdot 10^{-09}$	$1.26 \cdot 10^{-11}$
Unsigned 32	$2.85 \cdot 10^{-09}$	$2.46 \cdot 10^{-11}$
Signed 32	$2.87 \cdot 10^{-09}$	$2.44 \cdot 10^{-11}$
Float 32	$3.18 \cdot 10^{-09}$	$1.45 \cdot 10^{-11}$
Float 64	$4.27 \cdot 10^{-09}$	$3.36 \cdot 10^{-11}$
Complex 32	$3.55 \cdot 10^{-09}$	$1.49 \cdot 10^{-11}$
Complex 64	$4.64 \cdot 10^{-09}$	$3.85 \cdot 10^{-11}$

Tabelle 2: Werte für *Clamp*.

Eingabeformat	Ausgabeformat	a	a Std.-Abw.	b	b Std.-Abw.
Unsigned 8	Unsigned 8	$1.20 \cdot 10^{-10}$	$9.70 \cdot 10^{-11}$	1.020	0.042
Unsigned 8	Signed 8	$2.37 \cdot 10^{-9}$	$1.14 \cdot 10^{-10}$	1.006	0.003
Unsigned 8	Unsigned 16	$8.61 \cdot 10^{-10}$	$1.65 \cdot 10^{-10}$	1.007	0.010
Unsigned 8	Signed 16	$2.98 \cdot 10^{-9}$	$1.17 \cdot 10^{-10}$	0.997	0.002
Unsigned 8	Unsigned 32	$3.64 \cdot 10^{-9}$	$1.13 \cdot 10^{-9}$	0.937	0.016
Unsigned 8	Signed 32	$3.11 \cdot 10^{-9}$	$6.27 \cdot 10^{-10}$	0.964	0.011
Unsigned 8	Float 32	$3.84 \cdot 10^{-9}$	$1.49 \cdot 10^{-9}$	0.925	0.020
Unsigned 8	Float 64	$9.32 \cdot 10^{-10}$	$2.85 \cdot 10^{-10}$	1.038	0.016
Unsigned 8	Complex 32	$1.36 \cdot 10^{-9}$	$3.55 \cdot 10^{-10}$	1.018	0.014
Unsigned 8	Complex 64	$3.19 \cdot 10^{-12}$	$2.65 \cdot 10^{-12}$	1.369	0.043
Signed 8	Unsigned 8	$2.85 \cdot 10^{-9}$	$1.19 \cdot 10^{-10}$	0.995	0.002
Signed 8	Signed 8	$7.84 \cdot 10^{-11}$	$3.83 \cdot 10^{-11}$	1.042	0.026
Signed 8	Unsigned 16	$1.86 \cdot 10^{-9}$	$2.02 \cdot 10^{-10}$	1.020	0.006
Signed 8	Signed 16	$9.86 \cdot 10^{-10}$	$1.08 \cdot 10^{-10}$	1.000	0.006
Signed 8	Unsigned 32	$3.30 \cdot 10^{-9}$	$7.05 \cdot 10^{-10}$	0.976	0.011
Signed 8	Signed 32	$3.11 \cdot 10^{-9}$	$9.48 \cdot 10^{-10}$	0.945	0.016
Signed 8	Float 32	$4.04 \cdot 10^{-9}$	$1.65 \cdot 10^{-9}$	0.922	0.021
Signed 8	Float 64	$1.38 \cdot 10^{-9}$	$3.04 \cdot 10^{-10}$	1.017	0.011
Signed 8	Complex 32	$1.28 \cdot 10^{-9}$	$2.46 \cdot 10^{-10}$	1.022	0.010
Signed 8	Complex 64	$3.22 \cdot 10^{-12}$	$2.47 \cdot 10^{-12}$	1.368	0.040
Unsigned 16	Unsigned 8	$5.35 \cdot 10^{-10}$	$1.23 \cdot 10^{-10}$	0.993	0.012
Unsigned 16	Signed 8	$2.96 \cdot 10^{-9}$	$2.24 \cdot 10^{-10}$	0.995	0.004
Unsigned 16	Unsigned 16	$5.60 \cdot 10^{-11}$	$2.62 \cdot 10^{-11}$	1.094	0.024
Unsigned 16	Signed 16	$2.63 \cdot 10^{-9}$	$2.17 \cdot 10^{-10}$	1.003	0.004
Unsigned 16	Unsigned 32	$3.03 \cdot 10^{-9}$	$7.62 \cdot 10^{-10}$	0.950	0.013
Unsigned 16	Signed 32	$2.88 \cdot 10^{-9}$	$6.10 \cdot 10^{-10}$	0.969	0.011
Unsigned 16	Float 32	$3.71 \cdot 10^{-9}$	$1.27 \cdot 10^{-9}$	0.927	0.018
Unsigned 16	Float 64	$1.12 \cdot 10^{-9}$	$2.86 \cdot 10^{-10}$	1.028	0.013
Unsigned 16	Complex 32	$1.21 \cdot 10^{-9}$	$2.82 \cdot 10^{-10}$	1.025	0.012
Unsigned 16	Complex 64	$1.81 \cdot 10^{-12}$	$1.71 \cdot 10^{-12}$	1.400	0.049
Signed 16	Unsigned 8	$2.60 \cdot 10^{-9}$	$1.07 \cdot 10^{-10}$	1.000	0.002
Signed 16	Signed 8	$6.41 \cdot 10^{-10}$	$2.57 \cdot 10^{-10}$	0.983	0.021
Signed 16	Unsigned 16	$2.95 \cdot 10^{-9}$	$1.39 \cdot 10^{-10}$	0.996	0.002
Signed 16	Signed 16	$2.25 \cdot 10^{-10}$	$8.26 \cdot 10^{-11}$	1.022	0.019
Signed 16	Unsigned 32	$3.05 \cdot 10^{-9}$	$3.30 \cdot 10^{-10}$	0.981	0.006
Signed 16	Signed 32	$3.35 \cdot 10^{-9}$	$6.09 \cdot 10^{-10}$	0.941	0.010
Signed 16	Float 32	$3.76 \cdot 10^{-9}$	$1.32 \cdot 10^{-9}$	0.927	0.018
Signed 16	Float 64	$1.14 \cdot 10^{-9}$	$2.87 \cdot 10^{-10}$	1.028	0.013
Signed 16	Complex 32	$1.06 \cdot 10^{-9}$	$2.85 \cdot 10^{-10}$	1.032	0.014
Signed 16	Complex 64	$3.06 \cdot 10^{-12}$	$2.75 \cdot 10^{-12}$	1.372	0.047

Tabelle 4: Werte für *Convert Depth*, Teil 1.

Eingabeformat	Ausgabeformat	a	a Std.-Abw.	b	b Std.-Abw.
Unsigned 32	Unsigned 8	$3.91 \cdot 10^{-10}$	$1.01 \cdot 10^{-10}$	1.024	0.013
Unsigned 32	Signed 8	$2.77 \cdot 10^{-9}$	$1.67 \cdot 10^{-10}$	0.999	0.003
Unsigned 32	Unsigned 16	$1.04 \cdot 10^{-9}$	$1.63 \cdot 10^{-10}$	0.974	0.008
Unsigned 32	Signed 16	$2.75 \cdot 10^{-9}$	$1.19 \cdot 10^{-10}$	1.001	0.002
Unsigned 32	Unsigned 32	$3.57 \cdot 10^{-8}$	$2.74 \cdot 10^{-8}$	0.780	0.040
Unsigned 32	Signed 32	$3.06 \cdot 10^{-9}$	$5.88 \cdot 10^{-10}$	0.967	0.010
Unsigned 32	Float 32	$2.65 \cdot 10^{-9}$	$7.77 \cdot 10^{-10}$	0.946	0.015
Unsigned 32	Float 64	$2.26 \cdot 10^{-10}$	$1.60 \cdot 10^{-10}$	1.115	0.037
Unsigned 32	Complex 32	$1.89 \cdot 10^{-10}$	$1.26 \cdot 10^{-10}$	1.125	0.035
Unsigned 32	Complex 64	$2.56 \cdot 10^{-12}$	$1.76 \cdot 10^{-12}$	1.384	0.036
Signed 32	Unsigned 8	$2.48 \cdot 10^{-9}$	$1.46 \cdot 10^{-10}$	1.000	0.003
Signed 32	Signed 8	$4.91 \cdot 10^{-10}$	$1.50 \cdot 10^{-10}$	0.998	0.016
Signed 32	Unsigned 16	$2.59 \cdot 10^{-9}$	$1.81 \cdot 10^{-10}$	1.000	0.004
Signed 32	Signed 16	$6.36 \cdot 10^{-10}$	$1.21 \cdot 10^{-10}$	0.999	0.010
Signed 32	Unsigned 32	$3.99 \cdot 10^{-9}$	$5.89 \cdot 10^{-10}$	0.981	0.008
Signed 32	Signed 32	$4.78 \cdot 10^{-8}$	$3.81 \cdot 10^{-8}$	0.765	0.042
Signed 32	Float 32	$3.31 \cdot 10^{-9}$	$9.27 \cdot 10^{-10}$	0.932	0.015
Signed 32	Float 64	$1.87 \cdot 10^{-10}$	$1.33 \cdot 10^{-10}$	1.124	0.037
Signed 32	Complex 32	$1.73 \cdot 10^{-10}$	$1.15 \cdot 10^{-10}$	1.128	0.035
Signed 32	Complex 64	$1.65 \cdot 10^{-12}$	$1.20 \cdot 10^{-12}$	1.407	0.038
Float 32	Unsigned 8	$2.51 \cdot 10^{-9}$	$8.20 \cdot 10^{-11}$	1.000	0.002
Float 32	Signed 8	$2.94 \cdot 10^{-9}$	$1.49 \cdot 10^{-10}$	0.992	0.003
Float 32	Unsigned 16	$6.83 \cdot 10^{-9}$	$4.42 \cdot 10^{-10}$	0.997	0.003
Float 32	Signed 16	$2.77 \cdot 10^{-9}$	$1.11 \cdot 10^{-10}$	0.998	0.002
Float 32	Unsigned 32	$3.65 \cdot 10^{-9}$	$3.85 \cdot 10^{-10}$	0.968	0.006
Float 32	Signed 32	$4.44 \cdot 10^{-9}$	$1.05 \cdot 10^{-9}$	0.941	0.012
Float 32	Float 32	$3.32 \cdot 10^{-8}$	$2.88 \cdot 10^{-8}$	0.784	0.046
Float 32	Float 64	$8.02 \cdot 10^{-11}$	$8.11 \cdot 10^{-11}$	1.152	0.053
Float 32	Complex 32	$2.32 \cdot 10^{-10}$	$1.63 \cdot 10^{-10}$	1.110	0.037
Float 32	Complex 64	$1.44 \cdot 10^{-12}$	$1.10 \cdot 10^{-12}$	1.412	0.040
Float 64	Unsigned 8	$2.23 \cdot 10^{-9}$	$1.72 \cdot 10^{-10}$	1.012	0.004
Float 64	Signed 8	$2.03 \cdot 10^{-9}$	$1.40 \cdot 10^{-10}$	1.017	0.004
Float 64	Unsigned 16	$2.04 \cdot 10^{-9}$	$8.50 \cdot 10^{-11}$	1.020	0.002
Float 64	Signed 16	$1.98 \cdot 10^{-9}$	$1.14 \cdot 10^{-10}$	1.022	0.003
Float 64	Unsigned 32	$1.29 \cdot 10^{-9}$	$5.18 \cdot 10^{-10}$	1.044	0.021
Float 64	Signed 32	$1.40 \cdot 10^{-9}$	$5.48 \cdot 10^{-10}$	1.036	0.020
Float 64	Float 32	$3.65 \cdot 10^{-10}$	$4.80 \cdot 10^{-10}$	1.054	0.069
Float 64	Float 64	$2.96 \cdot 10^{-14}$	$7.97 \cdot 10^{-16}$	1.561	0.000
Float 64	Complex 32	$2.70 \cdot 10^{-12}$	$2.72 \cdot 10^{-12}$	1.357	0.052
Float 64	Complex 64	$1.11 \cdot 10^{-12}$	$7.02 \cdot 10^{-13}$	1.438	0.033

Tabelle 6: Werte für *Convert Depth*, Teil 2.

Eingabeformat	Ausgabeformat	a	a Std.-Abw.	b	b Std.-Abw.
Complex 32	Unsigned 8	$2.74 \cdot 10^{-9}$	$1.15 \cdot 10^{-10}$	1.008	0.002
Complex 32	Signed 8	$3.27 \cdot 10^{-9}$	$1.79 \cdot 10^{-10}$	0.997	0.003
Complex 32	Unsigned 16	$6.80 \cdot 10^{-9}$	$2.41 \cdot 10^{-10}$	0.999	0.002
Complex 32	Signed 16	$2.89 \cdot 10^{-9}$	$1.49 \cdot 10^{-10}$	1.006	0.003
Complex 32	Unsigned 32	$7.17 \cdot 10^{-10}$	$2.88 \cdot 10^{-10}$	1.072	0.021
Complex 32	Signed 32	$1.08 \cdot 10^{-9}$	$4.44 \cdot 10^{-10}$	1.046	0.021
Complex 32	Float 32	$3.52 \cdot 10^{-10}$	$2.07 \cdot 10^{-10}$	1.089	0.031
Complex 32	Float 64	$7.96 \cdot 10^{-12}$	$7.42 \cdot 10^{-12}$	1.307	0.049
Complex 32	Complex 32	$4.33 \cdot 10^{-14}$	$1.19 \cdot 10^{-15}$	1.541	0.000
Complex 32	Complex 64	$7.55 \cdot 10^{-13}$	$5.37 \cdot 10^{-13}$	1.446	0.037
Complex 64	Unsigned 8	$1.10 \cdot 10^{-9}$	$1.47 \cdot 10^{-10}$	1.064	0.007
Complex 64	Signed 8	$7.79 \cdot 10^{-10}$	$1.35 \cdot 10^{-10}$	1.080	0.009
Complex 64	Unsigned 16	$6.56 \cdot 10^{-10}$	$1.27 \cdot 10^{-10}$	1.094	0.010
Complex 64	Signed 16	$5.50 \cdot 10^{-10}$	$1.24 \cdot 10^{-10}$	1.102	0.012
Complex 64	Unsigned 32	$5.06 \cdot 10^{-10}$	$8.34 \cdot 10^{-11}$	1.110	0.009
Complex 64	Signed 32	$4.02 \cdot 10^{-10}$	$6.30 \cdot 10^{-11}$	1.121	0.008
Complex 64	Float 32	$5.29 \cdot 10^{-11}$	$1.31 \cdot 10^{-11}$	1.212	0.013
Complex 64	Float 64	$6.60 \cdot 10^{-12}$	$4.35 \cdot 10^{-12}$	1.337	0.034
Complex 64	Complex 32	$1.15 \cdot 10^{-11}$	$1.09 \cdot 10^{-11}$	1.283	0.050
Complex 64	Complex 64	$1.40 \cdot 10^{-11}$	$1.43 \cdot 10^{-11}$	1.289	0.053

Tabelle 8: Werte für *Convert Depth*, Teil 3.

Format	a	a Std.-Abw.
Unsigned 8	$8.64 \cdot 10^{-10}$	$3.45 \cdot 10^{-11}$
Signed 8	$8.50 \cdot 10^{-10}$	$1.04 \cdot 10^{-11}$
Unsigned 16	$1.97 \cdot 10^{-9}$	$9.83 \cdot 10^{-12}$
Signed 16	$1.94 \cdot 10^{-9}$	$1.40 \cdot 10^{-11}$
Unsigned 32	$3.08 \cdot 10^{-9}$	$1.41 \cdot 10^{-11}$
Signed 32	$3.09 \cdot 10^{-9}$	$1.19 \cdot 10^{-11}$
Float 32	$3.13 \cdot 10^{-9}$	$1.50 \cdot 10^{-11}$
Float 64	$6.03 \cdot 10^{-9}$	$3.28 \cdot 10^{-11}$

Tabelle 9: Werte für *Extract Channel*.

a	Bedingung
a_2	$\gamma \neq 0.125 \wedge \left(\gamma \leq 0.25 \vee \gamma \in \{0.4, 4\}\right)$
a_3	$\gamma = 0.125$
a_4	$\gamma = 0.5$
a_5	$\gamma = 1$
a_6	$\gamma = 2$
a_7	$\gamma = 4$
a_1	sonst

Tabelle 10: Fallunterscheidungen von *Gamma*.

Format	a_1	a_1 Std.-Abw.	a_2	a_2 Std.-Abw.
Signed 8	$4.81 \cdot 10^{-8}$	$2.61 \cdot 10^{-11}$	$5.17 \cdot 10^{-8}$	$2.52 \cdot 10^{-11}$
Signed 16	$4.83 \cdot 10^{-8}$	$5.28 \cdot 10^{-11}$	$5.20 \cdot 10^{-8}$	$2.17 \cdot 10^{-11}$
Unsigned 32	$4.90 \cdot 10^{-8}$	$8.61 \cdot 10^{-11}$	$5.30 \cdot 10^{-8}$	$9.85 \cdot 10^{-11}$
Signed 32	$4.82 \cdot 10^{-8}$	$7.81 \cdot 10^{-11}$	$5.21 \cdot 10^{-8}$	$3.37 \cdot 10^{-11}$
Float 32	$4.80 \cdot 10^{-8}$	$3.10 \cdot 10^{-11}$	$4.81 \cdot 10^{-8}$	$8.16 \cdot 10^{-11}$
Float 64	$4.88 \cdot 10^{-8}$	$3.38 \cdot 10^{-11}$	$4.89 \cdot 10^{-8}$	$9.09 \cdot 10^{-11}$
	a_3	a_3 Std.-Abw.	a_4	a_4 Std.-Abw.
Signed 8	$1.14 \cdot 10^{-8}$	$1.65 \cdot 10^{-11}$	$1.07 \cdot 10^{-8}$	$1.41 \cdot 10^{-11}$
Signed 16	$1.15 \cdot 10^{-8}$	$1.82 \cdot 10^{-11}$	$1.09 \cdot 10^{-8}$	$2.13 \cdot 10^{-11}$
Unsigned 32	$1.22 \cdot 10^{-8}$	$3.90 \cdot 10^{-11}$	$1.08 \cdot 10^{-8}$	$4.40 \cdot 10^{-11}$
Signed 32	$1.18 \cdot 10^{-8}$	$8.69 \cdot 10^{-11}$	$1.09 \cdot 10^{-8}$	$5.62 \cdot 10^{-11}$
Float 32	$1.02 \cdot 10^{-8}$	$4.35 \cdot 10^{-11}$	$9.57 \cdot 10^{-9}$	$4.34 \cdot 10^{-11}$
Float 64	$1.18 \cdot 10^{-8}$	$6.25 \cdot 10^{-11}$	$1.11 \cdot 10^{-8}$	$5.30 \cdot 10^{-11}$
	a_5	a_5 Std.-Abw.	a_6	a_6 Std.-Abw.
Signed 8	$8.02 \cdot 10^{-9}$	$9.22 \cdot 10^{-12}$	$4.80 \cdot 10^{-8}$	$2.28 \cdot 10^{-11}$
Signed 16	$8.21 \cdot 10^{-9}$	$1.42 \cdot 10^{-11}$	$4.80 \cdot 10^{-8}$	$2.60 \cdot 10^{-11}$
Unsigned 32	$8.75 \cdot 10^{-9}$	$4.62 \cdot 10^{-11}$	$4.83 \cdot 10^{-8}$	$7.73 \cdot 10^{-11}$
Signed 32	$8.28 \cdot 10^{-9}$	$4.69 \cdot 10^{-11}$	$4.76 \cdot 10^{-8}$	$4.01 \cdot 10^{-11}$
Float 32	$4.23 \cdot 10^{-9}$	$4.86 \cdot 10^{-11}$	$4.78 \cdot 10^{-8}$	$2.75 \cdot 10^{-11}$
Float 64	$6.94 \cdot 10^{-9}$	$6.02 \cdot 10^{-11}$	$4.85 \cdot 10^{-8}$	$4.78 \cdot 10^{-11}$
	a_7	a_7 Std.-Abw.		
Signed 8	$4.16 \cdot 10^{-8}$	$9.30 \cdot 10^{-12}$		
Signed 16	$4.17 \cdot 10^{-8}$	$1.49 \cdot 10^{-11}$		
Unsigned 32	$4.25 \cdot 10^{-8}$	$7.47 \cdot 10^{-11}$		
Signed 32	$4.20 \cdot 10^{-8}$	$3.36 \cdot 10^{-11}$		
Float 32	$4.10 \cdot 10^{-8}$	$2.34 \cdot 10^{-11}$		
Float 64	$4.22 \cdot 10^{-8}$	$2.11 \cdot 10^{-11}$		

Tabelle 11: Werte für *Gamma*.

Format	a	a Std.-Abw.
Unsigned 8	$2.04 \cdot 10^{-10}$	$1.38 \cdot 10^{-12}$
Signed 8	$1.89 \cdot 10^{-10}$	$1.17 \cdot 10^{-12}$
Unsigned 16	$2.07 \cdot 10^{-10}$	$1.58 \cdot 10^{-12}$
Signed 16	$2.09 \cdot 10^{-10}$	$1.49 \cdot 10^{-12}$
Unsigned 32	$2.41 \cdot 10^{-10}$	$1.58 \cdot 10^{-12}$
Signed 32	$2.18 \cdot 10^{-10}$	$1.51 \cdot 10^{-12}$
Float 32	$3.12 \cdot 10^{-10}$	$1.26 \cdot 10^{-12}$
Float 64	$3.16 \cdot 10^{-10}$	$1.72 \cdot 10^{-12}$
Complex 32	$6.03 \cdot 10^{-10}$	$3.01 \cdot 10^{-12}$
Complex 64	$5.93 \cdot 10^{-10}$	$1.49 \cdot 10^{-12}$

Tabelle 12: Werte für *Gaussian Blur*.

Format	a_1	a_1 Std.-Abw.	a_2	a_2 Std.-Abw.
Unsigned 8	$2.36 \cdot 10^{-10}$	$4.33 \cdot 10^{-11}$	$8.52 \cdot 10^{-10}$	$6.97 \cdot 10^{-12}$
Signed 8	$2.31 \cdot 10^{-10}$	$1.08 \cdot 10^{-11}$	$8.56 \cdot 10^{-10}$	$4.10 \cdot 10^{-12}$
Unsigned 16	$3.36 \cdot 10^{-10}$	$2.48 \cdot 10^{-11}$	$1.96 \cdot 10^{-9}$	$4.49 \cdot 10^{-12}$
Signed 16	$3.44 \cdot 10^{-10}$	$2.53 \cdot 10^{-11}$	$1.95 \cdot 10^{-9}$	$6.11 \cdot 10^{-12}$
Unsigned 32	$6.86 \cdot 10^{-10}$	$2.13 \cdot 10^{-11}$	$3.12 \cdot 10^{-9}$	$1.09 \cdot 10^{-11}$
Signed 32	$6.52 \cdot 10^{-10}$	$1.91 \cdot 10^{-11}$	$3.14 \cdot 10^{-9}$	$8.00 \cdot 10^{-12}$
Float 32	$6.59 \cdot 10^{-10}$	$1.94 \cdot 10^{-11}$	$3.12 \cdot 10^{-9}$	$8.56 \cdot 10^{-12}$
Float 64	$1.32 \cdot 10^{-9}$	$1.73 \cdot 10^{-11}$	$5.54 \cdot 10^{-9}$	$8.65 \cdot 10^{-12}$

Tabelle 13: Werte für *Join Channels*.

Format	a	a Std.-Abw.	b	b Std.-Abw.
Unsigned 8	$7.14 \cdot 10^{-11}$	$3.69 \cdot 10^{-12}$	1.15023	0.00270394
Signed 8	$5.56 \cdot 10^{-11}$	$4.37 \cdot 10^{-12}$	1.16746	0.00409828
Unsigned 16	$4.26 \cdot 10^{-11}$	$5.49 \cdot 10^{-12}$	1.18039	0.00672224
Signed 16	$4.59 \cdot 10^{-11}$	$6.44 \cdot 10^{-12}$	1.17694	0.00732145
Unsigned 32	$2.42 \cdot 10^{-10}$	$3.02 \cdot 10^{-11}$	1.09089	0.00650261
Signed 32	$2.84 \cdot 10^{-10}$	$3.91 \cdot 10^{-11}$	1.08243	0.00718767
Float 32	$2.29 \cdot 10^{-10}$	$3.52 \cdot 10^{-11}$	1.08911	0.00802407
Float 64	$6.40 \cdot 10^{-10}$	$9.71 \cdot 10^{-11}$	1.06948	0.00792718
Complex 32	$4.89 \cdot 10^{-10}$	$6.58 \cdot 10^{-11}$	1.092	0.00702223
Complex 64	$1.31 \cdot 10^{-09}$	$3.84 \cdot 10^{-10}$	1.07204	0.0153513

Tabelle 14: Werte für *Levels*.

Format	a	a Std.-Abw.
Unsigned 8	$5.59 \cdot 10^{-10}$	$2.47 \cdot 10^{-12}$
Signed 8	$6.03 \cdot 10^{-10}$	$2.19 \cdot 10^{-12}$
Unsigned 16	$5.90 \cdot 10^{-10}$	$2.37 \cdot 10^{-12}$
Signed 16	$5.96 \cdot 10^{-10}$	$2.51 \cdot 10^{-12}$
Unsigned 32	$6.30 \cdot 10^{-10}$	$2.23 \cdot 10^{-12}$
Signed 32	$6.26 \cdot 10^{-10}$	$2.41 \cdot 10^{-12}$
Float 32	$5.68 \cdot 10^{-10}$	$2.48 \cdot 10^{-12}$
Float 64	$9.51 \cdot 10^{-10}$	$4.26 \cdot 10^{-12}$
Complex 32	$1.28 \cdot 10^{-09}$	$3.82 \cdot 10^{-12}$
Complex 64	$2.24 \cdot 10^{-09}$	$6.42 \cdot 10^{-12}$

Tabelle 15: Werte für *Linear Blend*.

Bedingung	Funktionsform
$x = y = 1$	$a \cdot n + 0.00043$
$x \in \mathbb{N} \land y \in \mathbb{N}$	$a \cdot (n + 1) \cdot (s + 1)$
$x = 1 \dot{\vee} y = 1$	$a \cdot n + f \cdot n \cdot (1 - s)^p$
Sonst	$a \cdot n \cdot s$

Tabelle 16: Fallunterscheidungen von *Scale*, wobei weiter oben stehende Einträge höhere Priorität haben, als weiter unten stehende.

Format	a	a Std.-Abw.
Unsigned 8	112.68	0.23
Signed 8	113.96	0.48
Unsigned 16	113.92	0.48
Signed 16	113.52	0.48
Unsigned 32	116.46	0.47
Signed 32	115.13	0.48
Float 32	138.72	0.30
Float 64	137.92	0.30
Complex 32	272.79	1.18
Complex 64	263.36	1.54

Tabelle 17: Werte für *Simple Channel Bilateral*.

φ_j	Speicher MiB	*Cache* MiB	L	E 0	E 1	E 2	B 0.375	B 0.5	Laden s	Ansicht s
	0	0	×	×	×	×	×	×	1.026	0.737
	188	0	✓	✓	✓	✓	✓	✓	0.421	0.054
	0	188	×	×	×	×	×	×	0.417	0.134
1	94	94	✓	×	×	×	✓	×	0.429	0.094
2	94	94	✓	✓	✓	✓	×	×	0.432	0.104
1	94	0	✓	×	×	×	✓	×	0.418	0.090
2	94	0	✓	✓	✓	✓	×	×	0.429	0.142
1	47	47	✓	×	×	×	×	×	0.428	0.139
2	47	47	✓	×	×	×	×	×	0.422	0.135
			L	E 0	E 1	E 2	B 0.375	B 0.5	Laden s	
	0	0	×	×	×	×	×	×	0.733	
	188	0	✓	✓	✓	✓	✓	✓	0.056	
	0	188	×	×	×	×	×	×	0.134	
1	94	94	✓	×	×	×	✓	×	0.087	
2	94	94	✓	✓	✓	✓	×	×	0.105	
1	94	0	✓	×	×	×	✓	×	0.088	
2	94	0	✓	✓	✓	✓	×	×	0.100	
1	47	47	✓	×	×	×	×	×	0.141	
2	47	47	✓	×	×	×	×	×	0.133	

Tabelle 18: Tests für den ersten Graphen.

φ_j	Speicher MiB	*Cache* MiB	L	G	S	L (0,5)	L (-1,1)	A	Laden s	Ansicht s
	0	0	×	×	×	×	×	×	9.460	1.584
	840	0	✓	✓	✓	✓	✓	✓	1.677	0.165
	0	840	×	×	×	×	×	×	8.144	0.932
1	420	420	✓	×	✓	✓	×	×	1.565	0.215
2	420	420	✓	×	✓	✓	×	×	1.563	0.215
	420	0	✓	×	✓	✓	×	×	1.563	0.213
4	420	0	✓	×	✓	✓	×	×	1.569	0.216
1	210	210	✓	×	×	✓	×	×	1.533	0.216
2	210	210	✓	×	×	✓	×	×	1.538	0.222
			L	G	S	L (0,2)	L (-1,1)	A	Laden s	Ansicht s
	0	0	×	×	×	×	×	×	9.917	1.319
	840	0	✓	✓	✓	✓	✓	✓	0.699	0.116
	0	840	×	×	×	×	×	×	8.737	0.884
1	420	420	✓	×	✓	✓	×	×	0.693	0.137
2	420	420	✓	×	✓	✓	×	×	0.692	0.138
1	420	0	✓	×	✓	✓	×	×	0.694	0.139
2	420	0	✓	×	✓	✓	×	×	0.695	0.139
1	210	210	✓	×	×	✓	×	×	1.437	0.139
2	210	210	✓	×	×	✓	×	×	1.457	0.140

Tabelle 19: Tests für den zweiten Graphen.

φ_j	Speicher MiB	*Cache* MiB	L	C 1	C 0	L	G	A	Laden s	Ansicht s
	0	0	×	×	×	×	×	×	29.8834	1.1658
	244	0	✓	✓	✓	✓	✓	✓	1.3254	0.0520
	0	244	×	×	×	×	×	×	29.6446	1.0995
1	122	122	✓	✓	✓	×	✓	×	1.3475	0.0742
2	122	122	✓	✓	✓	×	✓	×	1.3339	0.0709
1	122	0	✓	✓	✓	×	✓	×	1.3309	0.0699
2	122	0	✓	✓	✓	×	✓	×	1.3121	0.0735
1	61	61	✓	✓	✓	×	×	×	29.1005	1.0835
2	61	61	✓	✓	✓	×	×	×	29.1682	1.0818
			L	C 1	C 0	L	G	A	Laden s	Ansicht s
	0	0	×	×	×	×	×	×	30.7841	0.0352
	244	0	✓	✓	✓	✓	✓	✓	1.2414	0.0046
	0	244	×	×	×	×	×	×	30.4655	0.0047
1	122	122	✓	✓	✓	×	✓	×	1.2590	0.0044
2	122	122	✓	✓	✓	×	✓	×	1.2635	0.0075
1	122	0	✓	✓	✓	×	✓	×	1.2623	0.0074
2	122	0	✓	✓	✓	×	✓	×	1.2594	0.0042
1	61	61	✓	✓	✓	×	×	×	30.1958	0.0058
2	61	61	✓	✓	✓	×	×	×	30.1387	0.0057

Tabelle 20: Tests für den dritten Graphen.

φ_j	Speicher MiB	*Cache* MiB	L	G 8	S 0.5	G 4	S 2	A	L	Laden s	Ansicht s
	0	0	×	×	×	×	×	×	×	32.7073	1.4035
	296	0	✓	✓	✓	✓	✓	✓	✓	1.6167	0.0482
	0	296	×	×	×	×	×	×	×	32.3769	1.3543
1	148	148	✓	✓	×	×	×	✓	×	3.5055	0.0574
2	148	148	×	✓	✓	×	×	✓	×	2.9926	0.0573
1	148	0	✓	✓	×	×	×	✓	×	3.5010	0.0578
2	148	0	×	✓	✓	×	×	✓	×	3.0797	0.0577
1	74	74	×	×	✓	×	×	✓	×	17.1897	0.0582
2	74	74	×	×	✓	×	×	✓	×	17.4566	0.0550
			L	G 12	S 0.5	G 6	S 2	A	L	Laden s	Ansicht s
	0	0	×	×	×	×	×	×	×	66.6820	0.4152
	296	0	✓	✓	✓	✓	✓	✓	✓	2.0755	0.0470
	0	296	×	×	×	×	×	×	×	66.6878	0.3881
1	148	148	✓	✓	×	×	×	✓	×	5.1175	0.3911
2	148	148	×	✓	✓	×	×	✓	×	4.4835	0.3408
1	148	0	✓	✓	×	×	×	✓	×	5.1191	0.3850
2	148	0	×	✓	✓	×	×	✓	×	4.4833	0.3399
1	74	74	×	×	✓	×	×	✓	×	33.6225	0.3403
2	74	74	×	×	✓	×	×	✓	×	33.6204	0.3406

Tabelle 21: Tests für den vierten Graphen.